Java OOP Self-Assessment

By:
R.G. (Dick) Baldwin

Java OOP Self-Assessment

By:
R.G. (Dick) Baldwin

Online:
< http://cnx.org/content/col11987/1.1/ >

OpenStax-CNX

Collection structure revised: April 1, 2016
PDF generated: April 8, 2016
For copyright and attribution information for the modules contained in this collection, see p. 342.

Table of Contents

Chapter 1

Ap0005: Preface to OOP Self-Assessment[1]

Revised: Fri Apr 08 22:51:58 CDT 2016

This page is included in the following Books:

- *Java OOP Self-Assessment* [2]
- *Object-Oriented Programming (OOP) with Java* [3]

1.1 Welcome

Welcome to my group of modules titled *Java OOP Self-Assessment* .

This is a self-assessment test designed to help you determine how much you know about object-oriented programming *(OOP)* using Java.

In addition to being a self-assessment test, it is also a major learning tool. Each module consists of about ten to twenty questions with answers and explanations on two or three specific topics. In many cases, the explanations are extensive. You may find those explanations to be very educational in your journey towards understanding OOP using Java.

To give you some idea of the scope of this self-assessment test, when you can successfully answer most of the questions in modules Ap0010 [4] through Ap0140 [5] , your level of knowledge will be roughly equivalent to that of a student who has successfully completed an AP Computer Science course in a U.S. high school up to but not including data structures. This is based on my interpretation of the College Board's Computer Science A Course Description [6] .

1.2 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0005: Preface to OOP Self-Assessment

[1] This content is available online at <http://cnx.org/content/m45252/1.8/>.
[2] http://cnx.org/contents/1CVBGBJj
[3] http://cnx.org/contents/-2RmHFs_
[4] http://cnx.org/content/m45284
[5] http://cnx.org/content/m45302
[6] http://apcentral.collegeboard.com/apc/public/repository/ap-computer-science-course-description.pdf

- File: Ap0005.htm
- Published: 11/28/12

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.

-end-

Chapter 2

Ap0010: Self-assessment, Primitive Types[1]

Revised: Fri Apr 08 23:47:01 CDT 2016

This page is included in the following Books:

- *Java OOP Self-Assessment* [2]
- *Object-Oriented Programming (OOP) with Java* [3]

2.1 Table of Contents

2.2 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.

Questions and answers

The test consists of a series of questions (p. 4) with answers (p. 18) 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.

Programming challenge questions

The module also contains a section titled Programming challenge questions (p. 9) . This section provides specifications for one or more programs that you should be able to write once you understand the answers to all of the questions. *(Note that it is not always possible to confine the programming knowledge requirement*

[1] This content is available online at <http://cnx.org/content/m45284/1.9/>.
[2] http://cnx.org/contents/1CVBGBJj
[3] http://cnx.org/contents/-2RmHFs_

to this and earlier modules. Therefore, you may occasionally need to refer ahead to future modules in order to write the programs.)

Unlike the other questions, solutions are not provided for the *Programming challenge questions* . However, in most cases, the specifications will describe the output that your program should produce.

Listings

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 17) to easily find and view the listings while you are reading about them.

2.3 Questions

2.3.1 Question 1 .

What output is produced by the program in Listing 1 (p. 4) ?

- A. Compiler error
- B. Runtime error
- C. Hello World
- D. Goodbye Cruel World

Listing 1 . Question 1.

```
   public class Ap001{
  public static void main(
                       String args[]){
    new Worker().hello();
  }//end main()
}//end class definition

class Worker{
  public void hello(){
    System.out.println("Hello World");
  }//end hello()
}//end class definition
```

Table 2.1

Answer and Explanation (p. 23)

2.3.2 Question 2 .

What is the largest (algebraic) value of type int?

- A. 32767
- B. 2147483647
- C. -2147483647
- D. -32768

Answer and Explanation (p. 22)

2.3.3 Question 3 .

What is the smallest (algebraic) value of type int?

- A. -2147483648
- B. -2147483647
- C. 32767
- D. -32768

Answer and Explanation (p. 22)

2.3.4 Question 4 .

What two values are displayed by the program in Listing 2 (p. 5) ?

- A. -2147483648
- B. 1.7976931348623157E308
- C. -2147483647
- D. 4.9E-324

Listing 2 . Question 4.

```
    public class Ap003{
  public static void main(
                        String args[]){
    new Worker().printDouble();
  }//end main()
}//end class definition

class Worker{
  public void printDouble(){
    System.out.println(
                     Double.MAX_VALUE);
    System.out.println(
                     Double.MIN_VALUE);
  }//end printDouble()
}//end class definition
```

Table 2.2

Answer and Explanation (p. 21)

2.3.5 Question 5 .

What output is produced by the program in Listing 3 (p. 6) ?

- A. true
- B. false
- C. 1
- D. 0

Listing 3 . Question 5.

```
    public class Ap004{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  private boolean myVar;
  public void printBoolean(){
    System.out.println(myVar);
  }//end printBoolean()
}//end class definition
```

Table 2.3

Answer and Explanation (p. 21)

2.3.6 Question 6 .

What output is produced by the program shown in Listing 4 (p. 6) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 4 . Question 6.

```
    public class Ap005{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  public void printBoolean(){
    boolean myVar;
    System.out.println(myVar);
  }//end printBoolean()
}//end class definition
```

Table 2.4

Answer and Explanation (p. 20)

2.3.7 Question 7 .

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

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 5 . Question 7.

```
    public class Ap006{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  public void printBoolean(){
    boolean myVar = true;
    myVar = false;
    System.out.println(myVar);
  }//end printBoolean()
}//end class definition
```

Table 2.5

Answer and Explanation (p. 20)

2.3.8 Question 8 .

The plus (+) character can be used to perform numeric addition in Java. What output is produced by the program shown in Listing 6 (p. 8) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. 2
- E. 1

Listing 6 . Question 8.

continued on next page

```
    public class Ap007{
  public static void main(
                        String args[]){
    new Worker().printBoolean();
  }//end main()
}//end class definition

class Worker{
  public void printBoolean(){
    boolean myVar = true;
    System.out.println(1 + myVar);
  }//end printBoolean()
}//end class definition
```

Table 2.6

Answer and Explanation (p. 19)

2.3.9 Question 9 .

The plus (+) character can be used to perform numeric addition in Java. What output is produced by the program shown in Listing 7 (p. 8) ?

- A. Compiler Error
- B. Runtime Error
- C. 6
- D. 6.0

Listing 7 . Question 9.

```
    public class Ap008{
  public static void main(
                        String args[]){
    new Worker().printMixed();
  }//end main()
}//end class definition

class Worker{
  public void printMixed(){
    double x = 3;
    int y = 3;
    System.out.println(x+y);
  }//end printMixed()
}//end class definition
```

Table 2.7

Answer and Explanation (p. 19)

2.3.10 Question 10 .

The slash (/) character can be used to perform numeric division in Java. What output is produced by the program shown in Listing 8 (p. 9) ?

- A. Compiler Error
- B. Runtime Error
- C. 0.33333334
- D. 0.3333333333333333

Listing 8 . Question 10.

```
   public class Ap009{
  public static void main(
                        String args[]){
    new Worker().printMixed();
  }//end main()
}//end class definition

class Worker{
  public void printMixed(){
    System.out.println(1.0/3);
  }//end printMixed()
}//end class definition
```

Table 2.8

Answer and Explanation (p. 18)

2.4 Programming challenge questions

2.4.1 Question 11

Write the program described in Listing 9 (p. 10) .

Listing 9 . Question 11.

```
/*File Ap0010a1.java Copyright 2012, R.G.Baldwin

Instructions to student:
This program refuses to compile without errors.

Make the necessary corrections to cause the program to
compile and run successfully to produce the output shown
below:

ITSE
2321
**********************************************************/
public class Ap0010a1{
  public static void main(String args[]){
    System.out.println("ITSE");
    new Worker().doIt();
  }//end main()
}//end class definition
//=======================================================//

Class Worker{
  public void doIt(){
    System.out.println("2321");
  }//end doIt()
}//end class definition
//=======================================================//
```

Table 2.9

2.4.2 Question 12

Write the program described in Listing 10 (p. 11) .

Listing 10 . Question 12.

```
/*File Ap0010b1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Receives and displays an incoming parameter of type int.

The result should be similar to the following but the
values should be different each time the program is
run.

484495695
484495695
**********************************************************/
//Student is not expected to understand import directives
// at this point.
import java.util.Random;
import java.util.Date;

public class Ap0010b1{
  public static void main(String args[]){
    //Create a random number for testing. Student is not
    // expected to understand how this works at this point.
    Random random = new Random(new Date().getTime());
    int intVar = random.nextInt();

    //Student should understand the following
    int var = intVar;
    System.out.println(var);
    new Worker().doIt(var);
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------//
}//end class definition
//========================================================//
```

Table 2.10

2.4.3 Question 13

Write the program described in Listing 11 (p. 12) .

Listing 11 . Question 13.

```
/*File Ap0010c1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that returns the largest value of type
int as type float.

The result should be 2.14748365E9
**************************************************************/
public class Ap0010c1{
  public static void main(String args[]){
    float val = new Worker().doIt();
    System.out.println(val);
  }//end main()
}//end class definition
//============================================================//

class Worker{
  //----------------------------------------------------------//
  //Insert the method named doIt between these lines.
  //----------------------------------------------------------//
}//end class definition
//============================================================//
```

Table 2.11

2.4.4 Question 14

Write the program described in Listing 12 (p. 13) .

Listing 12 . Question 14.

```
/*File Ap0010d1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Receives an incoming parameter of type double.
2. Converts that value to type int.
3. Returns the int

The result should be similar to the following but the
values should be different each time the program is
run.

6.672032181818181E8
667203218
**********************************************************/
//Student is not expected to understand import directives
// at this point.
import java.util.Random;
import java.util.Date;

public class Ap0010d1{
  public static void main(String args[]){
    //Create a random number for testing. Student is not
    // expected to understand how this works at this point.
    Random random = new Random(new Date().getTime());
    int intVar = random.nextInt();

    //Student should understand the following
    double var = intVar/1.1;
    System.out.println(var);
    System.out.println(new Worker().doIt(var));
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

Table 2.12

2.4.5 Question 15

Write the program described in Listing 13. (p. 14)

Listing 13 . Question 15.

```
/*File Ap0010e1.java Copyright 2012, R.G.Baldwin

Instructions to student:
This program refuses to compile without errors.

Make the necessary corrections to cause the program to
compile and run successfully to produce an output similar
to that shown below. Note that the values should be
different each time the program is
run.

-1.30240579E8
-1.30240579E8
**********************************************************/
//Student is not expected to understand import directives
// at this point.
import java.util.Random;
import java.util.Date;

public class Ap0010e1{
  public static void main(String args[]){
    //Create a random number for testing. Student is not
    // expected to understand how this works at this point.
    Random random = new Random(new Date().getTime());
    double doubleVar = random.nextInt()/1.0;

    //Student should understand the following
    double var = doubleVar;
    System.out.println(doubleVar);
    new Worker().doIt(doubleVar);
  }//end main()
}//end class definition
//========================================================//

class Worker{
  public void doIt(double val){
    int var = val;
    System.out.println(var);
  }//end doIt()
}//end class definition
//========================================================//
```

Table 2.13

2.4.6 Question 16

Write the program described in Listing 14 (p. 15) .

Listing 14 . Question 16.

```
/*File Ap0010f1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code shown below, modify the
code in the method named doIt so that the program
displays

3.3333333333333335 instead of 3

Then modify the method again so that the program displays

3.3333333 instead of 3
**********************************************************/
public class Ap0010f1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=========================================================//

class Worker{
  public void doIt(){
    System.out.println(10/3);
  }//end doIt()
}//end class definition
//=========================================================//
```

Table 2.14

2.4.7 Question 17

Write the program described in Listing 15 (p. 16) .

Listing 15 . Question 17.

continued on next page

```
/*File Ap0010g1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code shown below, modify the
code in the method named doIt so that the program
displays

2048 instead of 2730

Did you notice anything particularly interesting about the
values involved?
**********************************************************/
public class Ap0010g1{
  public static void main(String args[]){
    new Worker().doIt(16384);
  }//end main()
}//end class definition
//========================================================//

class Worker{
  public void doIt(int val){
    System.out.println(val/6);
  }//end doIt()
}//end class definition
//========================================================//
```

Table 2.15

2.4.8 Question 18

Write the program described in Listing 16 (p. 17) .

Listing 16 . Question 18.

```
/*File Ap0010h1.java Copyright 2012, R.G.Baldwin

Instructions to student:
This program refuses to compile without errors.

Make the necessary corrections to cause the program to
compile and run successfully to produce the output shown
below:

false
**********************************************************/

public class Ap0010h1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=======================================================//

class Worker{
  public void doIt(){
    boolean var;
    System.out.println(var);
  }//end doIt()
}//end class definition
//=======================================================//
```

Table 2.16

2.5 Listings

- Listing 16 (p. 17) . Question 18.
- Listing 17 (p. 24) . Answer 1.

2.6 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0010: Self-assessment, Primitive types
- File: Ap0010.htm
- Originally published: December 17, 2001
- Published at cnx.org: 12/01/12

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.

2.7 Answers

2.7.1 Answer 10 .

D. 0.3333333333333333

2.7.1.1 Explanation 10

Divide floating type by integer type

This program divides the literal floating value of 1.0 by the literal integer value of 3 *(no decimal point is specified in the integer literal value)* .

Automatic conversion from narrow to wider type

To begin with, whenever division is performed between a floating type and an integer type, the integer type is automatically converted *(sometimes called promoted)* to a floating type and floating arithmetic is performed.

What is the actual floating type, float or double?

The real question here is, what is the type of the literal shown by 1.0 *(with a decimal point separating the 1 and the 0)* . Is it a **double** or a **float** ?

Type double is the default

By default, a literal floating value is treated as a **double** .

The result is type double

Consequently, this program divides a **double** type by an integer type, producing a result of type **double** . This is somewhat evident in the output, which shows about 17 digits plus a decimal point in the result. *(Recall that the maximum value for a float shown earlier had only about eight digits plus the decimal point and the exponent.)*

How can you create literals of type float?

What if you don't want your literal floating value to be treated as a **double** , but would prefer that it be treated as a **float** instead.

You can usually force this to be the case by adding a suffix of either F or f to the end of the literal value *(as in 1.0F)* . If you were to modify this program to cause it to divide 1.0F by 3, the output would be 0.33333334 with only nine digits in the result.

Back to Question 10 (p. 9)

2.7.2 Answer 9 .

D. 6.0

2.7.2.1 Explanation 9

Declare and initialize two local variables

This program declares and initializes two local variables, one of type **double** and the other of type **int** . Each variable is initialized with the integer value 3.

Automatic conversion to floating type double

However, before the value of 3 is stored in the **double** variable, it is automatically converted to type **double** .

Automatic conversion in mixed-type arithmetic

Numeric addition is performed on the two variables. Whenever addition is performed between a floating type and an integer type, the integer type is automatically converted to a floating type and floating arithmetic is performed.

A floating result

This produces a floating result. When this floating result is passed to the **println** method for display, a decimal point and a zero are displayed to indicate a floating type, even though in this case, the fractional part of the result is zero.

Back to Question 9 (p. 8)

2.7.3 Answer 8 .

A. Compiler Error

2.7.3.1 Explanation 8

Initialize boolean variable to true

This program declares and initializes a **boolean** variable with the value *true* . Then it attempts to add the literal value 1 to the value stored in the **boolean** variable named **myVar** .

Arithmetic with boolean values is not allowed

As mentioned earlier, unlike C++, **boolean** types in Java cannot participate in arithmetic expressions.

Therefore, this program will not compile. The compiler error produced by this program under JDK 1.3 reads partially as follows:

```
Ap007.java:13: operator + cannot be applied to int,boolean
System.out.println(1 + myVar);
```

Table 2.17

Back to Question 8 (p. 7)

2.7.4 Answer 7 .

D. false

2.7.4.1 Explanation 7

Format for variable initialization

This program declares a local **boolean** variable and initializes it to the value *true* . All variables, local or otherwise, can be initialized in this manner provided that the expression on the right of the equal sign evaluates to a value that is assignment compatible with the type of the variable. *(I will have more to say about assignment compatibility in a future module)* .

Value is changed before display

However, before calling the **println** method to display the initial value of the variable, the program uses the assignment operator (=) to assign the value *false* to the variable. Thus, when it is displayed, the value is *false* .

Back to Question 7 (p. 7)

2.7.5 Answer 6 .

A. Compiler Error

2.7.5.1 Explanation 6

A local boolean variable

In this program, the primitive variable named **myVar** is a local variable belonging to the method named **printBoolean** .

Local variables are not automatically initialized

Unlike instance variables, if you fail to initialize a local variable, the variable is not automatically initialized.

Cannot access value from uninitialized local variable

If you attempt to access and use the value from an uninitialized local variable before you assign a value to it, you will get a compiler error. The compiler error produced by this program under JDK 1.3 reads partially as follows:

```
Ap005.java:13: variable myVar might not have been initialized
System.out.println(myVar);
```

Table 2.18

Must initialize or assign value to all local variables

Thus, the programmer is responsible for either initializing all local variables, or assigning a value to them before attempting to access their value with code later in the program. The good news is that the system won't allow you to compute with garbage left over in memory occupied by variables, either local variables or member variables.

Back to Question 6 (p. 6)

2.7.6 Answer 5 .

B. false

2.7.6.1 Explanation 5

The boolean type

In this program, the primitive variable named **myVar** is an instance variable of the type **boolean** .

What is an instance variable?

An instance variable is a variable that is declared inside a class, outside of all methods and constructors of the class, and is not declared static. Every object instantiated from the class has one. That is why it is called an instance variable.

Cannot use uninitialized variables in Java

One of the great things about Java is that it is not possible to make the mistake of using variables that have not been initialized.

Can initialize when declared

All Java variables can be initialized when they are declared.

Member variables are automatically initialized

If the programmer doesn't initialize the variables declared inside the class but outside of a method *(often referred to as member variables as opposed to local variables)* , they are automatically initialized to a default value. The default value for a **boolean** variable is false.

Did you know the boolean default value?

I wouldn't be overly concerned if you had selected the answer A. true, because I wouldn't necessarily expect you to memorize the default initialization value.

Great cause for concern

However, I would be very concerned if you selected either C. 1 or D. 0.

Java has a true boolean type

Unlike C++, Java does not represent true and false by the numeric values of 1 and 0. *(At least the numeric values that represent true and false are not readily accessible by the programmer.)*

Thus, you cannot include boolean types in arithmetic expressions, as is the case in C++.

Back to Question 5 (p. 5)

2.7.7 Answer 4 .

- B. 1.7976931348623157E308
- D. 4.9E-324

2.7.7.1 Explanation 4

Floating type versus integer type

If you missed this one, shame on you!

I didn't expect you to memorize the maximum and minimum values represented by the floating type double, but I did expect you to be able to distinguish between the display of a floating value and the display of an integer value.

Both values are positive

Note that both of the values given above are positive values.

Unlike the integer types discussed earlier, the constants named **MAX_VALUE** and **MIN_VALUE** don't represent the ends of a signed number range for type **double** . Rather, they represent the largest and smallest *(non-zero)* values that can be expressed by the type.

An indication of granularity

MIN_VALUE is an indication of the degree of granularity of values expressed as type **double** . Any **double** value can be treated as either positive or negative.

Two floating types are available

Java provides two floating types: **float** and **double** . The **double** type provides the greater range, or to use another popular terminology, it is the *wider* of the two.

What is the value range for a float?

In case you are interested, using the same syntax as above, the value range for type float is from 1.4E-45 to 3.4028235E38

Double is often the default type

There is another thing that is significant about type **double** . In many cases where a value is automatically converted to a floating type, it is converted to type double rather than to type float. This will come up in future modules.

Back to Question 4 (p. 5)

2.7.8 Answer 3 .

A. -2147483648

2.7.8.1 Explanation 3

Could easily have guessed

As a practical matter, you had one chance in two of guessing the correct answer to this question, already having been given the value of the largest algebraic value for type int.

And the winner is ...

Did you answer B. -2147483647? – WRONG

If so, you may be wondering why the most negative value isn't equal to the negative version of the most positive value?

A twos-complement characteristic

Without going into the details of why, it is a well-known characteristic of binary twos-complement notation that the value range extends one unit further in the negative direction than in the positive direction.

What about the other two values?

Do the values of -32768 and 32767 in the set of multiple-choice answers to this question represent anything in particular?

Yes, they represent the extreme ends of the value range for a 16-bit binary number in twos-complement notation.

Does Java have a 16-bit integer type?

Just in case you are interested, the **short** type in Java is represented in 16-bit binary twos-complement signed notation, so this is the value range for type short.

What about type byte?

Similarly, a value of type **byte** is represented in 8-bit binary twos-complement signed notation, with a value range extending from -128 to 127.

Back to Question 3 (p. 5)

2.7.9 Answer 2 .

B. 2147483647

2.7.9.1 Explanation 2

First question on types

This is the first question on Java types in this group of self-assessment modules.

32-bit signed twos-complement integers

In Java, values of type **int** are stored as 32-bit signed integers in twos-complement notation.

Can you calculate the values?

There are no unsigned integer types in Java, as there are in C++. If you are handy with binary notation, you could calculate the largest positive value that can be stored in 32 bits in twos-complement notation.

See documentation for the Integer class

Otherwise, you can visit the documentation [4] for the **Integer** class, which provides a symbolic constant *(public static final variable)* named **MAX_VALUE** . The description of MAX_VALUE reads as follows:

"The largest value of type int. The constant value of this field is 2147483647."

Back to Question 2 (p. 4)

2.7.10 Answer 1 .

C. Hello World

2.7.10.1 Explanation 1

The answer to this first question is intended to be easy. The purpose of the first question is to introduce you to the syntax that will frequently be used for program code in this group of self-assessment modules.

The controlling class and the main method

In this example, the class named **Ap001** is the *controlling class* . It contains a method named **main** , with a signature that matches the required signature for the **main** method. When the user executes this program, the Java virtual machine automatically calls the method named **main** in the controlling class.

Create an instance of Worker

The **main** method uses the **new** operator along with the default constructor for the class named **Worker** to create a new instance of the class named **Worker** *(an object of the Worker class)* . This is often referred to as instantiating an object.

A reference to an anonymous object

The combination of the **new** operator and the default constructor for the **Worker** class returns a reference to the new object. In this case, the object is instantiated as an *anonymous object* , meaning that the object's reference is not saved in a named reference variable. *(Instantiation of a non-anonymous object will be illustrated later.)*

Call hello method on Worker object

The main method contains a single executable statement.

As soon as the reference to the new object is returned, the single statement in the **main** method calls the **hello** method on that reference.

Output to standard output device

This causes the **hello** method belonging to the new object *(of the class named **Worker**)* to execute. The code in the **hello** method calls the **println** method on the **static** variable of the **System** class named **out** .

Lots of OOP embodied in the hello method

I often tell my students that I can tell a lot about whether a student really understands object-oriented programming in Java by asking them to explain everything that they know about the following statement:

System.out.println("Hello World");

I would expect the answer to consume about ten to fifteen minutes if the student really understands Java OOP.

The one-minute version

[4] http://cnx.org/content/m45117

When the virtual machine starts a Java application running, it automatically instantiates an I/O stream object linked to the standard output device *(normally the screen)* and stores a reference to that object in the **static** variable named **out** belonging to the class named **System** .

Call the println instance method on out

Calling the **println** method on that reference, and passing a literal string *("Hello World")* to that method causes the contents of the literal **String** object to be displayed on the standard output device.

Display Hello World on the screen

In this case, this causes the words *Hello World* to be displayed on the standard output device. This is the answer to the original question.

Time for main method to terminate

When the **hello** method returns, the **main** method has nothing further to do, so it terminates. When the **main** method terminates in a Java application, the application terminates and returns control to the operating system. This causes the system prompt to reappear.

A less-cryptic form

A less cryptic form of this program is shown in Listing 17 (p. 24) .

Listing 17 . Answer 1.

```
    public class Ap002{
  public static void main(
                        String args[]){
    Worker refVar = new Worker();
    refVar.hello();
  }//end main()
}//end class definition
class Worker{

  public void hello(){
    System.out.println("Hello World");
  }//end hello()
}//end class definition
```

Table 2.19

Decompose single statement into two statements

In this version, the single statement in the earlier version of the **main** method is replaced by two statements.

A non-anonymous object

In the class named **Ap002** shown in Listing 2 (p. 5) , the object of the class named **Worker** is not instantiated anonymously. Rather, a new object of the **Worker** class is instantiated and the object's reference is stored in *(assigned to)* the named reference variable named **refVar** .

Call hello method on named reference

Then the **hello** method is called on that reference in a separate statement.

Produces the same result as before

The final result is exactly the same as before. The only difference is that a little more typing is required to create the source code for the second version.

Will often use anonymous objects

In order to minimize the amount of typing required, I will probably use the anonymous form of instantiation whenever appropriate in these modules.

Now that you understand the framework ...

Now that you understand the framework for the program code, I can present more specific questions. Also, the explanations will usually be shorter.

Back to Question 1 (p. 4)

-end-

Chapter 3

Ap0020: Self-assessment, Assignment and Arithmetic Operators[1]

3.1 Table of Contents

3.2 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.

Questions and answers

The test consists of a series of questions (p. 28) with answers (p. 46) 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.

Programming challenge questions

The module also contains a section titled Programming challenge questions (p. 39) . This section provides specifications for one or more programs that you should be able to write once you understand the answers to all of the questions. *(Note that it is not always possible to confine the programming knowledge requirement to this and earlier modules. Therefore, you may occasionally need to refer ahead to future modules in order to write the programs.)*

Unlike the other questions, solutions are not provided for the *Programming challenge questions* . However, in most cases, the specifications will describe the output that your program should produce.

Listings

I recommend that you open another copy of this document in a separate browser window and use the links to under Listings (p. 45) to easily find and view the listings while you are reading about them.

[1] This content is available online at <http://cnx.org/content/m45286/1.6/>.

3.3 Questions

3.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 28) ?

- A. Compiler Error
- B. Runtime Error
- C. 3.0
- D. 4.0
- E. 7.0

Listing 1 . Listing for Question 1.

```
   public class Ap010{
  public static void main(
                        String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    double myVar;
    myVar = 3.0;
    myVar += 4.0;
    System.out.println(myVar);
  }//end doAsg()
}//end class definition
```

Table 3.1

Answer and Explanation (p. 54)

3.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 29) ?

- A. Compiler Error
- B. Runtime Error
- C. 2.147483647E9
- D. 2.14748365E9

Listing 2 . Listing for Question 2.

continued on next page

```
    public class Ap011{
  public static void main(
                        String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    double myDoubleVar;
    //Integer.MAX_VALUE = 2147483647
    int myIntVar = Integer.MAX_VALUE;
    myDoubleVar = myIntVar;
    System.out.println(myDoubleVar);
  }//end doAsg()
}//end class definition
```

Table 3.2

Answer and Explanation (p. 53)

3.3.3 Question 3

What output is produced by the following program?

- A. Compiler Error
- B. Runtime Error
- C. 2147483647
- D. 2.147483647E9

Listing 3 . Listing for Question 3.

```
    public class Ap012{
  public static void main(
                        String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    //Integer.MAX_VALUE = 2147483647
    double myDoubleVar =
                    Integer.MAX_VALUE;
    int myIntVar;
    myIntVar = myDoubleVar;
    System.out.println(myIntVar);
  }//end doAsg()
}//end class definition
```

Table 3.3

Answer and Explanation (p. 53)

3.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 31) ?

- A. Compiler Error
- B. Runtime Error
- C. 2147483647
- D. 2.147483647E9

Listing 4 . Listing for Question 4.

```
    public class Ap013{
  public static void main(
                        String args[]){
    new Worker().doAsg();
  }//end main()
}//end class definition

class Worker{
  public void doAsg(){
    //Integer.MAX_VALUE = 2147483647
    double myDoubleVar =
                    Integer.MAX_VALUE;
    int myIntVar;
    myIntVar = (int)myDoubleVar;
    System.out.println(myIntVar);
  }//end doAsg()
}//end class definition
```

Table 3.4

Answer and Explanation (p. 52)

3.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 32) ?

- A. Compiler Error
- B. Runtime Error
- C. 4.294967294E9
- D. 4294967294

Listing 5 . Listing for Question 5.

```
    public class Ap014{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    //Integer.MAX_VALUE = 2147483647
    int myIntVar = Integer.MAX_VALUE;
    System.out.println(2.0 * myIntVar);
  }//end doMixed()
}//end class definition
```

Table 3.5

Answer and Explanation (p. 51)

3.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 32) ?

- A. Compiler Error
- B. Runtime Error
- C. 2147483649
- D. -2147483647

Listing 6 . Listing for Question 6.

```
    public class Ap015{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    //Integer.MAX_VALUE = 2147483647
    int myVar01 = Integer.MAX_VALUE;
    int myVar02 = 2;
    System.out.println(
                    myVar01 + myVar02);
  }//end doMixed()
}//end class definition
```

Table 3.6

Answer and Explanation (p. 51)

3.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 33) ?

- A. Compiler Error
- B. Runtime Error
- C. 33.666666
- D. 34
- E. 33

Listing 7 . Listing for Question 7.

```
    public class Ap016{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    int myVar01 = 101;
    int myVar02 = 3;
    System.out.println(
                      myVar01/myVar02);
  }//end doMixed()
}//end class definition
```

Table 3.7

Answer and Explanation (p. 50)

3.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 34) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. 11

Listing 8 . Listing for Question 8.

```
    public class Ap017{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    int myVar01 = 11;
    int myVar02 = 0;
    System.out.println(
                      myVar01/myVar02);
  }//end doMixed()
}//end class definition
```

Table 3.8

Answer and Explanation (p. 49)

3.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 34) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. 11

Listing 9 . Listing for Question 9.

```
    public class Ap018{
  public static void main(
                        String args[]){
    new Worker().doMixed();
  }//end main()
}//end class definition

class Worker{
  public void doMixed(){
    double myVar01 = 11;
    double myVar02 = 0;
    System.out.println(
                      myVar01/myVar02);
  }//end doMixed()
}//end class definition
```

Table 3.9

Answer and Explanation (p. 49)

3.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 35) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. -2

Listing 10 . Listing for Question 10.

```
    public class Ap019{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    int myVar01 = -11;
    int myVar02 = 3;
    System.out.println(
                       myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

Table 3.10

Answer and Explanation (p. 48)

3.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 36) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 11

Listing 11 . Listing for Question 11.

```
    public class Ap020{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    int myVar01 = -11;
    int myVar02 = 0;
    System.out.println(
                      myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

Table 3.11

Answer and Explanation (p. 48)

3.3.12 Question 12

What output is produced by the program shown in Listing 12 (p. 36) ?

- A. Compiler Error
- B. Runtime Error
- C. -0.010999999999999996
- D. 0.010999999999999996

Listing 12 . Listing for Question 12.

```
    public class Ap021{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    double myVar01 = -0.11;
    double myVar02 = 0.033;
    System.out.println(
                      myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

Table 3.12

Answer and Explanation (p. 48)

3.3.13 Question 13

What output is produced by the program shown in Listing 13 (p. 37) ?

- A. Compiler Error
- B. Runtime Error
- C. 0.0
- D. 1.5499999999999996

Listing 13 . Listing for Question 13.

```
    public class Ap022{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    double myVar01 = 15.5;
    double myVar02 = 1.55;
    System.out.println(
                       myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

Table 3.13

Answer and Explanation (p. 47)

3.3.14 Question 14

What output is produced by the program shown in Listing 14 (p. 38) ?

- A. Compiler Error
- B. Runtime Error
- C. Infinity
- D. NaN

Listing 14 . Listing for Question 14.

```
    public class Ap023{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    double myVar01 = 15.5;
    double myVar02 = 0.0;
    System.out.println(
                      myVar01 % myVar02);
  }//end doMod()
}//end class definition
```

Table 3.14

Answer and Explanation (p. 47)

3.3.15 Question 15

What output is produced by the program shown in Listing 15 (p. 38) ?

- A. Compiler Error
- B. Runtime Error
- C. -3 2
- D. -3 -2

Listing 15 . Listing for Question 15.

```
    public class Ap024{
  public static void main(
                        String args[]){
    new Worker().doMod();
  }//end main()
}//end class definition

class Worker{
  public void doMod(){
    int x = 11;
    int y = -3;
    System.out.println(
                      x/y + " " + x % y);
  }//end doMod()
}//end class definition
```

Table 3.15

Answer and Explanation (p. 46)

3.4 Programming challenge questions

3.4.1 Question 16

Write the program described in Listing 16 (p. 39) .

Listing 16 . Listing for Question 16.

```
/*File Ap0020a1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the proper use of the combined
arithmetic/assignment operators such as the following
operators:

+=
*=

**********************************************************/
public class Ap0020a1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=========================================================//

class Worker{
  //-------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //-------------------------------------------------------//
}//end class definition
//=========================================================//
```

Table 3.16

3.4.2 Question 17

Write the program described in Listing 17 (p. 40) .

Listing 17 . Listing for Question 17.

```
/*File Ap0020b1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the detrimental impact of integer arithmetic
overflow.

*******************************************************/
public class Ap0020b1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=====================================================//

class Worker{
  //-----------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //-----------------------------------------------------//
}//end class definition
//=====================================================//
```

Table 3.17

3.4.3 Question 18

Write the program described in Listing 18 (p. 41) .

Listing 18 . Listing for Question 18.

```
/*File Ap0020c1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of integer truncation that
occurs with integer division.

*****************************************************/
public class Ap0020c1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=====================================================//

class Worker{
  //-----------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //-----------------------------------------------//
}//end class definition
//=====================================================//
```

Table 3.18

3.4.4 Question 19

Write the program described in Listing 19 (p. 42) .

Listing 19 . Listing for Question 19.

```
/*File Ap0020d1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of double divide by zero.
2. Illustrates the effect of integer divide by zero.

*******************************************************/
public class Ap0020d1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=====================================================//

class Worker{
  //-----------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //-----------------------------------------------------//
}//end class definition
//=====================================================//
```

Table 3.19

3.4.5 Question 20

Write the program described in Listing 20 (p. 43) .

Listing 20 . Listing for Question 20.

```
/*File Ap0020e1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of the modulus operation with
integers.

**************************************************/
public class Ap0020e1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//=======================================================//

class Worker{
  //-----------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //-----------------------------------------------------//
}//end class definition
//=======================================================//
```

Table 3.20

3.4.6 Question 21

Write the program described in Listing 21 (p. 44) .

Listing 21 . Listing for Question 21.

```
/*File Ap0020f1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the effect of the modulus operation with
doubles.

**************************************************/
public class Ap0020f1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//======================================================//

class Worker{
  //------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //------------------------------------------------//
}//end class definition
//======================================================//
```

Table 3.21

3.4.7 Question 22

Write the program described in Listing 22 (p. 45) .

Listing 22 . Listing for Question 22.

```
/*File Ap0020g1.java Copyright 2012, R.G.Baldwin

Instructions to student:
Beginning with the code fragment shown below, write a
method named doIt that:
1. Illustrates the concatenation of the following strings
separated by space characters.

"This"
"is"
"fun"

Cause your program to produce the following output:
This
is
fun
This is fun
**********************************************************/
public class Ap0020g1{
  public static void main(String args[]){
    new Worker().doIt();
  }//end main()
}//end class definition
//========================================================//

class Worker{
  //--------------------------------------------------------//
  //Student: insert the method named doIt between these
  // lines.
  //--------------------------------------------------------//
}//end class definition
//========================================================//
```

Table 3.22

3.5 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. 28) . Listing for Question 1.
- Listing 2 (p. 29) . Listing for Question 2.
- Listing 3 (p. 30) . Listing for Question 3.
- Listing 4 (p. 31) . Listing for Question 4.
- Listing 5 (p. 32) . Listing for Question 5.
- Listing 6 (p. 32) . Listing for Question 6.
- Listing 7 (p. 33) . Listing for Question 7.

3.6 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0020: Self-assessment, Assignment and Arithmetic Operators
- File: Ap0020.htm
- Originally published: January 7, 2002
- Published at cnx.org: 12/01/12
- Revised: 12/03/14

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.

3.7 Answers

3.7.1 Answer 15

C. -3 2

3.7.1.1 Explanation 15

String concatenation

This program uses **String** concatenation, which has not been previously discussed in this group of self-assessment modules.

In this case, the program executes both an integer divide operation and an integer modulus operation, using **String** concatenation to display both results on a single line of output.

Quotient = -3 with a remainder of 2

Thus, the displayed result is the integer quotient followed by the remainder.

What is String concatenation?

If either operand of the plus (+) operator is of type **String** , no attempt is made to perform arithmetic addition. Rather, the other operand is converted to a **String** , and the two strings are concatenated.

A space character, " "

The string containing a space character (" ") in this expression appears as the right operand of one plus operator and as the left operand of the other plus operator.

If you already knew about **String** concatenation, you should have been able to figure out the correct answer to the question on the basis of the answers to earlier questions in this module.

Back to Question 15 (p. 38)

3.7.2 Answer 14

D. NaN

3.7.2.1 Explanation 14

Floating modulus operation involves floating divide

The modulus operation with floating operands and 0.0 as the right operand produces **NaN** , which stands for *Not a Number* .

What is the actual value of Not a Number?

A symbolic constant that is accessible as **Double.NaN** specifies the value that is returned in this case.

Be careful what you try to do with it. It has some peculiar behavior of its own.

Back to Question 14 (p. 37)

3.7.3 Answer 13

D. 1.5499999999999996

3.7.3.1 Explanation 13

A totally incorrect result

Unfortunately, due to floating arithmetic inaccuracy, the modulus operation in this program produces an entirely incorrect result.

The result should be 0.0, and that is the result produced by my hand calculator.

Terminates one step too early

However, this program terminates the repetitive subtraction process one step too early and produces an incorrect remainder.

Be careful

This program is included here to emphasize the need to be very careful how you interpret the result of performing modulus operations on floating operands.

Back to Question 13 (p. 37)

3.7.4 Answer 12

C. -0.010999999999999996

3.7.4.1 Explanation 12

Modulus operator can be used with floating types

In this case, the program returns the remainder that would be produced by dividing a double value of -0.11 by a double value of 0.033 and terminating the divide operation at the beginning of the fractional part of the quotient.

Say that again

Stated differently, the result of the modulus operation is the remainder that results after

- subtracting the right operand from the left operand an integral number of times, and
- terminating the repetitive subtraction process when the result of the subtraction is less than the right operand

Modulus result is not exact

According to my hand calculator, taking into account the fact that the left operand is negative, this operation should produce a modulus result of -0.011. As you can see, the result produced by the application of the modulus operation to floating types is not exact.

Back to Question 12 (p. 36)

3.7.5 Answer 11

B. Runtime Error

3.7.5.1 Explanation 11

Integer modulus involves integer divide

The modulus operation with integer operands involves an integer divide.

Therefore, it is subject to the same kind of problem as an ordinary integer divide when the right operand has a value of zero.

Program produces a runtime error

In this case, the program produced a runtime error that terminated the program. The error produced by JDK 1.3 is as follows:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Worker.doMod(Ap020.java:14)
    at Ap020.main(Ap020.java:6)
```

Table 3.23

Dealing with the problem

As with integer divide, you can either test the right operand for a zero value before performing the modulus operation, or you can deal with the problem after the fact using try-catch.

Back to Question 11 (p. 35)

3.7.6 Answer 10

D. -2

3.7.6.1 Explanation 10

What is a modulus operation?

In elementary terms, we like to say that the modulus operation returns the remainder that results from a divide operation.

In general terms, that is true.

Some interesting behavior

However, the modulus operation has some interesting behaviors that are illustrated in this and the next several questions.

This program returns the modulus of -11 and 3, with -11 being the left operand.

What is the algebraic sign of the result?

Here is a rule:

The result of the modulus operation takes the sign of the left operand, regardless of the sign of the quotient and regardless of the sign of the right operand. In this program, that produced a result of -2.

Changing the sign of the right operand would ***not*** have changed the sign of the result.

Exercise care with sign of modulus result

Thus, you may need to exercise care as to how you interpret the result when you perform a modulus operation having a negative left operand.

Back to Question 10 (p. 35)

3.7.7 Answer 9

C. Infinity

3.7.7.1 Explanation 9

Floating divide by zero

This program attempts to divide the **double** value of 11 by the **double** value of zero.

No runtime error with floating divide by zero

In the case of floating types, an attempt to divide by zero does not produce a runtime error. Rather, it returns a value that the **println** method interprets and displays as Infinity.

What is the actual value?

The actual value returned by this program is provided by a **static final** variable in the **Double** class named **POSITIVE_INFINITY** .

(There is also a value for NEGATIVE_INFINITY, which is the value that would be returned if one of the operands were a negative value.)

Is this a better approach?

Is this a better approach than throwing an exception as is the case for integer divide by zero?

I will let you be the judge of that.

In either case, you can test the right operand before the divide to assure that it isn't equal to zero.

Cannot use exception handling in this case

For floating divide by zero, you cannot handle the problem by using try-catch.

However, you can test the result following the divide to see if it is equal to either of the infinity values mentioned above.

Back to Question 9 (p. 34)

3.7.8 Answer 8

B. Runtime Error

3.7.8.1 Explanation 8

Dividing by zero

This program attempts to divide the **int** value of 11 by the **int** value of zero.

Integer divide by zero is not allowed

This produces a runtime error and terminates the program.

The runtime error is as follows under JDK 1.3:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
     at Worker.doMixed(Ap017.java:14)
     at Ap017.main(Ap017.java:6)
```

Table 3.24

Two ways to deal with this sort of problem

One way is to test the right operand before each divide operation to assure that it isn't equal to zero, and to take appropriate action if it is.

A second *(possibly preferred)* way is to use exception handling and surround the divide operation with a *try* block, followed by a *catch* block for the type

```
    java.lang.ArithmeticException.
```

Table 3.25

The code in the catch block can be designed to deal with the problem if it occurs. *(Exception handling will be discussed in a future module.)*

Back to Question 8 (p. 33)

3.7.9 Answer 7

E. 33

3.7.9.1 Explanation 7

Integer truncation

This program illustrates the integer truncation that results when the division operator is applied to operands of the integer types.

The result of simple long division

We all know that when we divide 101 by 3, the result is 33.666666 with the sixes extending out to the limit of our arithmetic accuracy.

The result of rounding

If we round the result to the next closest integer, the result is 34.

Integer division does not round

However, when division is performed using operands of integer types in Java, the fractional part is simply discarded *(not rounded)* .

The result is the whole number result without regard for the fractional part or the remainder.

Thus, with integer division, 101/3 produces the integer value 33.

If either operand is a floating type ...
If either operand is one of the floating types,

- the integer operand will be converted to the floating type,
- the result will be of the floating type, and
- the fractional part of the result will be preserved to some degree of accuracy

Back to Question 7 (p. 33)

3.7.10 Answer 6

D. -2147483647

3.7.10.1 Explanation 6

Danger, integer overflow ahead!
This program illustrates a very dangerous situation involving arithmetic using operands of integer types. This situation involves a condition commonly known as *integer overflow* .

The good news
The good news about doing arithmetic using operands of integer types is that as long as the result is within the allowable value range for the wider of the integer types, the results are exact *(floating arithmetic often produces results that are not exact)* .

The bad news
The bad news about doing arithmetic using operands of integer types is that when the result is not within the allowable value range for the wider of the integer types, the results are garbage, having no usable relationship to the correct result *(floating arithmetic has a high probability of producing approximately correct results, even though the results may not be exact).*

For this specific case ...
As you can see by the answer to this question, when a value of 2 was added to the largest positive value that can be stored in type **int** , the incorrect result was a very large negative value.
The result is simply incorrect. *(If you know how to do binary arithmetic, you can figure out how this happens.)*

No safety net in this case – just garbage
Furthermore, there was no compiler error and no runtime error. The program simply produced an incorrect result with no warning.
You need to be especially careful when writing programs that perform arithmetic using operands of integer types. Otherwise, your programs may produce incorrect results.
Back to Question 6 (p. 32)

3.7.11 Answer 5

C. 4.294967294E9

3.7.11.1 Explanation 5

Mixed-type arithmetic
This program illustrates the use of arithmetic operators with operands of different types.

Declare and initialize an int
The method named **doMixed** declares a local variable of type int named **myIntVar** and initializes it with the largest positive value that can be stored in type **int** .

Evaluate an arithmetic expression
An arithmetic expression involving **myIntVar** is evaluated and the result is passed as a parameter to the **println** method where it is displayed on the computer screen.

Multiply by a literal double value

The arithmetic expression uses the multiplication operator (*) to multiply the value stored in **myIntVar** by 2.0 *(this literal operand is type* ***double*** *by default)* .

Automatic conversion to wider type

When arithmetic is performed using operands of different types, the type of the operand of the narrower type is automatically converted to the type of the operand of the wider type, and the arithmetic is performed on the basis of the wider type.

Result is of the wider type

The type of the result is the same as the wider type.

In this case ...

Because the left operand is type **double** , the **int** value is converted to type **double** and the arithmetic is performed as type **double** .

This produces a result of type **double** , causing the floating value 4.294967294E9 to be displayed on the computer screen.

Back to Question 5 (p. 31)

3.7.12 Answer 4

C. 2147483647

3.7.12.1 Explanation 4

Uses a cast operator

This program, named **Ap013.java** , differs from the earlier program named **Ap012.java** in one important respect.

This program uses a *cast operator* to force the compiler to allow a narrowing conversion in order to assign a **double** value to an **int** variable.

The cast operator

The statement containing the cast operator is shown below for convenient viewing.

```
myIntVar = (int)myDoubleVar;
```

Table 3.26

Syntax of a cast operator

The cast operator consists of the name of a type contained within a pair of matching parentheses.

A unary operator

The cast operator always appears to the left of an expression whose type is being converted to the type specified by the cast operator.

Assuming responsibility for potential problems

When dealing with primitive types, the cast operator is used to notify the compiler that the programmer is willing to assume the risk of a possible loss of precision in a narrowing conversion.

No loss of precision here

In this case, there was no loss in precision, but that was only because the value stored in the **double** variable was within the allowable value range for an **int** .

In fact, it was the largest positive value that can be stored in the type **int** . Had it been any larger, a loss of precision would have occurred.

More on this later ...

I will have quite a bit more to say about the cast operator in future modules. I will also have more to say about the use of the assignment operator in conjunction with the non-primitive types.

Back to Question 4 (p. 30)

3.7.13 Answer 3

A. Compiler Error

3.7.13.1 Explanation 3

Conversion from double to int is not automatic

This program attempts to assign a value of type **double** to a variable of type **int** .

Even though we know that the specific double value involved would fit in the **int** variable with no loss of precision, the conversion from **double** to **int** is not a *widening* conversion.

This is a narrowing conversion

In fact, it is a *narrowing* conversion because the allowable value range for an **int** is less than the allowable value range for a **double** .

The conversion is not allowed by the compiler. The following compiler error occurs under JDK 1.3:

```
    Ap012.java:16: possible loss of precision
found   : double
required: int
    myIntVar = myDoubleVar    myIntVar = myDoubleVar;
```

Table 3.27

Back to Question 3 (p. 29)

3.7.14 Answer 2

C. 2.147483647E9

3.7.14.1 Explanation 2

Declare a double

The method named **doAsg** first declares a local variable of type **double** named **myDoubleVar** without providing an initial value.

Declare and initialize an int

Then it declares an **int** variable named **myIntVar** and initializes its value to the integer value 2147483647 *(you learned about Integer.MAX_VALUE in an earlier module)* .

Assign the int to the double

Following this, the method assigns contents of the **int** variable to the **double** variable.

An assignment compatible conversion

This is an *assignment compatible* conversion. In particular, the integer value of 2147483647 is automatically converted to a **double** value and stored in the **double** variable.

The **double** representation of that value is what appears on the screen later when the value of **myDoubleVar** is displayed.

What is an assignment compatible conversion?

An assignment compatible conversion for the primitive types occurs when the required conversion is a *widening* conversion.

What is a widening conversion?

A widening conversion occurs when the allowable value range of the type of the left operand of the assignment operator is greater than the allowable value range of the right operand of the assignment operator.

A double is wider than an int

Since the allowable value range of type **double** is greater than the allowable value range of type **int** , assignment of an **int** value to a **double** variable is allowed, with conversion from **int** to **double** occurring automatically.

A safe conversion

It is also significant to note that there is no loss in precision when converting from an **int** to a **double** .

An unsafe but allowable conversion

However, a loss of precision may occur when an **int** is assigned to a **float** , or when a **long** is assigned to a **double** .

What would a float produce ?

The value of 2.14748365E9 shown for selection D is what you would see for this program if you were to change the **double** variable to a **float** variable. *(Contrast this with 2147483647 to see the loss of precision.)*

Widening is no guarantee that precision will be preserved

The fact that a type conversion is a widening conversion does not guarantee that there will be no loss of precision in the conversion. It simply guarantees that the conversion will be allowed by the compiler. In some cases, such as that shown above (p. 54) , an assignment compatible conversion can result in a loss of precision, so you always need to be aware of what you are doing.

Back to Question 2 (p. 28)

3.7.15 Answer 1

E. 7.0

3.7.15.1 Explanation 1

Declare but don't initialize a double variable

The method named **doAsg** begins by declaring a **double** variable named **myVar** without initializing it.

Use the simple assignment operator

The simple assignment operator (=) is then used to assign the **double** value 3.0 to the variable. Following the execution of that statement, the variable contains the value 3.0.

Use the arithmetic/assignment operator

The next statement uses the combined arithmetic/assignment operator (+=) to add the value 4.0 to the value of 3.0 previously assigned to the variable. The following two statements are functionally equivalent:

```
myVar += 4.0;

myVar = myVar + 4.0;
```

Table 3.28

Two statements are equivalent

This program uses the first statement listed above. If you were to replace the first statement with the second statement, the result would be the same.

In this case, either statement would add the value 4.0 to the value of 3.0 that was previously assigned to the variable named **myVar** , producing the sum of 7.0. Then it would assign the sum of 7.0 back to the variable. When the contents of the variable are then displayed, the result is that 7.0 appears on the computer screen.

No particular benefit

To the knowledge of this author, there is no particular benefit to using the combined arithmetic/assignment notation other than to reduce the amount of typing required to produce the source code. However, if you ever plan to interview for a job as a Java programmer, you need to know how to use the combined version.

Four other similar operators

Java support several combined operators. Some involve arithmetic and some involve other operations such as bit shifting. Five of the combined operators are shown below. These five all involve arithmetic.

- +=
- -=
- *=
- /=
- %=

In all five cases, you can construct a functionally equivalent arithmetic and assignment statement in the same way that I constructed the functionally equivalent statement for += above.

Back to Question 1 (p. 28)

-end-

Chapter 4

Ap0030: Self-assessment, Relational Operators, Increment Operator, and Control Structures[1]

4.1 Table of Contents

4.2 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. 72) to easily find and view the listings while you are reading about them.

4.3 Questions

4.3.1 Question 1 .

Given: The use of **String** concatenation in the argument list of the call to the **println** method in the program shown in Listing 1 (p. 58) will cause seven items to be displayed on the screen, separated by spaces.

True or False? The program produces the output shown below:

[1] This content is available online at <http://cnx.org/content/m45287/1.4/>.

```
    false true false false true true false
```

Table 4.1

Listing 1 . Listing for Question 1.

```
    public class Ap025{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    int a = 1, b = 2, c = 3, d = 2;

    System.out.println(
                      (a == b) + " " +
                      (b == d) + " " +
                      (b != d) + " " +
                      (c < a) + " " +
                      (b <= d) + " " +
                      (c > d) + " " +
                      (a >= c));
  }//end doRelat()
}//end class definition
```

Table 4.2

Answer and Explanation (p. 81)

4.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 59) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 2 . Listing for Question 2.

```
    public class Ap026{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

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

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

Table 4.3

Answer and Explanation (p. 80)

4.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 60) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 3 . Listing for Question 3.

```
    public class Ap027{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = new Dummy();
    System.out.println(x.equals(y));
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

Table 4.4

Answer and Explanation (p. 80)

4.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 61) ?

- A. Compiler Error
- B. Runtime Error
- C. true false
- D. false true
- E. true true

Listing 4 . Listing for Question 4.

```
    public class Ap028{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = x;
    System.out.println(
         (x == y) + " " + x.equals(y));
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

Table 4.5

Answer and Explanation (p. 79)

4.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 62) ?

- A. Compiler Error
- B. Runtime Error
- C. true
- D. false

Listing 5 . Listing for Question 5.

```
    public class Ap029{
  public static void main(
                        String args[]){
    new Worker().doRelat();
  }//end main()
}//end class definition

class Worker{
  public void doRelat(){
    Dummy x = new Dummy();
    Dummy y = new Dummy();
    System.out.println(x > y);
  }//end doRelat()
}//end class definition

class Dummy{
  int x = 5;
  double y = 5.5;
  String z = "A String Object";
}//end class Dummy
```

Table 4.6

Answer and Explanation (p. 79)

4.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 63) ?

- A. Compiler Error
- B. Runtime Error
- C. 5 5 8.3 8.3
- D. 6 4 9.3 7.300000000000001

Listing 6 . Listing for Question 6.

```
    public class Ap030{
  public static void main(
                        String args[]){
    new Worker().doIncr();
  }//end main()
}//end class definition

class Worker{
  public void doIncr(){
    int w = 5, x = 5;
    double y = 8.3, z = 8.3;

    w++;
    x--;
    y++;
    z--;

    System.out.println(w + "  " +
                       x + "  " +
                       y + "  " +
                       z);
  }//end doIncr()
}//end class definition
```

Table 4.7

Answer and Explanation (p. 78)

4.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 64) ?

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

Listing 7 . Listing for Question 7.

```
    public class Ap031{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 5, y = 6;
    if(x - y){
      System.out.println("Hello");
    }//end if
  }//end doIf()
}//end class definition
```

Table 4.8

Answer and Explanation (p. 78)

4.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 65) ?

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

Listing 8 . Listing for Question 8.

```
    public class Ap032{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 5, y = 6;
    if(x < y){
      System.out.print("Hello ");
    }//end if
    System.out.println("World");
  }//end doIf()
}//end class definition
```

Table 4.9

Answer and Explanation (p. 77)

4.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 66) ?

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

Listing 9 . Listing for Question 9.

```
    public class Ap033{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 5, y = 6;
    if(x == y){
      System.out.println(
                        "Hello World");
    }else{
      System.out.println(
                      "Goodbye World");
    }//end else
  }//end doIf()
}//end class definition
```

Table 4.10

Answer and Explanation (p. 76)

4.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 67) ?

- A. Compiler Error
- B. Runtime Error
- C. x = 4
- D. x = 5
- E. x = 6
- F. x != 4,5,6
- G. None of the above

Listing 10 . Listing for Question 10.

```
    public class Ap034{
  public static void main(
                        String args[]){
    new Worker().doIf();
  }//end main()
}//end class definition

class Worker{
  public void doIf(){
    int x = 2;
    if(x == 4){
      System.out.println("x = 4");
    }else if (x == 5){
      System.out.println("x = 5");
    }else if (x == 6){
      System.out.println("x = 6");
    }else{
      System.out.println("x != 4,5,6");
    }//end else
  }//end doIf()
}//end class definition
```

Table 4.11

Answer and Explanation (p. 76)

4.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 68) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 4
- D. 1 2 3 4 5
- E. None of the above

Listing 11 . Listing for Question 11.

```
    public class Ap035{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    int cnt = 0;
    while(cnt<5){
      cnt++;
      System.out.print(cnt + " ");
      cnt++;
    }//end while loop
    System.out.println("");
  }//end doLoop()
}//end class definition
```

Table 4.12

Answer and Explanation (p. 76)

4.3.12 Question 12

What output is produced by the program shown in Listing 12 (p. 69) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 4 5
- D. 1 2 3 4 5 5
- E. None of the above

Listing 12 . Listing for Question 12.

```
    public class Ap036{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    int cnt;
    for(cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
    }//end for loop
    System.out.println(cnt + " ");
  }//end doLoop()
}//end class definition
```

Table 4.13

Answer and Explanation (p. 75)

4.3.13 Question 13

What output is produced by the program shown in Listing 13 (p. 70) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 4 5
- D. 1 2 3 4 5 5
- E. None of the above

Listing 13 . Listing for Question 13.

```
    public class Ap037{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    for(int cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
    }//end for loop
    System.out.println(cnt + " ");
  }//end doLoop()
}//end class definition
```

Table 4.14

Answer and Explanation (p. 74)

4.3.14 Question 14

What output is produced by the program shown in Listing 14 (p. 71) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 3
- D. 0 1 2 3 4
- E. None of the above

Listing 14 . Listing for Question 14.

```
    public class Ap037{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public double doLoop(){
    for(int cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
      if(cnt == 3){
        System.out.println(cnt);
        return cnt;
      }//end if
    }//end for loop
    //return 3.5;
  }//end doLoop()
}//end class definition
```

Table 4.15

Answer and Explanation (p. 73)

4.3.15 Question 15

What output is produced by the program shown in Listing 15 (p. 72) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 1 2 3 3
- D. 0 1 2 3 4
- E. None of the above

Listing 15 . Listing for Question 15.

```
    public class Ap038{
  public static void main(
                        String args[]){
    new Worker().doLoop();
  }//end main()
}//end class definition

class Worker{
  public void doLoop(){
    for(int cnt = 0; cnt < 5; cnt++){
      System.out.print(cnt + " ");
      if(cnt == 3){
        System.out.println(cnt);
        return;
      }//end if
    }//end for loop
  }//end doLoop()
}//end class definition
```

Table 4.16

Answer and Explanation (p. 73)

4.4 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.

4.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0030: Self-assessment, Relational Operators, Increment Operator, and Control Structures
- File: Ap0030.htm
- Originally published: January, 2002
- Published at cnx.org: 12/02/12
- Revised: 12/03/14

4.6 Answers

4.6.1 Answer 15

C. 0 1 2 3 3

4.6.1.1 Explanation 15

No return statement is required

A method with a signature that specifies a **void** return type does not require a **return** statement.

However, such a method may contain a **return** statement, provided that it is terminated immediately with a semicolon *(no expression between the word return and the semicolon)* .

(Every method whose return type is not void must contain at least one return statement.)

Multiple return statements are allowed

Any method may contain any number of **return** statements provided that they make sense from a syntax viewpoint, and provided the expression *(or lack thereof)* between the word **return** and the semicolon evaluates to the type specified in the method signature *(or a type that will be automatically converted to the type specified in the method signature)* .

A return statement terminates a method immediately

Whenever the execution stream encounters any **return** statement, the method is terminated immediately, and control is returned to the method that called that method.

Back to Question 15 (p. 71)

4.6.2 Answer 14

A. Compiler Error

4.6.2.1 Explanation 14

Missing return statement

This program produces the following compiler error under JDK 1.3:

```
  Ap037.java:11: missing return statement
public int doLoop(){
```

Table 4.17

Even though this program contains a **return** statement inside the **for** loop, it is still necessary to place a **return** statement at the end of the method to satisfy the compiler. *(The one shown in the code is a comment.)*

The method named **doLoop** must return a value of type **double** . Apparently the compiler assumes that the **return** statement inside the **for** loop may never be executed *(although that isn't true in this case)* .

Both of the **return** statements must return a value that satisfies the **double** type requirement given in the method signature.

Returning a value of type **int** in the **for** loop will satisfy the type requirement because type **int** will be automatically converted to type **double** as it is returned. *(Conversion from type **int** to type **double** is a widening conversion.)*

Back to Question 14 (p. 70)

4.6.3 Answer 13

A. Compiler Error

4.6.3.1 Explanation 13

The scope of a local variable

In general, the scope of a local variable extends from the point at which it is declared to the curly brace that signals the end of the block in which it is declared.

This applies to for loop in an interesting way

While it is allowable to declare a variable within the first clause of a **for** loop, the scope of that variable is limited to the block of code contained in the loop structure.

The variable cannot be accessed outside the loop.

Attempts to access variable out of scope

This program attempts to access the value of the variable named **cnt** after the loop terminates.

The program displays the following compiler error under JDK 1.3. This error results from the attempt to display the value of the counter after the loop terminates.

```
    Ap037.java:15: cannot resolve symbol
symbol  : variable cnt
location: class Worker
    System.out.println(cnt + " ");
```

Table 4.18

Back to Question 13 (p. 69)

4.6.4 Answer 12

C. 0 1 2 3 4 5

4.6.4.1 Explanation 12

A simple for loop structure

This program illustrates a simple **for** loop that displays the value of its counter using a call to the **print** method inside the loop.

After the loop terminates, the program displays the value of the counter one last time using a call to **println** .

Three clauses separated by semicolons

The first line of a **for** loop structure always contains three clauses separated by semicolons.

The first and third clauses may be empty, but the semicolons are required in any case.

The first clause ...

The first clause is executed once and only once at the beginning of the loop.

It can contain just about any valid Java expression.

It can even contain more than one expression with the individual expression separated by commas.

When the first clause contains more than one expression separated by commas, the expressions are evaluated in left-to-right order.

The second clause

The second clause is a conditional clause. It must contain an expression that returns a **boolean** value.

(Actually, this clause can also be empty, in which case it is apparently assumed to be true. This leads to an infinite loop unless there is some code inside the loop to terminate it, perhaps by executing a return or a break statement.)

An entry-condition loop

The **for** loop is an entry condition loop, meaning that the conditional expression is evaluated once immediately after the first clause is executed, and once per iteration thereafter.

Behavior of the for loop

If the conditional expression returns true, the block of code following the closing parenthesis is executed.

If it returns false, the block of code is skipped, and control passes to the first executable statement following the block of code.

(For the case where the block contains only one statement, the matching curly brackets can be omitted.)

The third clause

The third clause can contain none, one, or more valid expressions separated by commas.

If there are more than one, they are evaluated in left-to-right order.

When they are evaluated

The expressions in the third clause are evaluated once during each iteration.

However, it is very important to remember that despite the physical placement of the clause in the first line, the expressions in the third clause are not evaluated until after the code in the block has been evaluated.

Typically an update clause

The third clause is typically used to update a counter, but this is not a technical requirement.

This clause can be used for just about any purpose.

However, the counter must be updated somewhere within the block of code or the loop will never terminate.

(Stated differently, something must occur within the block of code to eventually cause the conditional expression to evaluate to false. Otherwise, the loop will never terminate on its own. However, it is possible to execute a ***return*** *or* ***break*** *within the block to terminate the loop.)*

Note the first output value for this program

Because the update in the third clause is not executed until after the code in the block has been executed, the first value displayed (p. 75) by this program is the value zero.

Back to Question 12 (p. 68)

4.6.5 Answer 11

E. None of the above

4.6.5.1 Explanation 11

And the answer is ...

The output produced by this program is:

1 3 5

A simple while loop

This program uses a simple **while** loop to display the value of a counter, once during each iteration.

Behavior of a while loop

As long as the relational expression in the conditional clause returns true, the block of code immediately following the conditional clause is executed.

When the relational expression returns false, the block of code following the conditional clause is skipped and control passes to the next executable statement following that block of code.

An entry-condition loop

The **while** loop is an entry condition loop, meaning that the test is performed once during each iteration before the block of code is executed.

If the first test returns false, the block of code is skipped entirely.

An exit-condition loop

There is another loop, known as a **do-while** loop, that performs the test after the block of code has been executed once. This guarantees that the block of code will always be executed at least once.

Just to make things interesting ...

Two statements using the increment operator were placed inside the loop in this program.

Therefore, insofar as the conditional test is concerned, the counter is being incremented by twos. This causes the output to display the sequence 1 3 5.

Nested while loops

The **while** loop control structure can contain loops nested inside of loops, which leads to some interesting behavior.

Back to Question 11 (p. 67)

4.6.6 Answer 10

F. x != 4,5,6

4.6.6.1 Explanation 10

A multiple-choice structure

This is a form of control structure that is often used to make logical decisions in a *multiple-choice* sense.

This is a completely general control structure. It can be used with just about any type of data.

A switch structure

There is a somewhat more specialized, control structure named **switch** that can also be used to make decisions in a multiple choice sense under certain fairly restrictive conditions.

However, the structure shown in this program can always be used to replace a switch. Therefore, I find that I rarely use the **switch** structure, opting instead for the more general form of multiple-choice structure.

Back to Question 10 (p. 66)

4.6.7 Answer 9

D. Goodbye World

4.6.7.1 Explanation 9

An if-else control structure

This program contains a simple **if-else** control structure.

Behavior of if-else structure

If the expression in the conditional clause returns true, the block of code following the conditional clause is executed, and the block of code following the word **else** is skipped.

If the expression in the conditional clause returns false, the block of code following the conditional clause is skipped, and the block of code following the word **else** is executed.

This program executes the else block

In this program, the expression in the conditional clause returns false.

Therefore, the block of code following the word **else** is executed, producing the words *Goodbye World* on the computer screen.

Can result in very complex structures

While the structure used in this program is relatively simple, it is possible to create very complex control structures by nesting additional **if-else** structures inside the blocks of code.

Back to Question 9 (p. 65)

4.6.8 Answer 8

D. Hello World

4.6.8.1 Explanation 8

A simple if statement

This program contains a simple **if** statement that

- uses a relational expression
- to return a value of type **boolean** inside its conditional clause

Tests for x less than y

The relational expression tests to determine if the value of the variable named **x** is less than the value of the variable named **y** .

Since the value of **x** is 5 and the value of **y** is 6, this relational expression returns true.

Behavior of an if statement

If the expression in the conditional clause returns true, the block of code following the conditional clause is executed

What is a block of code?

A block of code is one or more statements surrounded by matching curly brackets.

For cases like this one where the block includes only one statement, the curly brackets can be omitted. However, I prefer to put them there anyway. They don't cause any harm and help me avoid programming errors if I come back later and add more statements to the body of the **if** statement.

Display the word Hello

In this program, execution of the code in the block causes the **print** method to be called and the word *Hello* to be displayed followed by a space, but without a newline following the space.

What if the conditional clause returns false?

If the expression in the conditional clause returns false, the block of code following the conditional clause is bypassed.

(That is not the case in this program.)

After the if statement ...

After the **if** statement is executed in this program, the **println** method is called to cause the word *World* to be displayed on the same line as the word *Hello* .

Back to Question 8 (p. 64)

4.6.9 Answer 7

A. Compiler Error

4.6.9.1 Explanation 7

Not the same as C and C++

Unlike C and C++, which can use an integer numeric expression in the conditional clause of an **if** statement, Java requires the conditional clause of an **if** statement to contain an expression that will return a **boolean** result.

Bad conditional expression

That is not the case in this program, and the following compiler error occurs under JDK 1.3:

```
    Ap031.java:13: incompatible types
found   : int
required: boolean
    if(x - y){
```

Table 4.19

Back to Question 7 (p. 63)

4.6.10 Answer 6

D. 6 4 9.3 7.300000000000001

4.6.10.1 Explanation 6

Postfix increment and decrement operators

This program illustrates the use of the increment (++) and decrement (–) operators in their postfix form.

Behavior of increment operator

Given a variable **x** , the following two statements are equivalent:

```
    x++;
x = x + 1;
```

Table 4.20

Behavior of decrement operator

Also, the following two statements are equivalent:

```
    x--;
x = x - 1;
```

Table 4.21

Prefix and postfix forms available

These operators have both a prefix form and a postfix form.

Can be fairly complex

It is possible to construct some fairly complex scenarios when using these operators and combining them into expressions.

In these modules ...

In this group of self-assessment modules, the increment and decrement operators will primarily be used to update control variables in loops.

Inaccurate results

Regarding the program output, you will note that there is a little arithmetic inaccuracy when this program is executed using JDK 1.3. *(The same is still true with JDK version 1.7.)*

Ideally, the output value 7.300000000000001 should simply be 7.3 without the very small additional fractional part, but that sort of thing often happens when using floating types.

Back to Question 6 (p. 62)

4.6.11 Answer 5

A. Compiler Error

4.6.11.1 Explanation 5

Cannot use > with reference variables

The only relational operator that can be applied to reference variables is the == operator.

As discussed in the previous questions, even then it can only be used to determine if two reference variables refer to the same object.

This program produces the following compiler error under JDK 1.3:

```
Ap029.java:14: operator > cannot be applied to Dummy,Dummy
  System.out.println(x > y);
```

Table 4.22

Back to Question 5 (p. 61)

4.6.12 Answer 4

E. true true

4.6.12.1 Explanation 4

Two references to the same object

In this case, the reference variables named **x** and **y** both refer to the same object. Therefore, when tested for equality, using either the == operator or the default **equals** method, the result is true.

Back to Question 4 (p. 60)

4.6.13 Answer 3

D. false

4.6.13.1 Explanation 3

Read question 2

In case you skipped it, you need to read the explanation for the answer to Question 2 (p. 58) before reading this explanation.

Objects appear to be equal

These two objects are of the same type and contain the same values. Why are they reported as not being equal?

Did not override the equals method

When I defined the class named **Dummy** used in the programs for Question 2 (p. 58) and Question 3 (p. 59) , I did not override the method named **equals** .

Therefore, my class named **Dummy** simply inherited the default version of the method named **equals** that is defined in the class named **Object** .

Default behavior of equals method

The default version of the **equals** method behaves essentially the same as the == operator.

That is to say, the inherited default version of the **equals** method will return true if the two objects being compared are actually the same object, and will return false otherwise.

As a result, this program displays false.

Overridden equals is required for valid testing

If you want to be able to determine if two objects instantiated from a class that you define are "equal", you must override the inherited **equals** method for your new class. You cannot depend on the inherited version of the **equals** method to do that job for you.

Overriding may not be easy

That is not to say that overriding the **equals** method is easy. In fact, it may be quite difficult in those cases where the class declares instance variables that refer to other objects. In this case, it may be necessary to test an entire tree of objects for equality.

Back to Question 3 (p. 59)

4.6.14 Answer 2

D. false

4.6.14.1 Explanation 2

Use of the == operator with references to objects

This program illustrates an extremely important point about the use of the == operator with objects and reference variables containing references to objects.

You cannot determine...

You cannot determine if two objects are "equal" by applying the == operator to the reference variables containing references to those objects.

Rather, that test simply determines if two reference variables refer to the same object.

Two references to the same object

Obviously, if there is only one object, referred to by two different reference variables, then it is "equal" to itself.

Objects of same type containing same instance values

On the other hand, two objects of the same type could contain exactly the same data values, but this test would not indicate that they are "equal." *(In fact, that is the case in this program.)*

So, how do you test two objects for equal?

In order to determine if two objects are "equal", you must devise a way to compare the types of the two objects and actually compare the contents of one object to the contents of the other object. Fortunately, there is a standard framework for doing this.

The equals method

In particular, the class named **Object** defines a default version of a method named **equals** that is inherited by all other classes.

Class author can override the equals method

The intent is that the author of a new class can override the **equals** method so that it can be called to determine if two objects instantiated from that class are "equal."

What does "equal" mean for objects?

Actually, that is up to the author of the class to decide.

After having made that decision, the author of the class writes that behavior into her overridden version of the method named **equals** .

Back to Question 2 (p. 58)

4.6.15 Answer 1

The answer is True.

4.6.15.1 Explanation 1

Not much to explain here

There isn't much in the way of an explanation to provide for this program.

Evaluate seven relational expressions

Each of the seven relational expressions in the argument list for the **println** method is evaluated and returns either true or false as a **boolean** value.

Concatenate the individual results, separated by a space

The seven **boolean** results are concatenated, separated by space characters, and displayed on the computer screen.

Brief description of the relational operators

Just in case your aren't familiar with the relational operators, here is a brief description.

Each of these operators returns the **boolean** value true if the specified condition is met. Otherwise, it returns false.

```
== Left operand equals the right operand

!= Left operand is not equal to the right operand

< Left operand is less than the right operand

<= Left operand is less than or equal to the right operand

> Left operand is greater than the right operand

>= Left operand is greater than or equal to the right operand
```

Table 4.23

Back to Question 1 (p. 57)
-end-

Chapter 5

Ap0040: Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method[1]

5.1 Table of Contents

5.2 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. 93) to easily find and view the listings while you are reading about them.

5.3 Questions

5.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 84) ?

- A. Compiler Error
- B. Runtime Error
- C. A

[1] This content is available online at <http://cnx.org/content/m45260/1.6/>.

- 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
```

Table 5.1

Answer and Explanation (p. 102)

5.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 85) ?

- 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
```

Table 5.2

Answer and Explanation (p. 101)

5.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 86) ?

- 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
```

Table 5.3

Answer and Explanation (p. 99)

5.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 86) ?

- 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
```

Table 5.4

Answer and Explanation (p. 98)

5.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 88) ?

- 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
```

Table 5.5

Answer and Explanation (p. 98)

5.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 89) ?

- 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 = -3.499999999999;

    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
```

Table 5.6

Answer and Explanation (p. 98)

5.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 90) ?

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

Listing 7 . Listing for Question 7.

continued on next page

```
    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
```

Table 5.7

Answer and Explanation (p. 96)

5.3.8 Question 8

Which of the following best approximates the output from the program shown in Listing 8 (p. 91) ?

- 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
```

Table 5.8

Answer and Explanation (p. 95)

5.3.9 Question 9

Which of the following best approximates the output from the program shown in Listing 9 (p. 92) ?

- 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
```

Table 5.9

Answer and Explanation (p. 95)

5.3.10 Question 10

Which of the following best approximates the output from the program shown in Listing 10 (p. 93) ? *(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
```

Table 5.10

Answer and Explanation (p. 94)

5.4 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.

5.5 Miscellaneous

This section contains a variety of miscellaneous information.

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: 12/02/12
• Revised: 12/03/14

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

5.6 Answers

5.6.1 Answer 10

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

5.6.1.1 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. 94) , but in milliseconds:

1354491300781

Back to Question 10 (p. 92)

5.6.2 Answer 9

D. Joe Age = 35 Weight = 162.5

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. 90) .

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.

Overridden *toString* method

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

Table 5.11

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. 91)

5.6.3 Answer 8

C. Dummy@273d3c

5.6.3.1 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. 90)

5.6.4 Answer 7

C. 3.5/9/true

5.6.4.1 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. 89)

5.6.5 Answer 6

C. 4 -3

5.6.5.1 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. 87))* .

Back to Question 6 (p. 88)

5.6.6 Answer 5

D. 3 -3

5.6.6.1 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. 87)

5.6.7 Answer 4

A. Compiler Error

5.6.7.1 Explanation 4

Cannot cast a boolean type

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

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

Table 5.12

Back to Question 4 (p. 86)

5.6.8 Answer 3

D. B

5.6.8.1 Explanation 3

The logical and operator

The logical and operator shown below

The *logical and* operator
&&

Table 5.13

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

The *boolean negation* operator !

Table 5.14

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.

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

Table 5.15

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:

```
!(x<y)
```

Table 5.16

The following expression is true

```
(x < y)
```

Table 5.17

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

```
!(x < y)
```

Table 5.18

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. 85)

5.6.9 Answer 2

C. A

5.6.9.1 Explanation 2

Short-circuit evaluation

Question 1 (p. 83) was intended to set the stage for this question.

This Question, in combination with Question 1 (p. 83) , 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. 83) :

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

Table 5.19

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. 83) .

Similar code from Question 2

Now consider the following code fragment from Question 2 (p. 84) .

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

Table 5.20

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. 84)

5.6.10 Answer 1

B. Runtime Error

5.6.10.1 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:

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

Table 5.21

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:

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

Table 5.22

Back to Question 1 (p. 83)
-end-

Chapter 6

Ap0050: Self-assessment, Escape Character Sequences and Arrays[1]

6.1 Table of Contents

6.2 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. 120) to easily find and view the listings while you are reading about them.

6.3 Questions

6.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 106) ?

```
    A. Compiler Error
B. Runtime Error
C. \"Backslash\"->\\\nUnderstand
```

[1] This content is available online at <http://cnx.org/content/m45280/1.8/>.

```
   D. "Backslash"->\
      Understand
```

Listing 1 . Listing for Question 1.

```
    public class Ap049{
  public static void main(
                        String args[]){
    new Worker().doEscape();
  }//end main()
}//end class definition

class Worker{
  public void doEscape(){
    System.out.println(
      "\"Backslash\"->\\\nUnderstand");
  }//end doEscape()
}// end class
```

Table 6.1

Answer and Explanation (p. 133)

6.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 107) ?

- A. Compiler Error
- B. Runtime Error
- C. St@273d3c St@256a7c St@720eeb
- D. Tom Dick Harry
- E. None of the above

Listing 2 . Listing for Question 2.

```
    public class Ap050{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    St[] myArray = {new St("Tom"),
                    new St("Dick"),
                    new St ("Harry")};
  for(int cnt = 0;
           cnt < myArray.length;cnt++){
    System.out.print(
                   myArray[cnt] + " ");

  }//end for loop
  System.out.println("");
  }//end doArrays()
}// end class

class St{
  private String name;

  public St(String name){
    this.name = name;
  }//end constructor

  public String toString(){
    return name;
  }//end toString()
}//end class
```

Table 6.2

Answer and Explanation (p. 131)

6.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 108) ?

A. Compiler Error

B. Runtime Error

```
C.  0 0 0 0 0
0 1 2 3 4
0 2 4 6 8


D.  None of the above
```

Listing 3 . Listing for Question 3.

```
   public class Ap051{
  public static void main(
                       String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int myArray[3][5];

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        myArray[i][j] = i*j;
      }//end inner for loop
    }//end outer for loop

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        System.out.print(
                   myArray[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

Table 6.3

Answer and Explanation (p. 129)

6.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 109) ?

```
A.  Compiler Error
```

B. Runtime Error

C. 1 1 1 1 1
1 2 3 4 5
1 3 5 7 9

D. None of the above

Listing 4 . Listing for Question 4.

```
    public class Ap052{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int myArray[][];
    myArray = new int[3][5];

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        myArray[i][j] = i*j + 1;
      }//end inner for loop
    }//end outer for loop

    for(int i=0;i<myArray.length;i++){
      for(int j=0;
              j<myArray[0].length;j++){
        System.out.print(
                  myArray[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

Table 6.4

Answer and Explanation (p. 128)

6.3.5 Question 5

What output is produced by program shown in Listing 5 (p. 110) ?

A. Compiler Error

B. Runtime Error

C. -1 -1 -1 -1 -1
-1 -2 -3 -4 -5
-1 -3 -5 -7 -9

D. None of the above

Listing 5 . Listing for Question 5.

```
    public class Ap053{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int myArray[][];
    myArray = new int[3][5];

    for(int i = 0;i < 3;i++){
      for(int j = 0;j < 5;j++){
        myArray[i][j] = (i*j+1)*(-1);
      }//end inner for loop
    }//end outer for loop

    for(int i = 0;i < 3;i++){
      for(int j = 0;j < 6;j++){
        System.out.print(
                  myArray[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

Table 6.5

Answer and Explanation (p. 127)

6.3.6 Question 6

What output is produced by program shown in Listing 6 (p. 111) ?

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

Listing 6 . Listing for Question 6.

```
    public class Ap054{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A = new int[2];
    A[0] = 1;
    A[1] = 2;
    System.out.println(A[0] + A[1]);

  }//end doArrays()
}// end class
```

Table 6.6

Answer and Explanation (p. 127)

6.3.7 Question 7

What output is produced by program shown in Listing 7 (p. 112) ?

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

Listing 7 . Listing for Question 7.

continued on next page

```
    import java.awt.Label;
public class Ap055{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Label[] A = new Label[2];
    A[0] = new Label("O");
    A[1] = new Label("K");
    System.out.println(A[0] + A[1]);
  }//end doArrays()
}// end class
```

Table 6.7

Answer and Explanation (p. 125)

6.3.8 Question 8

What output is produced by program shown in Listing 8 (p. 113) ?

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

Listing 8 . Listing for Question 8.

```
import java.awt.Label;
public class Ap056{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Label[] A = new Label[2];
    A[0] = new Label("O");
    A[1] = new Label("K");
    System.out.println(A[0].getText() + A[1].getText());
  }//end doArrays()
}// end class
```

Table 6.8

Answer and Explanation (p. 124)

6.3.9 Question 9

What output is produced by program shown in Listing 9 (p. 114) ?

- A. Compiler Error
- B. Runtime Error
- C. 1
- D. None of the above

Listing 9 . Listing for Question 9.

```
    public class Ap057{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Integer[] A = new Integer[2];
    A[0] = new Integer(1);
    System.out.println(
                      A[1].intValue());
  }//end doArrays()
}// end class
```

Table 6.9

Answer and Explanation (p. 124)

6.3.10 Question 10

What output is produced by program shown in Listing 10 (p. 115) ?

```
A.  Compiler Error

B.  Runtime Error

C.  0
0 1
0 2 4

D.  None of the above
```

Listing 10 . Listing for Question 10.

```
    public class Ap058{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int A[][] = new int[3][];
    A[0] = new int[1];
    A[1] = new int[2];
    A[2] = new int[3];

    for(int i = 0;i < A.length;i++){
      for(int j=0;j < A[i].length;j++){
        A[i][j] = i*j;
      }//end inner for loop
    }//end outer for loop

    for(int i=0;i<A.length;i++){
      for(int j=0;j < A[i].length;j++){
        System.out.print(
                        A[i][j] + " ");
      }//end inner for loop
      System.out.println("");
    }//end outer for loop

  }//end doArrays()
}// end class
```

Table 6.10

Answer and Explanation (p. 123)

6.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 116) ?

- A. Compiler Error
- B. Runtime Error
- C. Zero One Two
- D. None of the above

Listing 11 . Listing for Question 11.

```
    public class Ap059{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[3];
    //Note that there is a simpler and
    // better way to do the following
    // but it won't illustrate my point
    // as well as doing it this way.
    A[0] = new String("Zero");
    A[1] = new String("One");
    A[2] = new String("Two");

    System.out.println(A[0] + " " +
                       A[1] + " " +
                       A[2]);
  }//end doArrays()
}// end class
```

Table 6.11

Answer and Explanation (p. 123)

6.3.12 Question 12

What output is produced by program shown in Listing 12 (p. 117) ?

- A. Compiler Error
- B. Runtime Error
- C. Zero 1 2.0
- D. None of the above.

Listing 12 . Listing for Question 12.

```
    public class Ap060{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[3];
    //Note that there is a simpler and
    // better way to do the following
    // but it won't illustrate my point
    // as well as doing it this way.
    A[0] = new String("Zero");
    A[1] = new Integer(1);
    A[2] = new Double(2.0);

    System.out.println(A[0] + "  " +
                       A[1] + "  " +
                       A[2]);
  }//end doArrays()
}// end class
```

Table 6.12

Answer and Explanation (p. 122)

6.3.13 Question 13

What output is produced by program shown in Listing 13 (p. 118) ?

- A. Compiler Error
- B. Runtime Error
- C. Zero 1 2.0
- D. None of the above.

Listing 13 . Listing for Question 13.

```
    public class Ap061{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[3];
    //Note that there is a simpler and
    // better way to do the following
    // but it won't illustrate my point
    // as well as doing it this way.
    A[0] = new String("Zero");
    A[1] = new Integer(1);
    A[2] = new MyClass(2.0);

    System.out.println(A[0] + "  " +
                       A[1] + "  " +
                       A[2]);
  }//end doArrays()
}// end class

class MyClass{
  private double data;

  public MyClass(double data){
    this.data = data;
  }//end constructor
}// end MyClass
```

Table 6.13

Answer and Explanation (p. 122)

6.3.14 Question 14

What output is produced by program shown in Listing 14 (p. 119) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 14 . Listing for Question 14.

```
    public class Ap062{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[2];
    A[0] = new MyClass(1.0);
    A[1] = new MyClass(2.0);

    System.out.println(
      A[0].getData() + "  " +
      A[1].getData());
  }//end doArrays()
}// end class

class MyClass{
  private double data;

  public MyClass(double data){
    this.data = data;
  }//end constructor

  public double getData(){
    return data;
  }//end getData()
}// end MyClass
```

Table 6.14

Answer and Explanation (p. 122)

6.3.15 Question 15

What output is produced by program shown in Listing 15 (p. 120) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 15 . Listing for Question 15.

```
    public class Ap063{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Object[] A = new Object[2];
    A[0] = new MyClass(1.0);
    A[1] = new MyClass(2.0);

    System.out.println(
      ((MyClass)A[0]).getData() + "  "
      + ((MyClass)A[1]).getData());
  }//end doArrays()
}// end class

class MyClass{
  private double data;

  public MyClass(double data){
    this.data = data;
  }//end constructor

  public double getData(){
    return data;
  }//end getData()
}// end MyClass
```

Table 6.15

Answer and Explanation (p. 121)

6.4 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 8 (p. 113) . Listing for Question 8.
- Listing 9 (p. 114) . Listing for Question 9.
- Listing 10 (p. 115) . Listing for Question 10.
- Listing 11 (p. 116) . Listing for Question 11.
- Listing 12 (p. 117) . Listing for Question 12.
- Listing 13 (p. 118) . Listing for Question 13.
- Listing 14 (p. 119) . Listing for Question 14.
- Listing 15 (p. 120) . Listing for Question 15.

6.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0050: Self-assessment, Escape Character Sequences and Arrays
- File: Ap0050.htm
- Originally published: 2002
- Published at cnx.org: 12/03/12
- Revised: 08/17/15

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.

6.6 Answers

6.6.1 Answer 15

C. 1.0 2.0

6.6.1.1 Explanation 15

This is an upgrade of the program from Question 14 (p. 118) . This program applies the proper downcast operator to the references extracted from the array of type **Object** before attempting to call the method named **getData** on those references. *(For more information, see the discussion of Question 14* (p. 118) *.)*

As a result of applying a proper downcast, the program compiles and runs successfully.

Back to Question 15 (p. 119)

6.6.2 Answer 14

A. Compiler Error

6.6.2.1 Explanation 14

Storing references in a generic array of type Object

This program stores references to two objects instantiated from a new class named **MyClass** in the elements of an array object of declared type **Object** . That is OK.

Calling a method on the references

Then the program extracts the references to the two objects and attempts to call the method named **getData** on each of the references. That is not OK.

Downcast is required

Because the method named **getData** is not defined in the class named **Object** , in order to call this method on references extracted from an array of type **Object** , it is necessary to downcast the references to the class in which the method is defined. In this case, the method is defined in the new class named **MyClass** *(but it could be defined in an intermediate class in the class hierarchy if the new class extended some class further down the hierarchy)* .

Here is a partial listing of the compiler error produced by this program:

```
  Ap062.java:15: error: cannot find symbol
    A[0].getData() + "  " +
         ^
symbol:   method getData()
location: class Object
```

Back to Question 14 (p. 118)

6.6.3 Answer 13

D. None of the above.

6.6.3.1 Explanation 13

The array object of type **Object** in this program is capable of storing a reference to a new object instantiated from the new class named **MyClass** . However, because the new class does not override the **toString** method, when a string representation of the new object is displayed, the string representation is created using the version of the **toString** method that is inherited from the **Object** class. That causes this program to produce an output similar to the following:

Zero 1 MyClass@273d3c

Back to Question 13 (p. 117)

6.6.4 Answer 12

C. Zero 1 2.0

6.6.4.1 Explanation 12

A type-generic array object

As explained in Question 11 (p. 115) , an array object of the type **Object** is a generic array that can be used to store references to objects instantiated from any class.

Storing mixed reference types

This program instantiates objects from the classes **String** , **Integer** , and **Double** , and stores those objects' references in the elements of an array of type **Object** . Then the program accesses the references and uses them to display string representations of each of the objects.

Polymorphic behavior applies

Once again, polymorphic behavior involving overridden versions of the **toString** method were involved and it was not necessary to downcast the references to their true type to display string representations of the objects.

Back to Question 12 (p. 116)

6.6.5 Answer 11

C. Zero One Two

6.6.5.1 Explanation 11

Storing references to subclass types

When you create an array object for a type defined by a class definition, the elements of the array can be used to store references to objects of that class or any subclass of that class.

A type-generic array object

All classes in Java are subclasses of the class named **Object** . This program creates an array object with the declared type being type **Object** . An array of type **Object** can be used to store references to objects instantiated from any class.

After creating the array object, this program instantiates three objects of the class **String** and stores those object's references in the elements of the array. *(As I pointed out in the comments, there is a simpler and better way to instantiate* ***String*** *objects, but it wouldn't illustrate my point as well as doing the way that I did.)*

Sometimes you need to downcast

Although an array of type **Object** can be used to store references to objects of any type *(including mixed types and references to other array objects)* , you will sometimes need to downcast those references back to their true type once you extract them from the array and attempt to use them for some purpose.

Polymorphic behavior applies here

For this case, however, because the **toString** method is defined in the **Object** class and overridden in the **String** class, polymorphic behavior applies and it is not necessary to downcast the references to type **String** in order to be able to convert them to strings and display them.

Back to Question 11 (p. 115)

6.6.6 Answer 10

```
C.  0
0 1
0 2 4
```

6.6.6.1 Explanation 10

Defer size specification for secondary arrays

It is not necessary to specify the sizes of the secondary arrays when you create a multi-dimensional array in Java. Rather, since the elements in the primary array simply contain references to other array objects *(or null by default)* , you can defer the creation of those secondary array objects until later.

Independent array objects

When you do finally create the secondary arrays, they are essentially independent array objects *(except for the requirement for type commonality among them)* .

Ragged arrays

Each individual secondary array can be of any size, and this leads to the concept of a *ragged array* . *(On a two-dimensional basis, a ragged array might be thought of as a two-dimensional array where each row can have a different number of columns.)*

This program creates, populates, and displays the contents of such a two-dimensional ragged array. Although this program creates a two-dimensional array that is triangular in shape, even that is not a requirement. The number of elements in each of the secondary arrays need have no relationship to the number of elements in any of the other secondary arrays.

Back to Question 10 (p. 114)

6.6.7 Answer 9

B. Runtime Error

6.6.7.1 Explanation 9

NullPointerException

The following code fragment shows that this program attempts to perform an illegal operation on the value accessed from the array object at index 1.

```
    Integer[] A = new Integer[2];
A[0] = new Integer(1);
System.out.println(
                  A[1].intValue());
```

You can't call methods on null references

The reference value that was returned by accessing **A[1]** is the default value of null. This is the value that was deposited in the element when the array object was created *(no other value was ever stored there)* . When an attempt was made to call the **intValue** method on that reference value, the following runtime error occurred

```
   java.lang.NullPointerException
at Worker.doArrays(Ap057.java:14)
at Ap057.main(Ap057.java:6)
```

This is a common programming error, and most Java programmers have seen an error message involving a **NullPointerException** several **(perhaps many)** times during their programming careers.

Back to Question 9 (p. 113)

6.6.8 Answer 8

C. OK

6.6.8.1 Explanation 8

Success at last

This program finally gets it all together and works properly. In particular, after accessing the reference values stored in each of the elements, the program does something legal with those values.

Call methods on the object's references

In this case, the code calls one of the public methods belonging to the objects referred to by the reference values stored in the array elements.

```
System.out.println(A[0].getText() + A[1].getText());
```

The **getText** method that is called, returns the contents of the **Label** object as type **String** . This makes it possible to perform **String** concatenation on the values returned by the method, so the program compiles and executes properly.

Back to Question 8 (p. 112)

6.6.9 Answer 7

A. Compiler Error

6.6.9.1 Explanation 7

Java arrays may seem different to you

For all types other than the primitive types, you may find the use of arrays in Java to be considerably different from what you are accustomed to in other programming languages. There are a few things that you should remember.

Array elements may contain default values

If the declared type of an array is one of the primitive types, the elements of the array contain values of the declared type. If you have not initialized those elements or have not assigned specific values to the elements, they will contain default values.

The default values

You need to know that:

- The default for numeric primitive types is the zero value for that type
- The default for the **boolean** type is false
- The default for the **char** type is a 16-bit unsigned integer, all of whose bits have a zero value *(sometimes called a null character)*
- The default value for reference types is *null* , not to be confused with the *null character* above. *(An array element that contains null doesn't refer to an object.)*

Arrays of references

If the declared type for the array is not one of the primitive types, the elements in the array are actually reference variables. Objects are never stored directly in a Java array. Only references to objects are stored in a Java array.

If the array type is the name of a class ...

If the declared type is the name of a class, references to objects of that class or any subclass of that class can be stored in the elements of the array.

If the array type is the name of an interface ...

If the declared type is the name of an interface, references to objects of any class that implements the interface, or references to objects of any subclass of a class that implements the interface can be stored in the elements of the array.

Why did this program fail to compile?

Now back to the program at hand. Why did this program fail to compile? To begin with, this array was not designed to store any of the primitive types. Rather, this array was designed to store references to objects instantiated from the class named **Label**, as indicated in the following fragment.

```
Label[] A = new Label[2];
```

Elements initialized to null

This is a two-element array. When first created, it contains two elements, each having a default value of *null*. What this really means is that the reference values stored in each of the two elements don't initially refer to any object.

Populate the array elements

The next fragment creates two instances *(objects)* of the **Label** class and assigns those object's references to the two elements in the array object. This is perfectly valid.

```
A[0] = new Label("0");
A[1] = new Label("K");
```

You cannot add reference values

The problem arises in the next fragment. Rather than dealing with the object's references in an appropriate manner, this fragment attempts to access the text values of the two reference variables and concatenate those values.

```
System.out.println(A[0] + A[1]);
```

The compiler produces the following error message:

```
Ap055.java:14: error: bad operand types for binary operator '+'
    System.out.println(A[0] + A[1]);
                           ^
  first type:  Label
  second type: Label
1 error
```

This error message is simply telling us that it is not legal to add the values of reference variables.

Not peculiar to arrays

This problem is not peculiar to arrays. You would get a similar error if you attempted to add two reference variables even when they aren't stored in an array. In this case, the code to access the values of the elements is good. The problem arises when we attempt to do something illegal with those values after we access them.

Usually two steps are required

Therefore, except in some special cases such as certain operations involving the wrapper classes, to use Java arrays with types other than the primitive types, when you access the value stored in an element of the array *(a reference variable)* you must perform only those operations on that reference variable that are legal for an object of that type. That usually involves two steps. The first step accesses the reference to an object. The second step performs some operation on the object.

Back to Question 7 (p. 111)

6.6.10 Answer 6

C. 3

6.6.10.1 Explanation 6

Once you create an array object for a primitive type in Java, you can treat the elements of the array pretty much as you would treat the elements of an array in other programming languages. In particular, a statement such the following can be used to assign a value to an indexed element in an array referred to by a reference variable named **A** .

```
A[1] = 2;
```

Similarly, when you reference an indexed element in an expression such as the following, the value stored in the element is used to evaluate the expression.

```
System.out.println(A[0] + A[1]);
```

For all Java arrays, you must remember to create the new array object and to store the array object's reference in a reference variable of the correct type. Then you can use the reference variable to gain access to the elements in the array.

Back to Question 6 (p. 111)

6.6.11 Answer 5

B. Runtime Error

6.6.11.1 Explanation 5

Good fences make good neighbors

One of the great things about an array object in Java is that it knows how to protect its boundaries.

Unlike some other currently popular programming languages, if your program code attempts to access a Java array element outside its boundaries, an exception will be thrown. If your program doesn't catch and handle the exception, the program will be terminated.

Abnormal termination

While experiencing abnormal program termination isn't all that great, it is better than the alternative of using arrays whose boundaries aren't protected. Programming languages that don't protect the array boundaries simply overwrite other data in memory whenever the array boundaries are exceeded.

Attempt to access out of bounds element

The code in the **for** loop in the following fragment attempts to access the array element at the index value 5. That index value is out of bounds of the array.

```
    for(int j = 0;j < 6;j++){
  System.out.print(
            myArray[i][j] + " ");
}//end inner for loop
```

Because that index value is outside the boundaries of the array, an **ArrayIndexOutOfBoundsException** is thrown. The exception isn't caught and handled by program code, so the program terminates abnormally at runtime.

This program also illustrates that it is usually better to use the **length** property of an array to control iterative loops than to use hard-coded limit values, which may be coded erroneously.

Back to Question 5 (p. 109)

6.6.12 Answer 4

```
C.  1 1 1 1 1
1 2 3 4 5
1 3 5 7 9
```

6.6.12.1 Explanation 4

A two-dimensional array

This program illustrates how to create, populate, and process a two-dimensional array with three rows and five columns.

(As mentioned earlier, a Java programmer who understands the fine points of the language probably wouldn't call this a two-dimensional array. Rather, this is a one-dimensional array containing three elements. Each of those elements is a reference to a one-dimensional array containing five elements. That is the more general way to think of Java arrays.)

The following code fragment creates the array, using one of the acceptable formats discussed in Question 3 (p. 107) .

```
    int myArray[][];
myArray = new int[3][5];
```

Populating the array

The next code fragment uses a pair of nested **for** loops to populate the elements in the array with values of type **int** .

```
  for(int i=0;i<myArray.length;i++){
for(int j=0;
        j<myArray[0].length;j++){
  myArray[i][j] = i*j + 1;
}//end inner for loop
}//end outer for loop
```

This is where the analogy of a two-dimensional array falls apart. It is much easier at this point to think in terms of a three-element primary array, each of whose elements contains a reference to a secondary array containing five elements. *(Note that in Java, the secondary arrays don't all have to be of the same size. Hence, it is possible to create odd-shaped multi-dimensional arrays in Java.)*

Using the length property

Pay special attention to the two chunks of code that use the length properties of the arrays to determine the number of iterations for each of the **for** loops.

The first chunk determines the number of elements in the primary array. In this case, the length property contains the value 3.

The second chunk determines the number of elements in the secondary array that is referred to by the contents of the element at index 0 in the primary array. *(Think carefully about what I just said.)*

In this case, the length property of the secondary array contains the value 5.

Putting data into the secondary array elements

The code interior to the inner loop simply calculates some numeric values and stores those values in the elements of the three secondary array objects.

Let's look at a picture

Here is a picture that attempts to illustrate what is really going on here. I don't know if it will make sense to you or not, but hopefully, it won't make the situation any more confusing than it might already be.

```
[->]  [1][1][1][1][1]

[->]  [1][2][3][4][5]

[->]  [1][3][5][7][9]
```

The primary array

The three large boxes on the left represent the individual elements of the three-element primary array. The length property for this array has a value of 3. The arrows in the boxes indicate that the content of each of these three elements is a reference to one of the five-element arrays on the right.

The secondary arrays

Each of the three rows of five boxes on the right represents a separate five-element array object. Each element in each of those array objects contains the **int** value shown. The length property for each of those arrays has a value of 5.

Access and display the array data

The code in the following fragment is another pair of nested **for** loops.

```
  for(int i=0;i<myArray.length;i++){
for(int j=0;
       j<myArray[0].length;j++){
  System.out.print(
            myArray[i][j] + " ");
}//end inner for loop
System.out.println("");
}//end outer for loop
```

In this case, the code in the inner loop accesses the contents of the individual elements in the three five-element arrays and displays those contents. If you understand the earlier code in this program, you shouldn't have any difficulty understanding the code in this fragment.

Back to Question 4 (p. 108)

6.6.13 Answer 3

A. Compiler Error

6.6.13.1 Explanation 3

An incorrect statement

The following statement is not the proper way to create an array object in Java.

```
int myArray[3][5];
```

This statement caused the program to fail to compile, producing several error messages.

What is the correct syntax?

There are several different formats that can be used to create an array object in Java. One of the acceptable ways was illustrated by the code used in Question 2 (p. 106) . Three more acceptable formats are shown below.

```
int[][] myArrayA = new int[3][5];

int myArrayB[][] = new int[3][5];

int myArrayC[][];
myArrayC = new int[3][5];
```

Two steps are required

The key thing to remember is that an array is an object in Java. Just like all other *(non-anonymous)* objects in Java, there are two steps involved in creating and preparing an object for use.

Declare a reference variable

The first step is to declare a reference variable capable of holding a reference to the object.

The second step

The second step is to create the object and to assign the object's reference to the reference variable. From that point on, the reference variable can be used to gain access to the object.

Two steps can often be combined

Although there are two steps involved, they can often be combined into a single statement, as indicated by the first two acceptable formats shown above.

In both of these formats, the code on the left of the assignment operator declares a reference variable. The code on the right of the assignment operator creates a new array object and returns the array object's reference. The reference is assigned to the new reference variable declared on the left.

A two-dimensional array object

In the code fragments shown above, the array object is a two-dimensional array object that can be thought of as consisting of three rows and five columns.

(Actually, multi-dimensional array objects in Java can be much more complex than this. In fact, although I have referred to this as a two-dimensional array object, there is no such thing as a multi-dimensional array object in Java. The concept of a multi-dimensional array in Java is achieved by creating a tree structure of single-dimensional array objects that contain references to other single-dimensional array objects.)

The square brackets in the declaration

What about the placement and the number of matching pairs of empty square brackets? As indicated in the first two acceptable formats shown above, the empty square brackets can be next to the name of the type or next to the name of the reference variable. The end result is the same, so you can use whichever format you prefer.

How many pairs of square brackets are required?

Also, as implied by the acceptable formats shown above, the number of matching pairs of empty square brackets must match the number of so-called *dimensions* of the array. *(This tells the compiler to create*

a reference variable capable of holding a reference to a one-dimensional array object, whose elements are capable of holding references to other array objects.)

Making the two steps obvious

A third acceptable format, also shown above, separates the process into two steps.

One statement in the third format declares a reference variable capable of holding a reference to a two-dimensional array object containing data of type **int** . When that statement finishes executing, the reference variable exists, but it doesn't refer to an actual array object. The next statement creates an array object and assigns that object's reference to the reference variable.

Back to Question 3 (p. 107)

6.6.14 Answer 2

D. Tom Dick Harry

6.6.14.1 Explanation 2

An array is an object in Java

An array is a special kind of object in Java. Stated differently, all array structures are encapsulated in objects in Java. Further. all array structures are one-dimensional. I often refer to this special kind of object as an *array object* .

An array object always has a property named **length** . The value of the **length** property is always equal to the number of elements in the array. Thus, a program can always determine the size of an array be examining its **length** property.

Instantiating an array object

An array object can be instantiated in at least two different ways:

1. By using the **new** operator in conjunction with the type of data to be stored in the array.
2. By specifying an initial value for every element in the array, in which case the **new** operator is not used.

This program uses the second of the two ways listed above.

Declaring a reference variable for an array object

The following code fragment was extracted from the method named doArrays().

```
St[] myArray = {new St("Tom"),
                new St("Dick"),
                new St ("Harry")};
```

The code to the left of the assignment operator declares a reference variable named **myArray** . This reference variable is capable of holding a reference to an array object that contains an unspecified number of references to objects instantiated from the class named **St** *(or any subclass of the class named St)* .

Note the square brackets

You should note the square brackets in the declaration of the reference variable in the above code *(the declaration of a reference variable to hold a reference to an ordinary object doesn't include square brackets)* .

Create the array object

The code to the right of the assignment operator in the above fragment causes the new array object to come into being. Note that the **new** operator is not used to create the array object in this case. *(This is one of the few cases in Java, along with a literal* ***String*** *object, where it is possible to create a new object without using either the* ***new*** *operator or the* ***newInstance*** *method of the class whose name is* ***Class*** *.)*

Populate the array object

This syntax not only creates the new array object, it also populates it. The new array object created by the above code contains three elements, because three initial values were provided. The initial values are separated by commas in the initialization syntax.

Also instantiates three objects of the St class

The code in the above fragment also instantiates three objects of the class named **St** . Once the array object has come into being, each of the three elements in the array contains a reference to a new object of the class **St** . Each of those objects is initialized to contain the name of a student by using a parameterized constructor that is defined in the class.

The length property value is 3

Following execution of the above code, the **length** property of the array object will contain a value of 3, because the array contains three elements, one for each initial value that was provided.

Using the length property

The code in the following fragment uses the **length** property of the array object in the conditional clause of a **for** loop to display a **String** representation of each of the objects.

```
  for(int cnt = 0;
  cnt < myArray.length;
  cnt++){
System.out.print(
              myArray[cnt] + " ");
```

Overridden toString method

The class named **St** , from which each of the objects was instantiated, defines an overridden **toString** method that causes the string representation of an object of that class to consist of the **String** stored in an instance variable of the object.

Thus, the **for** loop shown above displays the student names that were originally encapsulated in the objects when they were instantiated.

The class named St

The code in the following fragment shows the beginning of the class named **St** including one instance variable and a parameterized constructor.

```
  class St{
private String name;

public St(String name){
  this.name = name;
}//end constructor
```

A very common syntax

This constructor makes use of a very common syntax involving the reference named **this** . Basically, this syntax says to get the value of the incoming parameter whose name is **name** and to assign that value to the instance variable belonging to *this object* whose name is also **name** .

Initializing the object of type St

Each time a new object of the **St** class is instantiated, that object contains an instance variable of type **String** whose value matches the **String** value passed as a parameter to the constructor.

Overridden toString method

The overridden **toString** method for the class named **St** is shown in the following code fragment.

```
    public String toString(){
  return name;
}//end toString()
```

This version causes the value in the **String** object, referred to by the instance variable named **name**, to be returned when it is necessary to produce a **String** representation of the object.

Back to Question 2 (p. 106)

6.6.15 Answer 1

The answer is item D, which reads as follows:

```
   "Backslash"->\
Understand
```

6.6.15.1 Explanation 1

Don't confuse the compiler

If you include certain characters inside a literal **String**, you will confuse the compiler. For example, if you simply include a quotation mark (") inside a literal **String**, the compiler will interpret that as the end of the string. From that point on, everything will be out of synchronization. Therefore, in order to include a quotation mark inside a literal string, you must precede it with a backslash character like this:

\"

Multiple lines

If you want your string to comprise two or more physical lines, you can include a newline code inside a **String** by including the following in the string:

\n

Escape character sequences

These character sequences are often referred to as *escape character sequences*. Since the backslash is used as the first character in such a sequence, if you want to include a backslash in a literal string, you must do it like this:

\\

There are some other escape sequences used in Java as well. You would do well to learn how to use them before going to an interview for a job as a Java programmer.

Back to Question 1 (p. 105)

-end-

Chapter 7

Ap0060: Self-assessment, More on Arrays[1]

7.1 Table of Contents

7.2 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. 151) to easily find and view the listings while you are reading about them.

7.3 Questions

7.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 136) ?

- A. Compiler Error
- B. Runtime Error
- C. I'm OK
- D. None of the above

[1] This content is available online at <http://cnx.org/content/m45264/1.5/>.

Listing 1 . Listing for Question 1.

```
    public class Ap064{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    System.out.println("I'm OK");
  }//end doArrays()
}// end class
```

Table 7.1

Answer and Explanation (p. 160)

7.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 137) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 2 . Listing for Question 2.

```
    public class Ap065{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class
```

Table 7.2

Answer and Explanation (p. 159)

7.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 138) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 3 . Listing for Question 3.

```
    public class Ap066{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    double C = (double)B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class
```

Table 7.3

Answer and Explanation (p. 159)

7.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 139) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 4 . Listing for Question 4.

```
    public class Ap067{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    double[] C = (double[])B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class
```

Table 7.4

Answer and Explanation (p. 159)

7.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 140) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. None of the above.

Listing 5 . Listing for Question 5.

```
    public class Ap068{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    String[] C = (String[])B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class
```

Table 7.5

Answer and Explanation (p. 158)

7.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 141) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 6 . Listing for Question 6.

```
    public class Ap069{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    System.out.println(
                    A[0] + " " + A[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

Table 7.6

Answer and Explanation (p. 158)

7.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 142) ?

- A. Compiler Error

- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 7 . Listing for Question 7.

```
    public class Ap070{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    Superclass[] B = A;
    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

Table 7.7

Answer and Explanation (p. 157)

7.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 143) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 8 . Listing for Question 8.

```
    public class Ap071{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Superclass[] A = new Superclass[2];
    A[0] = new Superclass(1);
    A[1] = new Superclass(2);

    Subclass[] B = (Subclass[])A;
    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

Table 7.8

Answer and Explanation (p. 157)

7.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 145) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 9 . Listing for Question 9.

```
    public class Ap072{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    Superclass[] B = A;
    Subclass[] C = (Subclass[])B;
    System.out.println(
                    C[0] + " " + C[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

Table 7.9

Answer and Explanation (p. 156)

7.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 146) ?

- A. Compiler Error
- B. Runtime Error
- C. 1.0 2.0
- D. D. None of the above

Listing 10 . Listing for Question 10.

```
    public class Ap073{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    double[] A = new double[2];
    A[0] = 1.0;
    A[1] = 2.0;
    Object B = A;

    System.out.println(
              ((double[])B)[0] + " " +
              ((double[])B)[1]);
  }//end doArrays()
}// end class
```

Table 7.10

Answer and Explanation (p. 156)

7.3.11 Question 11

What output is produced by the program shown in Listing 11 (p. 147) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 11 . Listing for Question 11.

continued on next page

```
    public class Ap074{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A = new int[2];
    A[0] = 1;
    A[1] = 2;

    double[] B = (double[])A;

    System.out.println(
                    B[0] + " " + B[1]);
  }//end doArrays()
}// end class
```

Table 7.11

Answer and Explanation (p. 155)

7.3.12 Question 12

What output is produced by the program shown in Listing 12 (p. 148) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 12 . Listing for Question 12.

```
    public class Ap075{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] B = returnArray();
    for(int i = 0; i < B.length;i++){
      System.out.print(B[i] + " ");
    }//end for loop
    System.out.println();
  }//end doArrays()

  public int[] returnArray(){
    int[] A = new int[2];
    A[0] = 1;
    A[1] = 2;
    return A;
  }//end returnArray()
}// end class
```

Table 7.12

Answer and Explanation (p. 155)

7.3.13 Question 13

What output is produced by the program shown in Listing 13 (p. 149) ?

A. Compiler Error

B. Runtime Error

C. 0 0 0
0 1 2

D. None of the above

Listing 13 . Listing for Question 13.

```
    public class Ap076{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A[];
    A = new int[2][3];

    for(int i=0; i<A.length;i++){
      for(int j=0;j<A[0].length;j++){
        A[i][j] = i*j;
      }//end inner loop
    }//end outer loop

    for(int i=0; i<A.length;i++){
      for(int j=0;j<A[0].length;j++){
        System.out.print(
                        A[i][j] + " ");
      }//end inner loop
      System.out.println();
    }//end outer loop

  }//end doArrays()
}// end class
```

Table 7.13

Answer and Explanation (p. 154)

7.3.14 Question 14

What output is produced by the program shown in Listing 14 (p. 150) ?

- A. Compiler Error
- B. Runtime Error
- C. 1 2
- D. None of the above

Listing 14 . Listing for Question 14.

```
    public class Ap077{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    Subclass[] A = new Subclass[2];
    A[0] = new Subclass(1);
    A[1] = new Subclass(2);

    Object X = A;
    Superclass B = A;
    Subclass[] C = (Subclass[])B;
    Subclass[] Y = (Subclass[])X;
    System.out.println(
                     C[0] + " " + Y[1]);
  }//end doArrays()
}// end class

class Superclass{
  private int data;
  public Superclass(int data){
    this.data = data;
  }//end constructor

  public int getData(){
    return data;
  }//end getData()

  public String toString(){
    return "" + data;
  }//end toString()
}//end class SuperClass

class Subclass extends Superclass{
  public Subclass(int data){
    super(data);
  }//end constructor
}//end class Subclass
```

Table 7.14

Answer and Explanation (p. 153)

7.3.15 Question 15

What output is produced by the program shown in Listing 15 (p. 151) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 0.0 false 0
- D. None of the above

Listing 15 . Listing for Question 15.

```
    public class Ap078{
  public static void main(
                        String args[]){
    new Worker().doArrays();
  }//end main()
}//end class definition

class Worker{
  public void doArrays(){
    int[] A = new int[1];
    double[] B = new double[1];
    boolean[] C = new boolean[1];
    int[] D = new int[0];

    System.out.println(A[0] + " " +
                       B[0] + " " +
                       C[0] + " " +
                       D.length);

  }//end doArrays()
}// end class
```

Table 7.15

Answer and Explanation (p. 152)

7.4 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. 136) . Listing for Question 1.
- Listing 2 (p. 137) . Listing for Question 2.
- Listing 3 (p. 138) . Listing for Question 3.
- Listing 4 (p. 139) . Listing for Question 4.
- Listing 5 (p. 140) . Listing for Question 5.
- Listing 6 (p. 141) . Listing for Question 6.
- Listing 7 (p. 142) . Listing for Question 7.

7.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0060: Self-assessment, More on Arrays
- File: Ap0060.htm
- Originally published: 2002
- Published at cnx.org: 12/03/12
- Revised: 12/03/14

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.

7.6 Answers

7.6.1 Answer 15

C. 0 0.0 false 0

7.6.1.1 Explanation 15

You can initialize array elements

You can create a new array object and initialize its elements using statements similar to the following:

```
    int[] A = {22, 43, 69};
X[] B = {new X(32), new X(21)};
```

What if you don't initialize array elements?

If you create a new array object without initializing its elements, the value of each element in the array is automatically initialized to a default value.

Illustrating array element default initialization

This program illustrates default initialization of **int** , **double** , and **boolean** arrays.

The default values are as follows:

- zero for all numeric values
- false for all **boolean** values
- all zero bits for char values
- null for object references

An array with no elements ...

This program also illustrates that it is possible to have an array object in Java that has no elements. In this case, the value of the **length** property for the array object is 0.

Give me an example

For example, when the user doesn't enter any arguments on the command line for a Java application, the incoming **String** array parameter to the **main** method has a length value of 0.

Another example

It is also possible that methods that return a reference to an array object may sometimes return a reference to an array whose length is 0. The method must satisfy the return type requirement by returning a reference to an array object. Sometimes, there is no data to be used to populate the array, so the method will simply return a reference to an array object with a **length** property value of 0.

Back to Question 15 (p. 151)

7.6.2 Answer 14

A. Compiler Error

7.6.2.1 Explanation 14

Assigning array reference to type Object

As you learned in an earlier module, you can assign an array object's reference to an ordinary reference variable of the type **Object** . It is not necessary to indicate that the reference variable is a reference to an array by appending square brackets to the type name or the variable name.

Only works with type Object

However, you cannot assign an array object's reference to an ordinary reference variable of any other type. For any type other than **Object** , the reference variable must be declared to hold a reference to an array object by appending empty square brackets onto the type name or the variable name.

The first statement in the following fragment compiles successfully.

```
        Object X = A;
    Superclass B = A;
```

However, the second statement in the above fragment produces a compiler error under JDK 1.3, which is partially reproduced below.

```
    Ap077.java:22: incompatible types
found   : Subclass[]
required: Superclass
    Superclass B = A;
```

Both **Superclass** and **Object** are superclasses of the array type referred to by the reference variable named **A** . However, because of the above rule, in order to cause this program to compile successfully, you would need to modify it as shown below by adding the requisite empty square brackets to the **Superclass** type name.

```
    Object X = A;
Superclass[] B = A;
```

Back to Question 14 (p. 149)

7.6.3 Answer 13

```
C.  0 0 0
0 1 2
```

7.6.3.1 Explanation 13

Syntactical ugliness

As I indicated in an earlier module, when declaring a reference variable that will refer to an array object, you can place the empty square brackets next to the name of the type or next to the name of the reference variable. In other words, either of the following formats will work.

```
    int[][] A;
int B[][];
```

What I may not have told you at that time is that you can place some of the empty square brackets in one location and the remainder in the other location.

Really ugly syntax

This is indicated by the following fragment, which declares a reference variable for a two-dimensional array of type **int** . Then it creates the two-dimensional array object and assigns the array object's reference to the reference variable.

```
    int[] A[];
A = new int[2][3];
```

While it doesn't matter which location you use for the square brackets in the declaration, it does matter how many pairs of square brackets you place in the two locations combined. The number of dimensions on the array *(if you want to think of a Java array as having dimensions)* will equal the total number of pairs of empty square brackets in the declaration of the reference variable. Thus, in this case, the array is a two-dimensional array because there is one pair of square brackets next to the type and another pair next to the variable name.

This program goes on to use nested for loops to populate the array and then to display the contents of the elements.

I personally don't use this syntax, and I hope that you don't either. However, even if you don't use it, you need to be able to recognize it when used by others.

Back to Question 13 (p. 148)

7.6.4 Answer 12

C. 1 2

7.6.4.1 Explanation 12

The length property

This program illustrates the use of the array property named **length** , whose value always matches the number of elements in the array.

As a Java programmer, you will frequently call methods that will return a reference to an array object of a specified type, but of an unknown length. *(See, for example, the method named* ***getEventSetDescriptors*** *that is declared in the interface named* ***BeanInfo*** *.)* This program simulates that situation.

Returning a reference to an array

The method named **returnArray** returns a reference to an array of type **int** having two elements. Although I fixed the size of the array in this example, I could just as easily have used a random number to set a different size for the array each time the method is called. Therefore, the **doArrays** method making the call to the method named **returnArray** has no way of knowing the size of the array referred to by the reference that it receives as a return value.

All array objects have a length property

This could be a problem, but Java provides the solution to the problem in the **length** property belonging to all array objects.

The **for** loop in the method named **doArrays** uses the **length** property of the array to determine how many elements it needs to display. This is a very common scenario in Java.

Back to Question 12 (p. 147)

7.6.5 Answer 11

A. Compiler Error

7.6.5.1 Explanation 11

You cannot cast primitive array references

You cannot cast an array reference from one primitive type to another primitive type, even if the individual elements in the array are of a type that can normally be converted to the new type.

This program attempts to cast a reference to an array of type **int[]** and assign it to a reference variable of type **double** []. Normally, a value of type **int** will be automatically converted to type **double** whenever there is a need for such a conversion. However, this attempted cast produces the following compiler error under JDK 1.3.

```
    Ap074.java:19: inconvertible types
found   : int[]
required: double[]
    double[] B = (double[])A;
```

Why is this cast not allowed?

I can't give you a firm reason why such a cast is not allowed, but I believe that I have a good idea why. I speculate that this is due to the fact that the actual primitive values are physically stored in the array object, and primitive values of different types require different amounts of storage. For example, the type **int** requires 32 bits of storage while the type **double** requires 64 bits of storage.

Would require reconstructing the array object

Therefore, to convert an array object containing **int** values to an array object containing **double** values would require reconstructing the array object and allocating twice as much storage space for each element in the array.

Restriction doesn't apply to arrays of references

As you have seen from previous questions, such a casting restriction does not apply to arrays containing references to objects. This may be because the amount of storage required to store a reference to an object is the same, regardless of the type of the object. Therefore, the allowable casts that you have seen in the previous questions did not require any change to the size of the array. All that changed was some supplemental information regarding the type of objects to which the elements in the array refer.

Back to Question 11 (p. 146)

7.6.6 Answer 10

C. 1.0 2.0

7.6.6.1 Explanation 10

Assigning array reference to variable of type Object

A reference to an array can be assigned to a non-array reference of the class named **Object**, as in the following statement extracted from the program, where A is a reference to an array object of type **double**.

```
Object B = A;
```

Note that there are no square brackets anywhere in the above statement. Thus, the reference to the array object is not being assigned to an array reference of the type **Object[]**. Rather, it is being assigned to an ordinary reference variable of the type **Object**.

Downcasting to an array type

Once the array reference has been assigned to the ordinary reference variable of the type **Object**, that reference variable can be downcast and used to access the individual elements in the array as illustrated in the following fragment. Note the empty square brackets in the syntax of the cast operator **(double[])**.

```
System.out.println(
       ((double[])B)[0] + " " +
       ((double[])B)[1]);
```

Placement of parentheses is critical

Note also that due to precedence issues, the placement of both sets of parentheses is critical in the above code fragment. You must downcast the reference variable before applying the index to that variable.

Back to Question 10 (p. 145)

7.6.7 Answer 9

C. 1 2

7.6.7.1 Explanation 9

General array casting rule

The general rule for casting array references *(for arrays whose declared type is the name of a class or an interface)* is:

A reference to an array object can be cast to another array type if the elements of the referenced array are of a type that can be cast to the type of the elements of the specified array type.

Old rules apply here also

Thus, the general rules covering conversion and casting up and down the inheritance hierarchy and among classes that implement the same interfaces also apply to the casting of references to array objects.

A reference to an object can be cast down the inheritance hierarchy to the actual class of the object. Therefore, an array reference can also be cast down the inheritance hierarchy to the declared class for the array object.

This program declares a reference to, creates, and populates an array of the class type **Subclass** . This reference is assigned to an array reference of a type that is a superclass of the actual class type of the array. Then the superclass reference is downcast to the actual class type of the array and assigned to a different reference variable. This third reference variable is used to successfully access and display the contents of the elements in the array.

Back to Question 9 (p. 144)

7.6.8 Answer 8

B. Runtime Error

7.6.8.1 Explanation 8

Another ClassCastException

While it is allowable to assign an array reference to an array reference variable declared for a class that is further up the inheritance hierarchy *(as illustrated earlier)* , it is not allowable to cast an array reference down the inheritance hierarchy to a subclass of the original declared class for the array.

This program declares a reference for, creates, and populates a two-element array for a class named **Superclass** . Then it downcasts that reference to a subclass of the class named **Superclass** . The compiler is unable to determine that this is a problem. However, the runtime system throws the following exception, which terminates the program at runtime.

```
   java.lang.ClassCastException: [LSuperclass;
at Worker.doArrays(Ap071.java:19)
at Ap071.main(Ap071.java:9)
```

Back to Question 8 (p. 143)

7.6.9 Answer 7

C. 1 2

7.6.9.1 Explanation 7

Assignment to superclass array reference variable

This program illustrates that, if you have a reference to an array object containing references to other objects, you can assign the array object's reference to an array reference variable whose type is a superclass

of the declared class of the array object. *(As we will see later, this doesn't work for array objects containing primitive values.)*

What can you do then?

Having made the assignment to the superclass reference variable, whether or not you can do anything useful with the elements in the array *(without downcasting)* depends on many factors.

No downcast required in this case

In this case, the ability to display the contents of the objects referred to in the array was inherited from the class named **Superclass** . Therefore, it is possible to access and display a **String** representation of the objects without downcasting the array object reference from **Superclass** to the actual type of the objects.

Probably need to downcast in most cases

However, that will often not be the case. In most cases, when using a reference of a superclass type, you will probably need to downcast in order to make effective use of the elements in the array object.

Back to Question 7 (p. 141)

7.6.10 Answer 6

C. 1 2

7.6.10.1 Explanation 6

Straightforward array application

This is a straightforward application of Java array technology for the storage and retrieval of references to objects.

The program declares a reference to, creates, and populates a two-element array of a class named **Subclass** . The class named **Subclass** extends the class named **Superclass** , which in turn, extends the class named **Object** by default.

The super keyword

The class named **Subclass** doesn't do anything particularly useful other than to illustrate extending a class.

However, it also provides a preview of the use of the **super** keyword for the purpose of causing a constructor in a subclass to call a parameterized constructor in its superclass.

Setting the stage for follow-on questions

The main purpose for showing you this program is to set the stage for several programs that will be using this class structure in follow-on questions.

Back to Question 6 (p. 140)

7.6.11 Answer 5

B. Runtime Error

7.6.11.1 Explanation 5

ClassCastException

There are some situations involving casting where the compiler cannot identify an erroneous condition that is later identified by the runtime system. This is one of those cases.

This program begins with an array of type **double** []. The reference to that array is converted to type **Object** . Then it is cast to type **String** []. All of these operations are allowed by the compiler.

However, at runtime, the runtime system expects to find references to objects of type **String** in the elements of the array. What it finds instead is values of type **double** stored in the elements of the array.

As a result, a **ClassCastException** is thrown. Since it isn't caught and handled by the program, the program terminates with the following error message showing on the screen.

```
    java.lang.ClassCastException: [D
  at Worker.doArrays(Ap068.java:17)
  at Ap068.main(Ap068.java:6)
```

Back to Question 5 (p. 139)

7.6.12 Answer 4

C. 1.0 2.0

7.6.12.1 Explanation 4

Finally, we got it right

Finally, we managed to get it all together. The program compiles and executes correctly. This program illustrates the assignment of an array object's reference to a reference variable of type **Object** , and the casting of that reference of type **Object** back to the correct array type in order to gain access to the elements in the array.

But don't go away, there is a lot more that you need to know about arrays in Java. We will look at some of those things in the questions that follow.

Back to Question 4 (p. 138)

7.6.13 Answer 3

A. Compiler Error

7.6.13.1 Explanation 3

Must use the correct cast syntax

While it is possible to store an array object's reference in a reference variable of type **Object** , and later cast it back to an array type to gain access to the elements in the array, you must use the correct syntax in performing the cast. This is not the correct syntax for performing that cast. It is missing the empty square brackets required to indicate a reference to an array object.

A portion of the compiler error produced by JDK 1.3 is shown below:

```
    Ap066.java:17: inconvertible types
found   : java.lang.Object
required: double
    double C = (double)B;
```

Back to Question 3 (p. 137)

7.6.14 Answer 2

A. Compiler Error

7.6.14.1 Explanation 2

Must cast back to an array type

This program illustrates another very important point. Although you can assign an array object's reference to a reference variable of type **Object**, you cannot gain access to the elements in the array while treating it as type **Object**. Instead, you must cast it back to an array type before you can gain access to the elements in the array object.

A portion of the compiler error produced by JDK 1.3 is shown below:

```
    Ap065.java:18: array required, but java.lang.Object found
B[0] + " " + B[1]);
```

Back to Question 2 (p. 136)

7.6.15 Answer 1

C. I'm OK

7.6.15.1 Explanation 1

Assigning array reference to type Object

This program illustrates a very important point. You can assign an array object's reference to an ordinary reference variable of type **Object**. Note that I didn't say **Object[]**. The empty square brackets are not required when the type is **Object**.

Standard containers or collections

Later on, when we study the various containers in the Java class libraries *(see the Java Collections Framework)*, we will see that they store references to all objects, including array objects, as type **Object**. Thus, if it were not possible to store a reference to an array object in a reference variable of type **Object**, it would not be possible to use the standard containers to store references to array objects.

Because it is possible to assign an array object's reference to a variable of type **Object**, it is also possible to store array object references in containers of type **Object**.

Back to Question 1 (p. 135)

-end-

Chapter 8

Ap0070: Self-assessment, Method Overloading[1]

8.1 Table of Contents

8.2 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. 170) to easily find and view the listings while you are reading about them.

8.3 Questions

8.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 162) ?

- A. Compiler Error
- B. Runtime Error
- C. 9 17.64
- D. None of the above

[1] This content is available online at <http://cnx.org/content/m45276/1.5/>.

Listing 1 . Listing for Question 1.

```
    public class Ap079{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int x = 3;
    double y = 4.2;
    System.out.println(square(x) + " "
                         + square(y));
  }//end doOverLoad()

  public int square(int y){
    return y*y;
  }//end square()

  public double square(double y){
    return y*y;
  }//end square()
}// end class
```

Table 8.1

Answer and Explanation (p. 177)

8.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 163) ?

- A. Compiler Error
- B. Runtime Error
- C. float 9.0 double 17.64
- D. None of the above

Listing 2 . Listing for Question 2.

```
    public class Ap080{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int x = 3;
    double y = 4.2;

    System.out.print(square(x) + " ");
    System.out.print(square(y));
    System.out.println();
  }//end doOverLoad()

  public float square(float y){
    System.out.print("float ");
    return y*y;
  }//end square()

  public double square(double y){
    System.out.print("double ");
    return y*y;
  }//end square()
}// end class
```

Table 8.2

Answer and Explanation (p. 176)

8.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 164) ?

- A. Compiler Error
- B. Runtime Error
- C. 10 17.64
- D. None of the above

Listing 3 . Listing for Question 3.

continued on next page

```
    public class Ap081{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    double w = 3.2;
    double x = 4.2;

    int y = square(w);
    double z = square(x);

    System.out.println(y + " " + z);
  }//end doOverLoad()

  public int square(double y){
    return (int)(y*y);
  }//end square()

  public double square(double y){
    return y*y;
  }//end square()

}// end class
```

Table 8.3

Answer and Explanation (p. 176)

8.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 165) ?

- A. Compiler Error
- B. Runtime Error
- C. 9 17.64
- D. None of the above

Listing 4 . Listing for Question 4.

continued on next page

```
    public class Ap083{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int w = 3;
    double x = 4.2;

    System.out.println(
      new Subclass().square(w) + " "
      + new Subclass().square(x));
  }//end doOverLoad()
}// end class

class Superclass{
  public double square(double y){
    return y*y;
  }//end square()
}//end class Superclass

class Subclass extends Superclass{
  public int square(int y){
    return y*y;
  }//end square()
}//end class Subclass
```

Table 8.4

Answer and Explanation (p. 175)

8.3.5 Question 5

Which of the following is produced by the program shown in Listing 5 (p. 167) ?

A. Compiler Error

B. Runtime Error

C. float 2.14748365E9
   float 9.223372E18
   double 4.2

D. None of the above

Listing 5 . Listing for Question 5.

```
    public class Ap084{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    int x = 2147483647;
    square(x);
    long y = 9223372036854775807L;
    square(y);
    double z = 4.2;
    square(z);

    System.out.println();
  }//end doOverLoad()

  public void square(float y){
    System.out.println("float" + " " +
                               y + " ");
  }//end square()

  public void square(double y){
    System.out.println("double" + " " +
                               y + " ");
  }//end square()
}// end class
```

Table 8.5

Answer and Explanation (p. 173)

8.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 168) ?

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

Listing 6 . Listing for Question 6.

```
    public class Ap085{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    Test a = new Test();
    DumIntfc b = new Test();
    overLoadMthd(a);
    overLoadMthd(b);
    System.out.println();
  }//end doOverLoad()

  public void overLoadMthd(Test x){
    System.out.print("Test ");
  }//end overLoadMthd

  public void overLoadMthd(DumIntfc x){
    System.out.print("DumIntfc ");
  }//end overLoadMthd
}// end class

interface DumIntfc{
}//end DumIntfc

class Test implements DumIntfc{
}//end class Test
```

Table 8.6

Answer and Explanation (p. 172)

8.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 169) ?

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

Listing 7 . Listing for Question 7.

```
    public class Ap086{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    Test a = new Test();
    Object b = new Test();
    overLoadMthd(a);
    overLoadMthd(b);
    System.out.println();
  }//end doOverLoad()

  public void overLoadMthd(Test x){
    System.out.print("Test ");
  }//end overLoadMthd

  public void overLoadMthd(Object x){
    System.out.print("Object ");
  }//end overLoadMthd

}// end class

class Test{
}//end class Test
```

Table 8.7

Answer and Explanation (p. 172)

8.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 170) ?

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

Listing 8 . Listing for Question 8.

```
    public class Ap087{
  public static void main(
                        String args[]){
    new Worker().doOverLoad();
  }//end main()
}//end class definition

class Worker{
  public void doOverLoad(){
    SubC a = new SubC();
    SuperC b = new SubC();

    SubC obj = new SubC();
    obj.overLoadMthd(a);
    obj.overLoadMthd(b);

    System.out.println();
  }//end doOverLoad()

}// end class

class SuperC{
  public void overLoadMthd(SuperC x){
    System.out.print("SuperC ");
  }//end overLoadMthd
}//end SuperC

class SubC extends SuperC{
  public void overLoadMthd(SubC x){
    System.out.print("SubC ");
  }//end overLoadMthd
}//end class SubC
```

Table 8.8

Answer and Explanation (p. 171)

8.4 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.

8.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0070: Self-assessment, Method Overloading
- File: Ap0070.htm
- Originally published: 2002
- Published at cnx.org: 12/04/12
- Revised: 12/03/14

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.

8.6 Answers

8.6.1 Answer 8

C. SubC SuperC

8.6.1.1 Explanation 8

While admittedly a little convoluted, this is another relatively straightforward application of method overloading using types from the class hierarchy.

Type **SubC** , **SuperC** , or **Object?**

This method defines a class named **SuperC** , which extends **Object** , and a class named **SubC** , which extends **SuperC** . Therefore, an object instantiated from the class named **SubC** can be treated as any of the following types: **SubC** , **SuperC** , or **Object** .

Two overloaded methods in different classes

Two overloaded methods named **overLoadMthd** are defined in two classes in the inheritance hierarchy. The class named **SuperC** defines a version that requires an incoming parameter of type **SuperC** . The class named **SubC** defines a version that requires an incoming parameter of type **SubC** . When called, each of these overloaded methods prints the type of its formal argument.

Two objects of type SubC

The program instantiates two objects of the **SubC** class, storing the reference to one of them in a reference variable of type **SubC**, and storing the reference to the other in a reference variable of type **SuperC**.

Call the overloaded method twice

The next step is to call the overloaded method named **overLoadMthd** twice in succession, passing each of the reference variables of type **SubC** and **SuperC** to the method.

Instance methods require an object

Because the two versions of the overloaded method are instance methods, it is necessary to have an object on which to call the methods. This is accomplished by instantiating a new object of the **SubC** class, storing the reference to that object in a reference variable named **obj**, and calling the overloaded method on that reference.

Overloaded methods not in same class

The important point here is that the two versions of the overloaded method were not defined in the same class. Rather, they were defined in two different classes in the inheritance hierarchy. However, they were defined in such a way that both overloaded versions were contained as instance methods in an object instantiated from the class named **SubC**.

No surprises

There were no surprises. When the overloaded method was called twice in succession, passing the two different reference variables as parameters, the output shows that the version that was called in each case had a formal argument type that matched the type of the parameter that was passed to the method.

Back to Question 8 (p. 169)

8.6.2 Answer 7

C. Test Object

8.6.2.1 Explanation 7

Another straightforward application

This is another straightforward application of method overloading, which produces no surprises.

This program defines a new class named **Test**, which extends the **Object** class by default. This means that an object instantiated from the class named **Test** can be treated either as type **Test**, or as type **Object**.

The program defines two overloaded methods named **overLoadMthd**. One requires an incoming parameter of type **Test**. The other requires an incoming parameter of type **Object**. When called, each of these methods prints the type of its incoming parameter.

The program instantiates two different objects of the class **Test**, storing a reference to one of them in a reference variable of type **Test**, and storing a reference to the other in a reference variable of type **Object**.

No surprises here

Then it calls the overloaded **overLoadMthd** method twice in succession, passing the reference of type **Test** during the first call, and passing the reference of type **Object** during the second call.

As mentioned above, the output produces no surprises. The output indicates that the method selected for execution during each call is the method with the formal argument type that matches the type of parameter passed to the method.

Back to Question 7 (p. 168)

8.6.3 Answer 6

C. Test DumIntfc

8.6.3.1 Explanation 6

Overloaded methods with reference parameters

This is a fairly straightforward application of method overloading. However, rather than requiring method parameters of primitive types as in the previous questions in this module, the overloaded methods in this program require incoming parameters of class and interface types respectively.

Type Test or type DumIntfc?

The program defines an interface named **DumIntfc** and defines a class named **Test** that implements that interface. The result is that an object instantiated from the **Test** class can be treated either as type **Test** or as type **DumIntfc** *(it could also be treated as type Object as well)* .

Two overloaded methods

The program defines two overloaded methods named **overLoadMthd** . One requires an incoming parameter of type **Test** , and the other requires an incoming parameter of type **DumIntfc** . When called, each of the overloaded methods prints a message indicating the type of its argument.

Two objects of the class Test

The program instantiates two objects of the class **Test** . It assigns one of the object's references to a reference variable named **a** , which is declared to be of type **Test** .

The program assigns the other object's reference to a reference variable named **b** , which is declared to be of type **DumIntfc** . *(Remember, both objects were instantiated from the class **Test** .)*

No surprises here

Then it calls the overloaded method named **overLoadMthd** twice in succession, passing first the reference variable of type **Test** and then the reference variable of type **DumIntfc** .

The program output doesn't produce any surprises. When the reference variable of type **Test** is passed as a parameter, the overloaded method requiring that type of parameter is selected for execution. When the reference variable of type **DumIntfc** is passed as a parameter, the overloaded method requiring that type of parameter is selected for execution.

Back to Question 6 (p. 167)

8.6.4 Answer 5

```
C.  float 2.14748365E9
float 9.223372E18
double 4.2
```

8.6.4.1 Explanation 5

Another subtle method selection issue

This program illustrates a subtle issue in the automatic selection of an overloaded method based on assignment compatibility.

This program defines two overloaded methods named **square** . One requires an incoming parameter of type **float** , and the other requires an incoming parameter of type **double** .

When called, each of these methods prints the type of its formal argument along with the value of the incoming parameter as represented by its formal argument type. In other words, the value of the incoming parameter is printed after it has been automatically converted to the formal argument type.

Printout identifies the selected method

This printout makes it possible to determine which version is called for different types of parameters. It also makes it possible to determine the effect of the automatic conversion on the incoming parameter. What we are going to see is that the conversion process can introduce serious accuracy problems.

Call the method three times

The **square** method is called three times in succession, passing values of type **int** , **long** , and **double** during successive calls.

(Type ***long*** *is a 64-bit integer type capable of storing integer values that are much larger than can be stored in type* ***int*** *. The use of this type here is important for illustration of data corruption that occurs through automatic type conversion.)*

The third invocation of the **square** method, passing a **double** as a parameter, is not particularly interesting. There is a version of **square** with a matching argument type, and everything behaves as would be expected for this invocation. The interesting behavior occurs when the **int** and **long** values are passed as parameters.

Passing an int parameter

The first thing to note is the behavior of the program produced by the following code fragment.

```
    int x = 2147483647;
square(x);
```

The above fragment assigns a large integer value (2147483647)to the **int** variable and passes that variable to the **square** method. This fragment produces the following output on the screen:

```
    float 2.14748365E9
```

As you can see, the system selected the overloaded method that requires an incoming parameter of type **float** for execution in this case *(rather than the version that requires type* ***double*** *).*

Conversion from int to float loses accuracy

Correspondingly, it converted the incoming **int** value to type **float** , losing one decimal digit of accuracy in the process. *(The original* ***int*** *value contained ten digits of accuracy. This was approximated by a nine-digit* ***float*** *value with an exponent value of 9.)*

This seems like an unfortunate choice of overloaded method. Selecting the other version that requires a **double** parameter as input would not have resulted in any loss of accuracy.

A more dramatic case

Now, consider an even more dramatic case, as illustrated in the following fragment where a very large **long** integer value(9223372036854775807) is passed to the **square** method.

```
    long y = 9223372036854775807L;
square(y);
```

The above code fragment produced the following output:

```
    float 9.223372E18
```

A very serious loss of accuracy

Again, unfortunately, the system selected the version of the **square** method that requires a **float** parameter for execution. This caused the **long** integer to be converted to a **float** . As a result, the **long** value containing 19 digits of accuracy was converted to an estimate consisting of only seven digits plus an exponent. *(Even if the overloaded* ***square*** *method requiring a* ***double*** *parameter had been*

selected, the conversion process would have lost about three digits of accuracy, but that would have been much better than losing twelve digits of accuracy.)

The moral to the story is ...

Don't assume that just because the system knows how to automatically convert your integer data to floating data, it will protect the integrity of your data. Oftentimes it won't.

To be really safe ...

To be really safe, whenever you need to convert either **int** or **long** types to floating format, you should write your code in such a way as to ensure that it will be converted to type **double** instead of type **float** .

For example, the following modification would solve the problem for the **int** data and would greatly reduce the magnitude of the problem for the **long** data. Note the use of the **(double)** cast to force the **double** version of the **square** method to be selected for execution.

```
    int x = 2147483647;
square((double)x);
long y = 9223372036854775807L;
square((double)y);
```

The above modification would cause the program to produce the following output:

```
    double 2.147483647E9
double 9.223372036854776E18
double 4.2
```

This output shows no loss of accuracy for the **int** value, and the loss of three digits of accuracy for the long value.

*(Because a **long** and a **double** both store their data in 64 bits, it is not possible to convert a very large **long** value to a **double** value without some loss in accuracy, but even that is much better than converting a 64-bit **long** value to a 32-bit **float** value.)*

Back to Question 5 (p. 165)

8.6.5 Answer 4

C. 9 17.64

8.6.5.1 Explanation 4

When the **square** method is called on an object of the **Subclass** type passing an **int** as a parameter, there is an exact match to the required parameter type of the **square** method defined in that class. Thus, the method is properly selected and executed.

When the **square** method is called on an object of the **Subclass** type passing a **double** as a parameter, the version of the **square** method defined in the **Subclass** type is not selected. The **double** value is not assignment compatible with the required type of the parameter *(an **int** is narrower than a **double**)*.

Having made that determination, the system continues searching for an overloaded method with a required parameter that is either type **double** or assignment compatible with **double** . It finds the version inherited from **Superclass** that requires a **double** parameter and calls it.

The bottom line is, overloaded methods can occur up and down the inheritance hierarchy.

Back to Question 4 (p. 164)

8.6.6 Answer 3

A. Compiler Error

8.6.6.1 Explanation 3

Return type is not a differentiating feature

This is not a subtle issue. This program illustrates the important fact that the return type does not differentiate between overloaded methods having the same name and formal argument list.

For a method to be overloaded, two or more versions of the method must have the same name and different formal arguments lists.

The return type can be the same, or it can be different *(it can even be void)* . It doesn't matter.

These two methods are not a valid overload

This program attempts to define two methods named **square** , each of which requires a single incoming parameter of type **double** . One of the methods casts its return value to type **int** and returns type **int** . The other method returns type **double** .

The JDK 1.3 compiler produced the following error:

```
    Ap081.java:28: square(double) is already defined
in Worker

  public double square(double y){
```

Back to Question 3 (p. 163)

8.6.7 Answer 2

C. float 9.0 double 17.64

8.6.7.1 Explanation 2

This program is a little more subtle

Once again, the program defines two overloaded methods named **square** . However, in this case, one of the methods requires a single incoming parameter of type **float** and the other requires a single incoming parameter of type **double** . *(Suffice it to say that the* ***float*** *type is similar to the* ***double*** *type, but with less precision. It is a floating type, not an integer type. The* ***double*** *type is a 64-bit floating type and the* ***float*** *type is a 32-bit floating type.)*

Passing a type int as a parameter

This program does not define a method named **square** that requires an incoming parameter of type **int** . However, the program calls the **square** method passing a value of type **int** as a parameter.

What happens to the int parameter?

The first question to ask is, will this cause one of the two overloaded methods to be called, or will it cause a compiler error? The answer is that it will cause one of the overloaded methods to be called because a value of type **int** is assignment compatible with both type **float** and type **double** .

Which overloaded method will be called?

Since the type **int** is assignment compatible with type **float** and also with type **double** , the next question is, which of the two overloaded methods will be called when a value of type **int** is passed as a parameter?

Learn through experimentation

I placed a print statement in each of the overloaded methods to display the type of that method's argument on the screen when the method is called. By examining the output, we can see that the method

with the **float** parameter was called first *(corresponding to the parameter of type* ***int*** *)*. Then the method with the **double** parameter was called *(corresponding to the parameter of type* ***double*** *)*.

Converted int to float

Thus, the system selected the overloaded method requiring an incoming parameter of type **float** when the method was called passing an **int** as a parameter. The value of type **int** was automatically converted to type **float** .

In this case, it wasn't too important which method was called to process the parameter of type **int** , because the two methods do essentially the same thing – compute and return the **square** of the incoming value.

However, if the behavior of the two methods were different from one another, it could make a lot of difference, which one gets called on an assignment compatible basis. *(Even in this case, it makes some difference. As we will see later, when a very large* ***int*** *value is converted to a* ***float*** *, there is some loss in accuracy. However, when the same very large* ***int*** *value is converted to a* ***double*** *, there is no loss in accuracy.)*

Avoiding the problem

One way to avoid this kind of subtle issue is to avoid passing assignment-compatible values to overloaded methods.

Passing assignment-compatible values to overloaded methods allows the system to resolve the issue through automatic type conversion. Automatic type conversion doesn't always provide the best choice.

Using a cast to force your choice of method

Usually, you can cast the parameter values to a specific type before calling the method and force the system to select your overloaded method of choice.

For example, in this problem, you could force the method with the **double** parameter to handle the parameter of type **int** by using the following cast when the method named **square** is called:

square((double)x)

However, as we will see later, casting may not be the solution in every case.

Back to Question 2 (p. 162)

8.6.8 Answer 1

C. 9 17.64

8.6.8.1 Explanation 1

What is method overloading?

A rigorous definition of method overloading is very involved and won't be presented here. However, from a practical viewpoint, a method is overloaded when two or more methods having the same name and different formal argument lists are defined in the class from which an object is instantiated, or are inherited into an object by way of superclasses of that class.

How does the compiler select among overloaded methods?

The exact manner in which the system determines which method to call in each particular case is also very involved. Basically, the system determines which of the overloaded methods to execute by matching the types of parameters passed to the method to the types of arguments defined in the formal argument list.

Assignment compatible matching

However, there are a number of subtle issues that arise, particularly when there isn't an exact match. In selecting the version of the method to call, Java supports the concept of an "assignment compatible" match *(or possibly more than one assignment compatible match)* .

Briefly, assignment compatibility means that it would be allowable to assign a value of the type that is passed as a parameter to a variable whose type matches the specified argument in the formal argument list.

Selecting the best match

According to *Java Language Reference* by Mark Grand,

"If more than one method is compatible with the given arguments, the method that most closely matches the given parameters is selected. If the compiler cannot select one of the methods as a better match than the others, the method selection process fails and the compiler issues an error message."

Understanding subtleties

If you plan to be a Java programmer, you must have some understanding of the subtle issues involving overloaded methods, and the relationship between *overloaded* methods and *overridden* methods. Therefore, the programs in this module will provide some of that information and discuss some of the subtle issues that arise.

Even if you don't care about the subtle issues regarding method overloading, many of those issues really involve automatic type conversion. You should study these questions to learn about the problems associated with automatic type conversion.

This program is straightforward

However, there isn't anything subtle about the program for Question 1 (p. 161) . This program defines two overloaded methods named **square** . One requires a single incoming parameter of type **int** . The other requires a single incoming parameter of type **double** . Each method calculates and returns the square of the incoming parameter.

The program calls a method named **square** twice in succession, and displays the values returned by those two invocations. In the first case, an **int** value is passed as a parameter. This causes the method with the formal argument list of type **int** to be called.

In the second case, a **double** value is passed as a parameter. This causes the method with the formal argument list of type **double** to be called.

Overloaded methods may have different return types

Note in particular that the overloaded methods have different return types. One method returns its value as type **int** and the other returns its value as type **double** . This is reflected in the output format for the two return vales as shown below:

9 17.64

Back to Question 1 (p. 161)

-end-

Chapter 9

Ap0080: Self-assessment, Classes, Constructors, and Accessor Methods[1]

9.1 Table of Contents

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. 189) to easily find and view the listings while you are reading about them.

9.3 Questions

9.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 180) ?

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

[1] This content is available online at <http://cnx.org/content/m45279/1.5/>.

Listing 1 . Listing for Question 1.

```
    public class Ap090{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = NewClass();
    System.out.println(obj);

  }//end makeObj()


}// end class

class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

Table 9.1

Answer and Explanation (p. 198)

9.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 181) ?

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

Listing 2 . Listing for Question 2.

```
    public class Ap091{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);

  }//end makeObj()

}// end class

Class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

Table 9.2

Answer and Explanation (p. 197)

9.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 182) ?

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

Listing 3 . Listing for Question 3.

```
    public class Ap092{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);

  }//end makeObj()

}// end class

class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

Table 9.3

Answer and Explanation (p. 196)

9.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 183) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 2
- D. None of the above

Listing 4 . Listing for Question 4.

```
    public class Ap093{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public String toString(){
    return "Object containing " + x;
  }//end toString()
}//end NewClass
```

Table 9.4

Answer and Explanation (p. 194)

9.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 184) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 2
- D. None of the above

Listing 5 . Listing for Question 5.

continued on next page

```
    public class Ap094{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    Subclass obj = new Subclass();
    System.out.println(obj);
  }//end makeObj()
}// end class

class Superclass{
  private int x;

  public Superclass(int x){
    this.x = x;
  }//end constructor

  public String toString(){
    return "Object containing " + x;
  }//end toString()

  public void setX(int x){
    this.x = x;
  }//end setX()
}//end Superclass

class Subclass extends Superclass{
  public Subclass(){
    setX(2);
  }//end noarg constructor
}//end Subclass
```

Table 9.5

Answer and Explanation (p. 194)

9.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 185) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 5
- D. Object containing 2
- E. None of the above

Listing 6 . Listing for Question 6.

```
    public class Ap095{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass(5);
    System.out.println(obj);
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(){
  }//end constructor

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public String toString(){
    return "Object containing " + x;
  }//end toString()
}//end NewClass
```

Table 9.6

Answer and Explanation (p. 193)

9.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 186) ?

- A. Compiler Error
- B. Runtime Error
- C. Object containing 0, 0.0, false
- D. Object containing 0.0, 0, true
- E. None of the above

Listing 7 . Listing for Question 7.

```
    public class Ap096{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass();
    System.out.println(obj);
  }//end makeObj()
}// end class

class NewClass{
  private int x;
  private double y;
  private boolean z;

  public String toString(){
    return "Object containing " +
                          x + ", " +
                          y + ", " + z;
  }//end toString()
}//end NewClass
```

Table 9.7

Answer and Explanation (p. 193)

9.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 187) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 5
- E. None of the above

Listing 8 . Listing for Question 8.

continued on next page

```
    public class Ap097{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass(5);
    System.out.println(obj.getX());
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(){
  }//end constructor

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public int getX(){
    return x;
  }//end getX()
}//end NewClass
```

Table 9.8

Answer and Explanation (p. 191)

9.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 188) ?

- A. Compiler Error
- B. Runtime Error
- C. 10
- D. None of the above

Listing 9 . Listing for Question 9.

```
    public class Ap098{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){

    NewClass obj = new NewClass();
    obj.setX(10);
    System.out.println(obj.getX());

  }//end makeObj()
}// end class

class NewClass{
  private int y;

  public void setX(int y){
    this.y = y;
  }//end setX()

  public int getX(){
    return y;
  }//end getX()
}//end NewClass
```

Table 9.9

Answer and Explanation (p. 191)

9.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 189) ?

- A. Compiler Error
- B. Runtime Error
- C. 2
- D. 5
- E. 10
- F. None of the above

Listing 10 . Listing for Question 10.

```
    public class Ap099{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    NewClass obj = new NewClass(5);
    obj.x = 10;
    System.out.println(obj.x);
  }//end makeObj()
}// end class

class NewClass{
  private int x = 2;

  public NewClass(){
  }//end constructor

  public NewClass(int x){
    this.x = x;
  }//end constructor

  public void setX(int x){
    this.x = x;
  }//end setX()

  public int getX(){
    return x;
  }//end getX()
}//end NewClass
```

Table 9.10

Answer and Explanation (p. 190)

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.

9.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0080: Self-assessment, Classes, Constructors, and Accessor Methods
- File: Ap0080.htm
- Originally published: 2002
- Published at cnx.org: 12/05/12
- Revised: 12/03/14

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.

9.6 Answers

9.6.1 Answer 10

A. Compiler Error

9.6.1.1 Explanation 10

Variables declared private really are private

The code in the following fragment attempts to ignore the setter and getter methods and directly access the **private** instance variable named **x** in the object referred to by the reference variable named **obj** .

```
    obj.x = 10;
System.out.println(obj.x);
```

This produces a compiler error. The compiler error produced by JDK 1.3 is reproduced below.

```
    Ap099.java:19: x has private access in
NewClass
    obj.x = 10;
       ^
Ap099.java:20: x has private access in
NewClass
    System.out.println(obj.x);
                       ^
```

Back to Question 10 (p. 188)

9.6.2 Answer 9

C. 10

9.6.2.1 Explanation 9

A setter and a getter

This is a very simple program that uses a setter *(modifier or mutator)* method named **setX** to set the value 10 in a property named **x** that is stored in an instance variable named **y** in an object instantiated from the class named **NewClass** ..

The program also uses a getter *(accessor)* method named **getX** to get and display the value of the property named **x** . *(Note that according to JavaBeans design patterns, the name of the property is unrelated to the name of variable in which the property value is stored.)*

Back to Question 9 (p. 187)

9.6.3 Answer 8

D. 5

9.6.3.1 Explanation 8

Hide your data and expose your methods

For reasons that I won't go into here, good object-oriented design principles state that in almost all cases where an instance variable is not declared to be **final** , it should be declared **private** . *(A final variable behaves like a constant.)*

What is private access?

When an instance variable is declared **private** , it is accessible only by methods of the class in which it is defined. Therefore, the only way that the "outside world" can gain access to a **private** instance variable is by going through an *(usually **public**)* instance method of the object.

Accessor, modifier, mutator, setter, and getter methods

Historically, methods that have been defined for the purpose of exposing **private** instance variables to the outside world have been referred to as *accessor* and *modifier* methods. *(Modifier methods are also sometimes called mutator methods.)*

(Note that since the advent of Sun's JavaBeans Component design patterns, these methods have also come to be known as getter methods and setter methods in deference to the design-pattern naming conventions for the methods.)

A private instance variable with an initializer

The class named **NewClass** declares a **private** instance variable named **x** and initializes its value to 2, as shown in the following code fragment:

```
    private int x = 2;
```

Two constructors

The class contains both a *noarg* constructor and a *parameterized* constructor as shown in the following fragment:

```
    public NewClass(){
}//end constructor

public NewClass(int x){
  this.x = x;
}//end constructor
```

Calling the noarg constructor

If an object of the class is instantiated by calling the *noarg* constructor, the initial value of 2 remains intact, and that object contains an instance variable with an initial value of 2.

Calling the parameterized constructor

If an object of the class is instantiated by calling the parameterized constructor, the initial value of 2 is overwritten by the value of the incoming parameter to the parameterized constructor. In this case, that value is 5, because the object is instantiated by the following code fragment that passes the literal value 5 to the parameterized constructor. Thus, the initial value of the instance variable in that object is 5.

```
    NewClass obj = new NewClass(5);
```

A getter method

Because the instance variable named **x** is **private**, it cannot be accessed directly for display by the code in the **makeObj** method of the **Worker** class. However, the **NewClass** class provides the following public *getter* or *accessor* method that can be used to get the value stored in the instance variable.

(The name of this method complies with JavaBeans design patterns. If you examine the name carefully, you will see why Java programmers often refer to methods like this as getter methods.)

```
    public int getX(){
  return x;
}//end getX()
```

Calling the getter method

Finally, the second statement in the following code fragment calls the getter method on the **NewClass** object to get and display the value of the instance variable named **x**.

```
    NewClass obj = new NewClass(5);
System.out.println(obj.getX());
```

Back to Question 8 (p. 186)

9.6.4 Answer 7

C. Object containing 0, 0.0, false

9.6.4.1 Explanation 7

Default initialization values

The purpose of this question is to confirm that you understand the default initialization of instance variables in an object when you don't write code to cause the initialization of the instance variable to differ from the default.

By default, all instance variables in a new object are initialized with default values if you don't provide a constructor *(or other mechanism)* that causes them to be initialized differently from the default.

- All instance variables of the numeric types are initialized to the value of zero for the type. This program illustrates default initialization to zero for **int** and **double** types.
- Instance variables of type **boolean** are initialized to false.
- Instance variables of type **char** are initialized to a 16-bit Unicode character for which all sixteen bits have been set to zero. I didn't include initialization of the **char** type in the output of this program because the **default** char value is not printable.
- Instance variables of reference types are initialized to null.

Back to Question 7 (p. 185)

9.6.5 Answer 6

C. Object containing 5

9.6.5.1 Explanation 6

A parameterized constructor

This program illustrates the straightforward use of a parameterized constructor.

The class named **NewClass** defines a parameterized constructor that requires an incoming parameter of type **int** .

(For good design practice, the class also defines a noarg constructor, even though it isn't actually used in this program. This makes it available if needed later when someone extends the class.)

Both constructors are shown in the following code fragment.

```
    public NewClass(){
}//end constructor

public NewClass(int x){
  this.x = x;
}//end constructor
```

The parameterized constructor stores its incoming parameter named **x** in an instance variable of the class, also named **x** .

*(The use of the keyword **this** is required in this case to eliminate the ambiguity of having a local parameter with the same name as an instance variable. This is very common Java programming style that you should recognize and understand.)*

Call the parameterized constructor

The following code fragment calls the parameterized constructor, passing the literal **int** value of 5 as a parameter.

```
NewClass obj = new NewClass(5);
```

Hopefully you will have no difficulty understanding the remaining code in the program that causes the value stored in the instance variable named **x** to be displayed on the computer screen.

Back to Question 6 (p. 184)

9.6.6 Answer 5

A. Compiler Error

9.6.6.1 Explanation 5

If you define any constructors, ...

The discussion for Question 4 (p. 182) explained that if you define any constructor in a new class, you must define all constructors that will ever be needed for that class. When you define one or more constructors, the default *noarg* constructor is no longer provided by the system on your behalf.

Question 4 (p. 182) illustrated a simple manifestation of a problem arising from the failure to define a *noarg* constructor that would be needed later. The reason that it was needed later was that the programmer attempted to explicitly use the non-existent *noarg* constructor to create an instance of the class.

A more subtle problem

The problem in this program is more subtle. Unless you *(or the programmer of the superclasses)* specifically write code to cause the system to behave otherwise, each time you instantiate an object of a class, the system automatically calls the *noarg* constructor on superclasses of that class up to and including the class named **Object** . If one or more of those superclasses don't have a *noarg* constructor, unless the author of the subclass constructor has taken this into account, the program will fail to compile.

Calling a non-existing *noarg* constructor

This program attempts to instantiate an object of a class named **Subclass** , which extends a class named **Superclass** . By default, when attempting to instantiate the object, the system will attempt to call a *noarg* constructor defined in **Superclass** .

Superclass has no *noarg* constructor

The **Superclass** class defines a parameterized constructor that requires a single incoming parameter of type **int** . However, it does not also define a *noarg* constructor. Because the parameterized constructor is defined, the default *noarg* constructor does not exist. As a result, JDK 1.3 produces the following compiler error:

```
Ap094.java:40: cannot resolve symbol

symbol  : constructor Superclass  ()
location: class Superclass
  public Subclass(){
```

Back to Question 5 (p. 183)

9.6.7 Answer 4

A. Compiler Error

9.6.7.1 Explanation 4

Constructors

Java uses the following kinds of constructors:

- Those that take arguments, often referred to as *parameterized constructors* , which typically perform initialization on the new object using parameter values.
- Those that don't take arguments, often referred to as *default* or *noarg* constructors, which perform default initialization on the new object.
- Those that don't take arguments but perform initialization on the new object in ways that differ from the default initialization.

Constructor definition is optional

You are not required to define a constructor when you define a new class. If you don't define a constructor for your new class, a default constructor will be provided on your behalf. This constructor requires no argument, and it is typically used in conjunction with the new operator to create an instance of the class using statements such as the following:

```
NewClass obj = new NewClass();
```

The default constructor

The default constructor typically does the following:

- Calls the *noarg* constructor of the superclass
- Assists in the process of allocating and organizing memory for the new object
- Initializes all instance variables of the new object with the following four default values:
  - numeric = 0,
  - boolean = false,
  - char = all zero bits
  - reference = null

Are you satisfied with default values?

As long as you are satisfied with the default initialization of all instance variables belonging to the object, there is no need for you to define a constructor of your own.

However, in the event that you have initialization needs that are not satisfied by the default constructor, you can define your own constructor. Your new constructor may or may not require arguments. *(In case you have forgotten, the name of the constructor is always the same of the name of the class in which it is defined.)*

A non-default *noarg* constructor

If your new constructor doesn't require arguments, you may need to write code that performs initialization in ways that differ from the default initialization. For example, you might decide that a particular **double** instance variable needs to be initialized with a random number each time a new object is instantiated. You could do that with a constructor of your own design that doesn't take arguments by defining the constructor to get initialization values from an object of the **Random** class.

A parameterized constructor

If your new constructor does take arguments, *(a parameterized constructor)* you can define as many overloaded versions as you need. Each overloaded version must have a formal argument list that differs from the formal argument list of all of the other overloaded constructors for that class.

(The rules governing the argument list for overloaded constructors are similar to the rules governing the argument list for overloaded methods, which were discussed in a previous module.)

Use parameter values for initialization

In this case, you will typically define your parameterized constructors to initialize some or all of the instance variables of the new object using values passed to the constructor as parameters.

What else can a constructor do?

You can also cause your new constructor to do other things if you so choose. For example, if you know how to do so, you could cause your constructor *(with or without parameters)* to play an audio clip each time a new object is instantiated. You could use a parameter to determine which audio clip to play in each particular instance.

The punch line

So far, everything that I have said is background information for this program. Here is the punch line insofar as this program is concerned.

If you define any constructor in your new class, you must define all constructors that your new class will ever need.

If you define any constructor, the default constructor is no longer provided on your behalf. If your new class needs a *noarg* constructor *(and it probably does, but that may not become apparent until later when you or someone else extends your class)* you must define the *noarg* version in addition to the other overloaded versions that you define.

A violation of the rule

This program violated the rule given above. It defined the parameterized constructor for the class named **NewClass** shown below

```
  public NewClass(int x){
this.x = x;
}//end constructor
```

However, the program did not also define a *noarg* constructor for the **NewClass** class.

Calling the *noarg* constructor

The code in the **makeObj** method of the **Worker** class attempted to instantiate a new object using the following code:

```
NewClass obj = new NewClass();
```

Since the class definition didn't contain a definition for a *noarg* constructor, the following compiler error was produced by JDK 1.3.

```
    Ap093.java:18: cannot resolve symbol
symbol  : constructor NewClass
()
location: class NewClass
    NewClass obj = new NewClass();
```

Back to Question 4 (p. 182)

9.6.8 Answer 3

C. An Object

9.6.8.1 Explanation 3

We finally got it right!

Did you identify the errors in the previous two programs before looking at the answers?

This program declares the class named **NewClass** correctly and uses the **new** operator correctly in conjunction with the default *noarg* constructor for the **NewClass** class to create a new instance of the class *(an object)* .

Making the class public

One of the things that I could do differently would be to make the declaration for the **NewClass** class public *(as shown in the following code fragment)* .

```
  public class NewClass{
public String toString(){
  return "An Object";
}//end toString()
}//end NewClass
```

I am a little lazy

The reason that I didn't declare this class **public** *(and haven't done so throughout this series of modules)* is because the source code for all **public** classes and interfaces must be in separate files. While that is probably a good requirement for large programming projects, it is overkill for simple little programs like I am presenting in this group of self-assessment modules.

Dealing with multiple files

Therefore, in order to avoid the hassle of having to deal with multiple source code files for every program, I have been using *package-private* access for class definitions other than the controlling class *(the controlling class is declared public)* . Although I won't get into the details at this point, when a class is not declared public, it is common to say that it has *package-private* access instead of **public** access.

Back to Question 3 (p. 181)

9.6.9 Answer 2

A. Compiler Error

9.6.9.1 Explanation 2

Java is a case-sensitive language

Java keywords must be written exactly as specified. The keyword **class** cannot be written as *Class* , which is the problem with this program.

The inappropriate use of the upper-case C in the word *Class* caused the following compiler error.

```
Ap091.java:25: 'class' or 'interface' expected
```

```
Class NewClass{
```

The solution to the problem

This problem can be resolved by causing the first character in the keyword **class** to be a lower-case character as shown in the following code fragment.

```
    class NewClass{
  public String toString(){
    return "An Object";
  }//end toString()
}//end NewClass
```

Back to Question 2 (p. 180)

9.6.10 Answer 1

A. Compiler Error

9.6.10.1 Explanation 1

Instantiating an object

There are several ways to instantiate an object in Java:

- Use the **newInstance** method of the class named **Class** .
- Reconstruct a serialized object using an I/O **readObject** method.
- Create an initialized array object such as {1,2,3}.
- Create a **String** object from a literal string such as "A String".
- Use the **new** operator.

Of all of these, the last two are by far the most common.

What you cannot do!

You cannot instantiate a new object using code like the following code fragment that was extracted from this program.

```
NewClass obj = NewClass();
```

This program produces the following compiler error:

```
    Ap090.java:18: cannot resolve symbol
symbol  : method NewClass  ()
location: class Worker
    NewClass obj = NewClass();
```

The solution to the problem

This problem can be solved by inserting the **new** operator to the left of the constructor as shown in the following code fragment.

```
NewClass obj = new NewClass();
```

Back to Question 1 (p. 179)

-end-

Chapter 10

Ap0090: Self-assessment, the super keyword, final keyword, and static methods[1]

10.1 Table of Contents

10.2 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. 208) to easily find and view the listings while you are reading about them.

10.3 Questions

10.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 200) ?

- A. Compiler Error
- B. Runtime Error
- C. 1, 2

[1] This content is available online at <http://cnx.org/content/m45270/1.5/>.

- D. 5, 10
- E. None of the above

Listing 1 . Listing for Question 1.

```
    public class Ap100{
  public static void main(
                         String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    Subclass obj = new Subclass();
    System.out.println(obj.getX() +
                     ", " + obj.getY());
  }//end makeObj()
}// end class

class Superclass{
  private int x = 1;

  public Superclass(){
    x = 5;
  }//end constructor

  public int getX(){
    return x;
  }//end getX()
}//end Superclass

class Subclass extends Superclass{
  private int y = 2;

  public Subclass(){
    super();
    y = 10;
  }//end constructor

  public int getY(){
    return y;
  }//end getY()
}//end Subclass
```

Table 10.1

Answer and Explanation (p. 216)

10.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 202) ?

- A. Compiler Error
- B. Runtime Error
- C. 1, 2
- D. 5, 2
- E. 5, 10
- F. 20, 10
- G. None of the above

Listing 2 . Listing for Question 2.

```
    public class Ap101{
  public static void main(
                        String args[]){
    new Worker().makeObj();
  }//end main()
}//end class definition

class Worker{
  public void makeObj(){
    Subclass obj = new Subclass();
    System.out.println(obj.getX() +
                     ", " + obj.getY());
  }//end makeObj()
}// end class

class Superclass{
  private int x = 1;

  public Superclass(){
    x = 5;
  }//end constructor

  public Superclass(int x){
    this.x = x;
  }//end constructor

  public int getX(){
    return x;
  }//end getX()
}//end Superclass

class Subclass extends Superclass{
  private int y = 2;

  public Subclass(){
    super(20);
    y = 10;
  }//end constructor

  public int getY(){
    return y;
  }//end getY()
}//end Subclass
```

Table 10.2

Answer and Explanation (p. 215)

10.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 203) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 3 . Listing for Question 3.

```
    public class Ap102{
  public static void main(
                        String args[]){
    new Worker().finalStuff();
  }//end main()
}//end class definition

class Worker{
  public void finalStuff(){
    final int x = 5;
    x = 10;
    System.out.println(x);
  }//end finalStuff()
}// end class
```

Table 10.3

Answer and Explanation (p. 214)

10.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 204) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 4 . Listing for Question 4.

```
    public class Ap103{
  public static void main(
                        String args[]){
    new Worker().finalStuff();
  }//end main()
}//end class definition

class Worker{
  public void finalStuff(){
    public final int x = 5;
    System.out.println(x);
  }//end finalStuff()
}// end class
```

Table 10.4

Answer and Explanation (p. 213)

10.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 204) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 5 . Listing for Question 5.

```
    public class Ap104{
  public static void main(
                        String args[]){
    new Worker().finalStuff();
  }//end main()
}//end class definition

class Worker{
  void finalStuff(){
    final int x = 5;
    System.out.println(x);
  }//end finalStuff()
}// end class
```

Table 10.5

Answer and Explanation (p. 212)

10.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 205) ?

- A. Compiler Error
- B. Runtime Error
- C. 3.141592653589793
- D. 3.1415927
- E. None of the above

Listing 6 . Listing for Question 6.

```
   public class Ap105{
  public static void main(
                       String args[]){
    System.out.println(Worker.fPi);
  }//end main()
}//end class definition

class Worker{
  public static final float fPi =
                       (float)Math.PI;
}// end class
```

Table 10.6

Answer and Explanation (p. 212)

10.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 206) ?

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

Listing 7 . Listing for Question 7.

```
    public class Ap106{
  public static void main(
                        String args[]){
    Worker.staticMethod();
  }//end main()
}//end class definition

class Worker{
  public static void staticMethod(){
    System.out.println(
                     "A static method");
  }//end staticMethod()

}// end class
```

Table 10.7

Answer and Explanation (p. 211)

10.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 206) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 8 . Listing for Question 8.

```
    public class Ap107{
  public static void main(
                        String args[]){
    Worker.staticMethod();
  }//end main()
}//end class Ap107

class Worker{
  private int x = 5;
  public static void staticMethod(){
    System.out.println(x);
  }//end staticMethod()

}// end class
```

Table 10.8

Answer and Explanation (p. 211)

10.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 207) ?

- A. Compiler Error
- B. Runtime Error
- C. 5
- D. None of the above

Listing 9 . Listing for Question 9.

```
    public class Ap108{
  public static void main(
                        String args[]){
    Worker.staticMethod();
  }//end main()
}//end class Ap108

class Worker{
  private int x = 5;
  public static void staticMethod(){
    System.out.println(
                  new Worker().getX());
  }//end staticMethod()

  public int getX(){
    return x;
  }//end getX()

}// end class
```

Table 10.9

Answer and Explanation (p. 210)

10.3.10 Question 10

Which output shown below is produced by the program shown in Listing 10 (p. 208) ?

```
      A.  Compiler Error
  B.  Runtime Error
  C.  38.48451000647496
      12.566370614359172
  D.  None of the above
```

Listing 10 . Listing for Question 10.

```
    public class Ap109{
  public static void main(String args[]){
    System.out.println(Worker.area(3.5));
    System.out.println(Worker.area(2.0));
    System.out.println();
  }//end main()
}//end class Ap109

class Worker{
  public static double area(double r){
    return r*r*Math.PI;
  }//end area()

}// end class
```

Table 10.10

Answer and Explanation (p. 209)

10.4 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.

10.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0090: Self-assessment, the super keyword, final keyword, and static methods

- File: Ap0090.htm
- Originally published: 2002
- Published at cnx.org: 12/05/12
- Revised: 12/03/14

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.

10.6 Answers

10.6.1 Answer 10

```
C.  38.48451000647496
12.566370614359172
```

10.6.1.1 Explanation 10

Use static methods sparingly

Good object-oriented design dictates that **static** methods be used sparingly, and only in those situations where they are appropriate. As you might guess, not all authors will agree on the issue of appropriateness in all cases.

Is this an appropriate use of a static method?

However, I believe that most authors will agree that this program illustrates an appropriate use of a **static** method.

No persistence requirement

This **static** method computes and returns a result on a non-persistent basis. That is to say, there is no attempt by the **static** method to save any historical information from one call of the method to the next. *(Of course, the method that calls the* ***static*** *method can save whatever it chooses to save.)*

Avoiding wasted computer resources

In situations such as this, it would often be a waste of computer resources to require a program to instantiate an object and call an instance method on that object just to be able to delegate a non-persistent computation to that method. *(This is just about as close to a global method as you can get in Java.)*

Computing the area of a circle

In this program, the **Worker** class provides a **static** method named **area** that receives a **double** parameter representing the radius of a circle. It computes and returns the area of the circle as a **double** value. The **static** method named area is shown in the following code fragment.

```
  class Worker{
public static double area(double r){
  return r*r*Math.PI;
```

```
  }//end area()

}// end class
```

As a driver, the **main** method of the controlling class calls the **area** method twice in succession, passing different values for the radius of a circle. In each case, the **main** method receives and displays the value that is returned by the **area** method representing the area of a circle.

Static methods in the class libraries

If you examine the Java API documentation carefully, you will find numerous examples of **static** methods that produce and return something on a non-persistent basis. *(Again, non-persistent in this context means that no attempt is made by the* ***static*** *method to store any historical information. It does a job, forgets it, and goes on to the next job when it is called again.)*

Factory methods

For example, the alphabetical index of the JDK 1.3 API lists several dozen **static** methods named **getInstance** , which are defined in different classes. These methods, which usually produce and return a reference to an object, are often called *factory methods* .

Here is the text from the API documentation describing one of them:

getInstance(int)

Static method in class java.awt.AlphaComposite

Creates an AlphaComposite object with the specified rule.

Back to Question 10 (p. 207)

10.6.2 Answer 9

C. 5

10.6.2.1 Explanation 9

Going through a reference to ...

This program illustrates a rather convoluted methodology by which a **static** method can gain access to an instance member of an object.

```
  class Worker{
private int x = 5;
public static void staticMethod(){
  System.out.println(
               new Worker().getX());
}//end staticMethod()

public int getX(){
  return x;
}//end getX()

}// end class
```

In this example, the **static** method calls a getter method on a reference to an object to gain access to an instance variable belonging to that object. This is what I meant in the discussion in the previous question when I said *"going through a reference to an object of the class."*

Back to Question 9 (p. 207)

10.6.3 Answer 8

A. Compiler Error

10.6.3.1 Explanation 8

A static method cannot access ...

A **static** method cannot access non-static or instance members of its class without going through a reference to an object of the class.

In this program, the **static** method attempts to directly access the instance variable named **x** . As a result, JDK 1.3 produces the following compiler error:

```
    Ap107.java:17: non-static variable x
cannot be referenced from a static context

    System.out.println(x);
```

Back to Question 8 (p. 206)

10.6.4 Answer 7

C. A static method

10.6.4.1 Explanation 7

Using a static method

This is a very straightforward example of the use of a **static** method.

When a method is declared **static** , it is not necessary to instantiate an object of the class containing the method in order to access the method *(although it is possible to do so unless the class is declared abstract)* . All that is necessary to access a **public** **static** method is to refer to the name of the class in which it is defined and the name of the method joined by a period.

(A method that is declared ***static*** *is commonly referred to as a class method. If the method is not declared* ***public*** *, it may not be accessible from your code.)*

Accessing the static method

This is illustrated by the following fragment from the program, with much of the code deleted for brevity.

```
    //...
        Worker.staticMethod();
  //...

class Worker{
  public static void staticMethod(){
    //...
  }//end staticMethod()

}// end class
```

The class named **Worker** defines a **public** **static** method named **staticMethod** . A statement in the **main** method of the controlling class calls the method by referring to the name of the class and the name of the method joined by a period.

When should you use static methods?

Static methods are very useful as utility methods *(getting the absolute value of a number, for example)* .

In my opinion, you should almost never use a **static** method in any circumstance that requires the storage and use of data from one call of the method to the next. In other words, a **static** method may be appropriate for use when it performs a specific task that is completed each time it is called without the requirement for data to persist between calls.

The **Math** class contains many good examples of the use of **static** methods, such as **abs** , **acos** , **asin** , etc.

Back to Question 7 (p. 205)

10.6.5 Answer 6

D. 3.1415927

10.6.5.1 Explanation 6

Using a public static final member variable

The class named **Worker** declares and initializes a member variable named **fPi** .

final

Because it is declared **final** , it is not possible to write code that will change its value after it has been initialized.

static

Because it is declared **static** , it can be accessed without a requirement to instantiate an object of the **Worker** class. All that is necessary to access the variable is to refer to the name of the class and the name of the variable joined by a period.

Because it is **static** , it can also be accessed by **static** methods.

public

Because it is declared **public** , it can be accessed by any code in any method in any object that can locate the class.

Type float is less precise than type double

Because the initialized value is cast from the type **double** that is returned by **Math.PI** to type **float** , an 8-digit approximation is stored in the variable named **fPi** .

The **double** value returned by **Math.PI** is 3.141592653589793

The cast to type **float** reduces the precision down to 3.1415927

Back to Question 6 (p. 205)

10.6.6 Answer 5

C. 5

10.6.6.1 Explanation 5

Using a final local variable

Well, I finally got rid of all the bugs. This program uses a **final** local variable properly. The program compiles and executes without any problems.

Back to Question 5 (p. 204)

10.6.7 Answer 4

A. Compiler Error

10.6.7.1 Explanation 4

The purpose of this question is to see if you are still awake.

What caused the compiler error?

The statement that caused the compiler error in this program is shown below. Now that you know that there was a compiler error, and you know which statement caused it, do you know what caused it?

```
public final int x = 5;
```

Using public static final member variables

As I mentioned in an earlier question, the **final** keyword can be applied either to local variables or to member variables. When applying the **final** keyword to member variables, it is common practice to declare them to be both **public** and **static** in order to make them as accessible as possible. For example, the math class has a **final** variable that is described as follows:

> public static final double PI
>
> The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

The constant named PI

You may recognize the constant named **PI** from your high school geometry class.

Whenever you need the value for the constant **PI**, you shouldn't have to instantiate an object just to get access to it. Furthermore, your class should not be required to have any special package relationship with the **Math** class just to get access to **PI**.

The good news ...

Because **PI** is declared to be both **public** and **static** in the **Math** class, it is readily available to any code in any method in any Java program that has access to the standard Java class library.

How is PI accessed?

PI can be accessed by using an expression as simple as that shown below, which consists simply of the name of the class and the name of the variable joined by a period *(Math.PI)*.

```
double piRSquare = Math.PI * R * R;
```

No notion of public local variables

As a result of the above, many of you may have become accustomed to associating the keyword **public** with the keyword **final**. However, if you missed this question and you have read the explanation to this point, you must also remember that there is no notion of **public** or **private** for local variables. Therefore, when this program was compiled under JDK 1.3, a compiler error was produced. That compiler error is partially reproduced below:

```
    Ap103.java:16: illegal start of
expression
    public final int x = 5;
```

Back to Question 4 (p. 203)

10.6.8 Answer 3

A. Compiler Error

10.6.8.1 Explanation 3

The **final** keyword

The **final** keyword can be applied in a variety of ways in Java. This includes:

- final parameters
- final methods
- final classes
- final variables *(constants)*

Behaves like a constant

When the **final** keyword is applied to a variable in Java, that causes the variable to behave like a constant. In other words, the value of the variable must be initialized when it is declared, and it cannot be changed thereafter *(see the exception discussed below)* .

Apply to local or member variables

The **final** keyword can be applied to either local variables or member variables. *(In case you have forgotten, local variables are declared inside a method or constructor, while member variables are declared inside a class, but outside a method.)*

So, what is the problem?

The problem with this program is straightforward. As shown in the following code fragment, after declaring a **final** local variable and initializing its value to 5, the program attempts to change the value stored in that variable to 10. This is not allowed.

```
        final int x = 5;
    x = 10;
```

A compiler error

JDK 1.3 produces the following error message:

```
    Ap102.java:17: cannot assign a value to
final
variable x


        x = 10;
```

An interesting twist - blank finals

An interesting twist of the use of the **final** keyword with local variables is discussed below.

Background information

Regardless of whether or not the local variable is declared **final** , the compiler will not allow you to access the value in a local variable if that variable doesn't contain a value. This means that you must always either initialize a local variable or assign a value to it before you can access it.

So, what is the twist?

Unlike **final** member variables of a class, the Java compiler and runtime system do not require you to initialize a **final** local variable when you declare it. Rather, you can wait and assign a value to it later.

(Some authors refer to this as a blank final.) However, once you have assigned a value to a **final** local variable, you cannot change that value later.

The bottom line

Whether you initialize the **final** local variable when you declare it, or assign a value to it later, the result is the same. It behaves as a constant. The difference is that if you don't initialize it when you declare it, you cannot access it until after you assign a value to it.

Back to Question 3 (p. 203)

10.6.9 Answer 2

F. 20, 10

10.6.9.1 Explanation 2

Calling a parameterized constructor

This is a relatively straightforward implementation of the use of the **super** keyword in a subclass constructor to call a parameterized constructor in the superclass.

The interesting code in the program is highlighted in the following fragment. Note that quite a lot of code was deleted from the fragment for brevity.

```
  class Superclass{
//...

  public Superclass(int x){
    //...
  }//end constructor

  //...
}//end Superclass

class Subclass extends Superclass{
  //...

  public Subclass(){
    super(20);
    //...
  }//end constructor

  //...
}//end Subclass
```

Using the super keyword

The code that is of interest is the use of **super(20)** as the first executable statement in the **Subclass** constructor to call the parameterized constructor in the superclass, passing a value of 20 as a parameter to the parameterized constructor.

Note that when the **super** keyword is used in this fashion in a constructor, it must be the ***first*** executable statement in the constructor.

As before, the program plays around a little with initial values for instance variables to see if you are alert, but the code that is really of interest is highlighted in the above fragment.

Back to Question 2 (p. 201)

10.6.10 Answer 1

D. 5, 10

10.6.10.1 Explanation 1

The execution of constructors

The purpose of this question and the associated answer is to illustrate explicitly what happens automatically by default regarding the execution of constructors.

The Subclass constructor

This program defines a class named **Subclass** , which extends a class named **Superclass** . A portion of the **Subclass** definition, including its *noarg* constructor is shown in the following code fragment. *(The class also defines a getter method, which was omitted here for brevity.)*

```
  class Subclass extends Superclass{
private int y = 2;

public Subclass(){
  super();
  y = 10;
}//end constructor

//...
}//end Subclass
```

The super keyword

The important thing to note in the above fragment is the statement containing the keyword **super** .

The **super** keyword has several uses in Java. As you might guess from the word, all of those uses have something to do with the superclass of the class in which the keyword is used.

Invoke the superclass constructor

When the **super** keyword *(followed by a pair of matching parentheses)* appears as the first executable statement in a constructor, this is an instruction to the runtime system to first call the constructor for the superclass, and then come back and finish executing the code in the constructor for the class to which the constructor belongs.

Call the *noarg* superclass constructor

If the parentheses following the **super** keyword are empty, this is an instruction to call the *noarg* constructor for the superclass.

Invoke a parameterized superclass constructor

If the parentheses are not empty, this is an instruction to find and call a parameterized constructor in the superclass whose formal arguments match the parameters in the parentheses.

Invoke the *noarg* superclass constructor by default

Here is an important point that is not illustrated above. If the first executable statement in your constructor is not an instruction to call the constructor for the superclass, an instruction to call the *noarg* constructor for the superclass will effectively be inserted into your constructor code before it is compiled.

Therefore, a constructor for the superclass is ***always called*** before the code in the constructor for your new class is executed.

You can choose the superclass constructor

The superclass constructor that is called may be the *noarg* constructor for the superclass, or you can force it to be a parameterized constructor by inserting something like

super(3,x,4.5);

as the first instruction in your constructor definition.

Always have a *noarg* constructor ...

Now you should understand why I told you in an earlier module that the classes you define should almost always have a *noarg* constructor, either the default *noarg* version, or a *noarg* version of your own design.

If your classes don't have a *noarg* constructor, then anyone who extends your classes will be required to put code in the constructor for their new class to call a parameterized constructor in your class.

In this program, the **super();** statement in the **Subclass** constructor causes the *noarg* constructor for the **Superclass** to be called. That *noarg* constructor is shown in the following code fragment.

```
  class Superclass{
private int x = 1;

public Superclass(){
  x = 5;
}//end constructor

//...
}//end Superclass
```

Additional code

Beyond an exposure and explanation of the use of the **super** keyword to call the superclass constructor, this program plays a few games with initial values of instance variables just to see if you are alert to that sort of thing. However, none of that should be new to you, so I won't discuss it further here.

Back to Question 1 (p. 199)

-end-

Chapter 11

Ap0100: Self-assessment, The this keyword, static final variables, and initialization of instance variables[1]

11.1 Table of Contents

11.2 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. 228) to easily find and view the listings while you are reading about them.

11.3 Questions

11.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 220) ?

- A. Compiler Error
- B. Runtime Error
- C. 33

[1] This content is available online at <http://cnx.org/content/m45296/1.4/>.

- D. None of the above

Listing 1 . Listing for Question 1.

```
    public class Ap110{
  public static void main(
                        String args[]){
                new Worker().doThis();
  }//end main()
}//end class Ap110

class Worker{
  private int data = 33;

  public void doThis(){
    new Helper().helpMe(this);
  }//end area()

  public String getData(){
    return data;
  }//end getData()
}// end class Worker

class Helper{
  public void helpMe(Worker param){
    System.out.println(
                      param.getData());
  }//end helpMe()
}//end class Helper
```

Table 11.1

Answer and Explanation (p. 241)

11.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 221) ?

- A. Compiler Error
- B. Runtime Error
- C. 33
- D. None of the above.

Listing 2 . Listing for Question 2.

```
    public class Ap111{
  public static void main(
                        String args[]){
        new Worker().doThis();
  }//end main()
}//end class Ap111

class Worker{
  private int data = 33;

  public void doThis(){
    new Helper().helpMe(this);
  }//end area()

  public String getData(){
    return "" + data;
  }//end getData()
}// end class Worker

class Helper{
  public void helpMe(Worker param){
    System.out.println(
                      param.getData());
  }//end helpMe()
}//end class Helper
```

Table 11.2

Answer and Explanation (p. 239)

11.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 222) ?

- A. Compiler Error
- B. Runtime Error
- C. 11
- D. 22
- E. 33
- F. 44
- G. None of the above.

Listing 3 . Listing for Question 3.

```
    public class Ap112{
  public static void main(
                        String args[]){
    Worker obj1 = new Worker(11);
    Worker obj2 = new Worker(22);
    Worker obj3 = new Worker(33);
    Worker obj4 = new Worker(44);
    obj2.doThis();
  }//end main()
}//end class Ap112

class Worker{
  private int data;

  public Worker(int data){
    this.data = data;
  }//end constructor

  public void doThis(){
    System.out.println(this);
  }//end area()

  public String toString(){
    return "" + data;
  }//end toString()
}// end class Worker
```

Table 11.3

Answer and Explanation (p. 237)

11.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 223) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. 6.283185307179586
- D. None of the above.

Listing 4 . Listing for Question 4.

continued on next page

```
    public class Ap113{
  public static void main(
                        String args[]){
    System.out.println(
                  new Worker().twoPI);
  }//end main()
}//end class Ap113

class Worker{
  public static final double twoPI;

  public Worker(){
    twoPI = 2 * Math.PI;
  }//end constructor
}// end class Worker
```

Table 11.4

Answer and Explanation (p. 237)

11.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 223) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. 6.283185307179586
- D. None of the above.

Listing 5 . Listing for Question 5.

```
    public class Ap114{
  public static void main(
                        String args[]){
    System.out.println(
                  new Worker().twoPI);
  }//end main()
}//end class Ap114

class Worker{
  public static final double twoPI
                         = 2 * Math.PI;
}// end class Worker
```

Table 11.5

Answer and Explanation (p. 236)

11.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 224) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. 6.283185307179586
- D. None of the above.

Listing 6 . Listing for Question 6.

```
   public class Ap115{
  public static void main(
                        String args[]){
    System.out.println(Worker.twoPI);
  }//end main()
}//end class Ap115

class Worker{
  public static final double twoPI
                         = 2 * Math.PI;
}// end class Worker
```

Table 11.6

Answer and Explanation (p. 235)

11.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 225) ?
Note that 6.283185307179586 is a correct numeric value.

- A. Compiler Error
- B. Runtime Error
- C. C. 6.283185307179586
- D. None of the above.

Listing 7 . Listing for Question 7.

```
    public class Ap116{
  public static void main(
                        String args[]){
    System.out.println(Worker.twoPI);
  }//end main()
}//end class Ap116

class Worker{
  public static final double twoPI
                             = 2 * myPI;
  public static final double myPI
                              = Math.PI;
}// end class Worker
```

Table 11.7

Answer and Explanation (p. 234)

11.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 226) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 0.0 false
- D. null null null
- E. None of the above.

Listing 8 . Listing for Question 8.

```
    public class Ap117{
  public static void main(
                        String args[]){
    new Worker().display();
  }//end main()
}//end class Ap117

class Worker{
  private int myInt;
  private double myDouble;
  private boolean myBoolean;

  public void display(){
    System.out.print(myInt);
    System.out.print("  " + myDouble);
    System.out.println(
                     "  " + myBoolean);
  }//end display()

}// end class Worker
```

Table 11.8

Answer and Explanation (p. 233)

11.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 227) ?

- A. Compiler Error
- B. Runtime Error
- C. 0 false 5 true
- D. None of the above.

Listing 9 . Listing for Question 9.

```
    public class Ap118{
  public static void main(
                        String args[]){
    new Worker().display();
    new Worker(5,true).display();
    System.out.println();
  }//end main()
}//end class Ap118

class Worker{
  private int myInt;
  private boolean myBoolean;

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  public void display(){
    System.out.print(myInt);
    System.out.print(
                " " + myBoolean + " ");
  }//end display()

}// end class Worker
```

Table 11.9

Answer and Explanation (p. 232)

11.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 228) ?

- A. Compiler Error
- B. Runtime Error
- C. 20 222.0 false — 5 222.0 true
- D. None of the above.

Listing 10 . Listing for Question 10.

continued on next page

```
    public class Ap119{
  public static void main(
                        String args[]){
    new Worker().display();
    System.out.print("--- ");
    new Worker(5,true).display();
    System.out.println();
  }//end main()
}//end class Ap119

class Worker{
  private int myInt = 100;
  private double myDouble = 222.0;
  private boolean myBoolean;

  public Worker(){
    myInt = 20;
  }//end noarg constructor

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  public void display(){
    System.out.print(myInt + " ");
    System.out.print(myDouble + " ");
    System.out.print(myBoolean + " ");
  }//end display()

}// end class Worker
```

Table 11.10

Answer and Explanation (p. 229)

11.4 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 9 (p. 227) . Listing for Question 9.
- Listing 10 (p. 228) . Listing for Question 10.

11.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0100: Self-assessment, The this keyword, static final variables, and initialization of instance variables
- File: Ap0100.htm
- Originally published: 2004
- Published at cnx.org: 12/08/12
- Revised: 12/03/14

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.

11.6 Answers

11.6.1 Answer 10

C. 20 222.0 false — 5 222.0 true

11.6.1.1 Explanation 10

Four ways to initialize instance variables

There are at least four ways to establish initial values for instance variables *(you may be able to think of others)* :

1. Allow them to take on their default values.
2. Establish their values using initialization expressions.
3. Establish their values using hard-coded values within a constructor.
4. Establish their values using parameter values passed to parameterized constructors.

Using the first two ways

The following fragment illustrates the first two of those four ways.

```
  class Worker{
private int myInt = 100;
private double myDouble = 222.0;
private boolean myBoolean;

//...
```

In the above fragment, the instance variables named **myInt** and **myDouble** receive their initial values from initialization expressions. In these two cases, the initialization expressions are very simple. They are simply literal expressions. However, they could be much more complex if needed.

The variable named **myBoolean** in the above fragment is allowed to take on its default value of false.

Replacing the default *noarg* constructor

The next fragment shows one of the two overloaded constructors in the class named **Worker**. This constructor is a replacement for the default *noarg* constructor.

```
  class Worker{
private int myInt = 100;
private double myDouble = 222.0;
private boolean myBoolean;

public Worker(){
  myInt = 20;
}//end noarg constructor

//...
```

Using hard-coded values for initialization

This fragment illustrates the third of the four ways listed earlier to establish the initial value of the instance variables of an object of the class named **Worker**. In particular, this fragment assigns the hard-coded value 20 to the instance variable named **myInt**, thus overwriting the value of 100 previously established for that variable by an initialization expression.

*(All objects instantiated from the **Worker** class using this noarg constructor would have the same initial value for the variable named **myInt**.)*

Note, that this constructor does not disturb the initial values of the other two instance variables that were earlier established by an initialization expression, or by taking on the default value. Thus, the initial values of these two instance variables remain as they were immediately following the declaration of the variables.

Initial values using this *noarg* constructor

When an object of the **Worker** class is instantiated using this constructor and the values of the three instance variables are displayed, the results are as shown below:

20 222.0 false

The value of **myInt** is 20 as established by the constructor. The value of **myDouble** is 222.0 as established by the initialization expression, and the value of **myBoolean** is false as established by default.

Using constructor parameters for initialization

The next fragment shows the last of the four ways listed earlier for establishing the initial value of an instance variable.

```
  public class Ap119{
public static void main(
```

```
                        String args[]){
    //...
    new Worker(5,true).display();
    //...
  }//end main()
}//end class Ap119

class Worker{
  private int myInt = 100;
  private double myDouble = 222.0;
  private boolean myBoolean;
  //...

  public Worker(int x, boolean y){
    myInt = x;
    myBoolean = y;
  }//end parameterized constructor

  //...
```

A parameterized constructor

The above fragment shows the second of two overloaded constructors for the class named **Worker** . This constructor uses two incoming parameter values to establish the values of two of the instance variables, overwriting whatever values may earlier have been established for those variables.

The above fragment uses this constructor to instantiate an object of the **Worker** class, assigning incoming parameter values of 5 and *true* to the instance variables named **myInt** and **myBoolean** respectively. This overwrites the value previously placed in the variable named **myInt** by the initialization expression. It also overwrites the default value previously placed in the instance variable named **myBoolean** .

(Note that this constructor doesn't disturb the value for the instance variable named ***myDouble*** *that was previously established through the use of an initialization expression.)*

Initial values using parameterized constructor

After instantiating the new object, this fragment causes the values of all three instance variables to be displayed. The result is:

5 222.0 true

As you can see, the values contained in the instance variables named **myInt** and **myBoolean** are the values of 5 and true placed there by the constructor, based on incoming parameter values. The value in the instance variable named **myDouble** is the value placed there by the initialization expression when the variable was declared.

Default initialization

If you don't take any steps to initialize instance variables, they will be automatically initialized. Numeric instance variables will be initialized with zero value for the type of variable involved. Instance variables of type **boolean** will be initialized to false. Instance variables of type **char** will be initialized to a Unicode value with all 16 bits set to zero. Reference variables will be initialized to null.

Initialization expression

If you provide an initialization expression for an instance variable, the value of the expression will overwrite the default value, and the value of the initialization expression will become the initial value for the instance variable.

Assignment in constructor code

If you use an assignment statement in a constructor to assign a value to an instance variable, that value will overwrite the value previously placed in the instance variable either by default, or by use of an

initialization expression. The constructor has the "last word" on the matter of initialization of instance variables.

Back to Question 10 (p. 227)

11.6.2 Answer 9

A. Compiler Error

11.6.2.1 Explanation 9

The default constructor

When you define a class, you are not required to define a constructor for the class. If you do not define a constructor for the class, a default constructor that takes no arguments will be provided on your behalf. You can instantiate new objects of the class by applying the new operator to the default constructor as shown in the following code fragment from Question 8 (p. 225) .

```
new Worker().display();
```

Behavior of the default constructor

As illustrated in Question 8 (p. 225) , when you don't provide a constructor that purposely initializes the values of instance variables, or initialize them in some other manner, they will automatically be initialized to the default values described in Question 8 (p. 225) .

Defining overloaded constructors

You can also define one or more overloaded constructors having different formal argument lists. The typical intended purpose of such constructors is to use incoming parameter values to initialize the values of instance variables in the new object.

A parameterized constructor

This is illustrated in the following code fragment. This fragment receives two incoming parameters and uses the values of those two parameters to initialize the values of two instance variables belonging to the new object.

```
  class Worker{
private int myInt;
private boolean myBoolean;

public Worker(int x, boolean y){
  myInt = x;
  myBoolean = y;
}//end parameterized constructor

//display() omitted for brevity

}// end class Worker
```

If you define any constructors ...

However, there is a pitfall that you must never forget.

If you define any constructors in your new class, you must define all constructors that will ever be required for your new class.

If you define any constructors, the default constructor will no longer be provided automatically. Therefore, if a constructor that takes no arguments will ever be needed for your new class, and you define one or more parameterized constructors, you must define the *noarg* constructor when you define your class.

A parameterized constructor for Worker

The class named **Worker** in this program defines a constructor that receives two incoming parameters, one of type **int** and the other of type **boolean** . It uses those two incoming parameters to initialize two instance variables of the new object.

Oops!

However, it does not define a constructor with no arguments in the formal argument list *(commonly called a noarg constructor)* .

Calling the missing *noarg* constructor

The following code in the **main** method of the controlling class attempts to instantiate two objects of the **Worker** class. The first call of the constructor passes no parameters to the constructor. Thus, it requires a *noarg* constructor in order to instantiate the object.

```
  public class Ap118{
public static void main(
                      String args[]){
  new Worker().display();
  new Worker(5,true).display();
  System.out.println();
 }//end main()
}//end class Ap118
```

A compiler error

Since there is no constructor defined in the **Worker** class with an empty formal argument list *(and the default version is not provided)* , the program produces the following compiler error.

```
    Ap118.java:11: cannot resolve symbol
symbol  : constructor Worker
()
location: class Worker
    new Worker().display();
```

Back to Question 9 (p. 226)

11.6.3 Answer 8

C. 0 0.0 false

11.6.3.1 Explanation 8

All instance variables are initialized to default values

All instance variables are automatically initialized to default values if the author of the class doesn't take explicit steps to cause them to initialized to other values.

The default values

Numeric variables are automatically initialized to zero, while **boolean** variables are automatically initialized to false. Instance variables of type **char** are initialized to a Unicode value with all 16 bits set to zero. Reference variables are initialized to null.

Back to Question 8 (p. 225)

11.6.4 Answer 7

A. Compiler Error

11.6.4.1 Explanation 7

Pushing the compiler beyond its limits

Compared to many programming environments, the Java compiler is very forgiving. However, there is a limit to how far even the Java compiler is willing to go to keep us out of trouble.

Initializing the value of a static variable

We can initialize the value of a **static** variable using an initialization expression as follows:

```
public static final MY_CONSTANT
   = initialization expression;
```

Important point

It is necessary for the compiler to be able to evaluate the initialization expression when it is encountered.

Illegal forward reference

This program attempts to use an initialization expression that makes use of the value of another **static** variable *(myPI)* that has not yet been established at that point in the compilation process. As a result, the program produces the following compiler error under JDK 1.3.

```
Ap116.java:18: illegal forward reference
 = 2 * myPI;
       ^
```

Reverse the order of the variable declarations

The problem can be resolved by reversing the order of the two **static** variable declarations in the following revised version of the program.

```
  public class Ap116{
public static void main(
                      String args[]){
  System.out.println(Worker.twoPI);
}//end main()
}//end class Ap116

class Worker{
  public static final double myPI
                             = Math.PI;
  public static final double twoPI
                            = 2 * myPI;

}// end class Worker
```

This revised version of the program compiles and executes successfully.

Back to Question 7 (p. 224)

11.6.5 Answer 6

C. 6.283185307179586

11.6.5.1 Explanation 6

Access via an object

Question 5 (p. 223) illustrated the fact that a **public static final** member variable of a class can be accessed via a reference to an object instantiated from the class.

Not the only way to access a *static* variable

However, that is not the only way in which **static** member variables can be accessed. More importantly, **public static** member variables of a class can be accessed simply by referring to the name of the class and the name of the member variable joined by a period.

(Depending on other factors, it may not be necessary for the ***static*** *variable to also be declared* ***public*** *, but that is the most general approach.)*

A *public static final* member variable

In this program, the **Worker** class declares and initializes a **public static final** member variable named **twoPI** as shown in the following fragment.

```
  class Worker{
public static final double twoPI
                        = 2 * Math.PI;
}// end class Worker
```

Accessing the *static* variable

The single statement in the **main** method of the controlling class accesses and displays the value of the **public static final** member variable named **twoPI** as shown in the following fragment.

```
  public class Ap115{
public static void main(
                      String args[]){
  System.out.println(Worker.twoPI);
}//end main()
}//end class Ap115
```

Objects share one copy of *static* variables

Basically, when a member variable is declared **static** , no matter how many objects are instantiated from a class *(including no objects at all)* , they all share a single copy of the variable.

Sharing can be dangerous

This sharing of a common variable leads to the same kind of problems that have plagued programs that use ***global*** variables for years. If the code in any object changes the value of the **static** variable, it is changed insofar as all objects are concerned.

Should you use non-final *static* variables?

Most authors will probably agree that in most cases, you probably should not use **static** variables unless you also make them **final** .

(There are some cases, such as counting the number of objects instantiated from a class, where a non-final ***static*** *variable may be appropriate. However, the appropriate uses of non-final* ***static*** *variables are few and far between.)*

Should you also make *static* variables *public* ?

If you make your variables **static** and **final** , you will often also want to make them **public** so that they are easy to access. There are numerous examples in the standard Java class libraries where variables are declared as **public** , **static** , and **final** . This is the mechanism by which the class libraries create constants and make them available for easy access on a widespread basis.

The Color class

For example, the **Color** class defines a number of **public static final** variables containing the information that represents generic colors such as ORANGE, PINK, and MAGENTA. *(By convention, constants in Java are written with all upper-case characters, but that is not a technical requirement.)*

If you need generic colors and not custom colors, you can easily access and use these color values without the requirement to mix red, green, and blue to produce the desired color values.

Back to Question 6 (p. 224)

11.6.6 Answer 5

C. 6.283185307179586

11.6.6.1 Explanation 5

A *public static final* variable

This program declares a **public static final** member variable named **twoPI** in the class named **Worker** , and properly initializes it when it is declared as shown in the following code fragment.

```
class Worker{  public static final double twoPI
                        = 2 * Math.PI;
}/}// end class Worker
```

From that point forward in the program, this member variable named **twoPI** behaves like a constant, meaning that any code that attempts to change its value will cause a compiler error *(as in the program in Question 4 (p. 222))* ..

Accessing the static variable

The following single statement that appears in the **main** method of the controlling class instantiates a new object of the **Worker** class, accesses, and displays the **public static final** member variable named **twoPI** .

```
 public static void main(
                    String args[]){
System.out.println(
                new Worker().twoPI);
}  }//end main()
```

*(Note for future discussion that the variable named **twoPI** is accessed via a reference to an object instantiated from the class named **Worker** .)*

This causes the **double** value 6.283185307179586 to be displayed on the standard output device.

Back to Question 5 (p. 223)

11.6.7 Answer 4

A. Compiler Error

11.6.7.1 Explanation 4

A **final** variable

When a member variable of a class *(not a local variable)* is declared **final** , its value must be established when the variable is declared. This program attempts to assign a value to a **final** member variable after it has been declared, producing the following compiler error under JDK 1.3.

```
    Ap113.java:20: cannot assign a value to
final variable twoPI
    twoPI = 2 * Math.PI;
```

Back to Question 4 (p. 222)

11.6.8 Answer 3

D. 22

11.6.8.1 Explanation 3

Two uses of the *this* keyword

This program illustrates two different uses of the **this** keyword.

Disambiguating a reference to a variable

Consider first the use of **this** that is shown in the following code fragment.

```
  class Worker{
private int data;

public Worker(int data){
  this.data = data;
}//end constructor
```

Very common usage

The code in the above fragment is commonly used by many Java programmers. All aspiring Java programmers need to know how to read such code, even if they elect not to use it. In addition, understanding this code should enhance your overall understanding of the use and nature of the **this** keyword.

A parameterized constructor

The above fragment shows a parameterized constructor for the class named **Worker** . This constructor illustrates a situation where there is a local parameter named **data** that has the same name as an instance variable belonging to the object.

Casting a shadow

The existence of the local parameter named **data** casts a shadow on the instance variable having the same name, making it inaccessible by using its name alone.

(A local variable having the same name as an instance variable casts a similar shadow on the instance variable.)

In this shadowing circumstance, when the code in the constructor refers simply to the name **data** , it is referring to the local parameter having that name. In order for the code in the constructor to refer to the instance variable having the name data, it must refer to it as **this.data** .

In other words ...

In other words, **this.data** is a reference to an instance variable named **data** belonging to the object being constructed by the constructor *(this object)* .

Not always necessary

You could always use this syntax to refer to an instance variable of the object being constructed if you wanted to. However, the use of this syntax is necessary only when a local parameter or variable has the same name as the instance variable and casts a shadow on the instance variable. When this is not the case, you can refer to the instance variable simply by referring to its name without the keyword **this** .

Finally, the main point ...

Now consider the main point of this program. The following fragment shows the **main** method of the controlling class for the application.

```
  public class Ap112{
public static void main(
                      String args[]){
  Worker obj1 = new Worker(11);
  Worker obj2 = new Worker(22);
  Worker obj3 = new Worker(33);
  Worker obj4 = new Worker(44);
  obj2.doThis();
 }//end main()
}//end class Ap112
```

Four different objects of type Worker

The code in the above fragment instantiates four different objects from the class named **Worker** , passing a different value to the constructor for each object. Thus, individual instance variable in each of the four objects contain the **int** values 11, 22, 33, and 44 respectively.

Call an instance method on one object

Then the code in the **main** method calls the instance method named **doThis** on only one of the objects, which is the one referred to by the reference variable named **obj2** .

An overridden **toString** method of the **Worker** class is eventually called to return a **String** representation of the value stored in the instance variable named **data** for the purpose of displaying that value on the standard output device.

Overridden toString method

The next fragment shows the overridden **toString** method for the **Worker** class. As you can see, this overridden method constructs and returns a reference to a **String** representation of the **int** value stored in the instance variable named **data** . Thus, depending on which object the **toString** method is called on, different string values will be returned by the overridden method.

```
 public String toString(){
 return "" + data;
 }//end toString()
}// end class Worker
```

Passing reference to this object to println method

The next fragment shows the **doThis** instance method belonging to each object instantiated from the **Worker** class. When this method is called on a specific object instantiated from the **Worker** class, it uses the **this** keyword to pass that specific object's reference to the **println** method. The **println** method uses that reference to call the **toString** method on that specific object. This, in turn causes a **String** representation of the value of the instance variable named **data** belonging to that specific object to be displayed.

```
    public void doThis(){
  System.out.println(this);
}//end area()
```

The bottom line

In this program, the instance variable in the object referred to by **obj2** contains the value 22. The instance variables in the other three objects instantiated from the same class contain different values.

The bottom line is that the following statement in the **main** method causes the value 22 to be displayed on the standard output device. Along the way, the **this** keyword is used to cause the **println** method to get and display the value stored in a specific object, and to ignore three other objects that were instantiated from the same class.

```
obj2.doThis();
```

Back to Question 3 (p. 221)

11.6.9 Answer 2

C. 33

11.6.9.1 Explanation 2

The **this** keyword

The key to an understanding of this program lies in an understanding of the single statement that appears in the method named **doThis** , as shown in the following fragment.

```
  public void doThis(){
  new Helper().helpMe(this);
}//end area()
```

The keyword named **this** has several uses in Java, some of which are explicit, and some of which take place behind the scenes.

What do you need to know about the this keyword?

One of the uses of the keyword **this** is passing the implicit parameter in its entirety to another method. That is exactly what this program does. But what is the implicit parameter named **this** anyway?

Every object holds a reference to itself

This implicit reference can be accessed using the keyword **this** in a non-static *(instance)* method belonging to the object. *(The implicit reference named **this** cannot be accessed from within a **static** method for reasons that won't be discussed here.)*

Calling an instance method

An instance method can only be called by referring to a specific object and joining that object's reference to the name of the instance method using a period as the joining operator. This is illustrated in the following statement, which calls the method named **doThis** on a reference to an object of the class named **Worker** .

```
new Worker().doThis();
```

An anonymous object

The above statement creates an anonymous object of the class named **Worker** . *(An anonymous object is an object whose reference is not assigned to a named reference variable.)*

The code to the left of the period returns a reference to the new object. Then the code calls the instance method named **doThis** on the reference to the object.

Which object is *this* object?

When the code in the instance method named **doThis** refers to the keyword **this** , it is a reference to the specific object on which the **doThis** method was called. The statement in the following fragment passes a reference to that specific instance of the **Worker** class to a method named **helpMe** in a new object of the **Helper** class.

```
 public void doThis(){
new Helper().helpMe(this);
}//end area()
```

A little help here please

The **helpMe** method is shown in the following fragment.

```
  class Helper{
public void helpMe(Worker param){
  System.out.println(
                    param.getData());
}//end helpMe()
}//end class Helper
```

Using the incoming reference

The code in the **helpMe** method uses the incoming reference to the object of the **Worker** class to call the **getData** method on that object.

Thus code in the **helpMe** method is able to call a method in the object containing the method that called the **helpMe** method in the first place.

A callback scenario

When a method in one object calls a method in another object, passing **this** as a parameter, that makes it possible for the method receiving the parameter to make a callback to the object containing the method that passed **this** as a parameter.

The **getData** method returns a **String** representation of the **int** instance variable named **data** with a value of 33 that is contained in the object of the **Worker** class.

Display the value

The code in the **helpMe** method causes that string to be displayed on the computer screen.

And the main point is ...

Any number of objects can be instantiated from a given class. A given instance method can be called on any of those objects. When the code in such an instance method refers to **this** , it is referring to the specific object on which it was called, and is not referring to any of the many other objects that may have been instantiated from the same class.

Back to Question 2 (p. 220)

11.6.10 Answer 1

A. Compiler Error

11.6.10.1 Explanation 1

A wakeup call

The purpose of this question is simply to give you a wakeup call. The declaration for the method named **getData** indicates that the method returns a reference to an object of the class **String** . However, the code in the method attempts to return an **int** . The program produces the following compiler error under JDK 1.3.

```
        found   : int
    required: java.lang.String
        return data;
```

Back to Question 1 (p. 219)

-end-

Chapter 12

Ap0110: Self-assessment, Extending classes, overriding methods, and polymorphic behavior[1]

12.1 Table of Contents

12.2 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. 253) to easily find and view the listings while you are reading about them.

12.3 Questions

12.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 244) ?

- A. Compiler Error
- B. Runtime Error
- C. A

[1] This content is available online at <http://cnx.org/content/m45308/1.5/>.

- D. None of the above.

Listing 1 . Listing for Question 1.

```
    public class Ap120{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap120

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.1

Answer and Explanation (p. 263)

12.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 245) ?

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

Listing 2 . Listing for Question 2.

continued on next page

```
    public class Ap121{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap121

class Worker{
  void doIt(){
    Base myVar = new A();
    ((A)myVar).test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.2

Answer and Explanation (p. 262)

12.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 246) ?

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

Listing 3 . Listing for Question 3.

```
    public class Ap122{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap122

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  abstract public void test();
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.3

Answer and Explanation (p. 261)

12.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 247) ?

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

Listing 4 . Listing for Question 4.

```
    public class Ap123{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap123

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

abstract class Base{
  abstract public void test();
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.4

Answer and Explanation (p. 260)

12.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 248) ?

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

Listing 5 . Listing for Question 5.

continued on next page

```
    public class Ap124{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap124

class Worker{
  void doIt(){
    Base myVar = new Base();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

abstract class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.5

Answer and Explanation (p. 260)

12.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 249) ?

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

Listing 6 . Listing for Question 6.

```
    public class Ap125{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap125

class Worker{
  void doIt(){
    Base myVar = new Base();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.6

Answer and Explanation (p. 259)

12.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 250) ?

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

Listing 7 . Listing for Question 7.

continued on next page

```
    public class Ap126{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap126

class Worker{
  void doIt(){
    Base myVar = new Base();
    ((A)myVar).test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.7

Answer and Explanation (p. 257)

12.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 251) ?

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

Listing 8 . Listing for Question 8.

```
    public class Ap127{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap127

class Worker{
  void doIt(){
    Base myVar = new A();
    ((A)myVar).test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.8

Answer and Explanation (p. 256)

12.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 252) ?

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

Listing 9 . Listing for Question 9.

continued on next page

```
    public class Ap128{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap128

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Table 12.9

Answer and Explanation (p. 256)

12.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 253) ?

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

Listing 10 . Listing for Question 10.

```
    public class Ap129{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap129

class Worker{
  void doIt(){
    Base myVar = new A();
    myVar.test();
    myVar = new B();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B
```

Table 12.10

Answer and Explanation (p. 254)

12.4 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 5 (p. 248) . Listing for Question 5.
- Listing 6 (p. 249) . Listing for Question 6.
- Listing 7 (p. 250) . Listing for Question 7.
- Listing 8 (p. 251) . Listing for Question 8.
- Listing 9 (p. 252) . Listing for Question 9.
- Listing 10 (p. 253) . Listing for Question 10.

12.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0110: Self-assessment, Extending classes, overriding methods, and polymorphic behavior
- File: Ap0110.htm
- Originally published: 2002
- Published at cnx.org: 12/08/12
- Revised: 12/03/14

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

12.6 Answers

12.6.1 Answer 10

D. A B

12.6.1.1 Explanation 10

Another illustration of simple polymorphic behavior

In this program, two classes named **A** and **B** extend the class named **Base** , each overriding the method named **test** to produce different behavior. *(Typically, overridden methods in different classes will produce different behavior, even though they have the same names.)*

Behavior appropriate for object on which method is called

In other words, the behavior of the method named **test** , when called on a reference to an object of type **A** , is different from the behavior of the method named **test** when called on a reference to an object of type **B** .

The method definitions

The definitions of the two classes named **A** and **B**, along with the two versions of the overridden method named **test** are shown in the following fragment.

```
    class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B
```

Store a subclass object's reference as a superclass type

The program declares a reference variable of type **Base**, instantiates a new object of the class named **A**, and assigns that object's reference to the reference variable of type **Base**. Then it calls the method named **test** on that reference as shown in the following fragment.

```
      Base myVar = new A();
  myVar.test();
```

Polymorphic behavior applies

Simple polymorphic behavior causes the overridden version of the method named **test**, defined in the class named **A**, *(as opposed to the versions defined in class **Base** or class **B**)* to be executed. This causes the letter **A** followed by a space character to be displayed on the standard output device.

Store another subclass object's reference as superclass type

Then the program instantiates a new object from the class named **B**, and assigns that object's reference to the same reference variable, overwriting the reference previously stored there. *(This causes the object whose reference was previously stored in the reference variable to become eligible for garbage collection in this case.)*

Then the program calls the method named **test** on the reference as shown in the following fragment.

```
      myVar = new B();
  myVar.test();
```

Polymorphic behavior applies again

This time, simple polymorphic behavior causes the overridden version of the method named **test**, defined in the class named **B**, *(as opposed to the versions defined in class **Base** or class **A**)* to be executed. This causes the letter **B** followed by a space character to be displayed on the standard output device.

Once again, what is runtime polymorphic behavior?

With runtime polymorphic behavior, the method selected for execution is based, not on the type of the reference variable holding the reference to the object, but rather on the actual class from which the object was instantiated.

If the method was properly overridden, the behavior exhibited by the execution of the method is appropriate for an object of the class from which the object was instantiated.

Back to Question 10 (p. 252)

12.6.2 Answer 9

D. A

12.6.2.1 Explanation 9

Compiles and executes successfully

This program compiles and executes successfully causing the version of the method named **test** , which is overridden in the class named **A** to be executed. That overridden method is shown in the following fragment.

```
  class A extends Base{
public void test(){
  System.out.print("A ");
}//end test()
}//end class A
```

So, what is the issue here?

The purpose of this program is to determine if you understand polymorphic behavior and the role of downcasting. Consider the following fragment taken from the program in Question 8 (p. 250) .

```
    Base myVar = new A();
((A)myVar).test();
```

The downcast is redundant

As you learned in the discussion of Question 8 (p. 250) , the downcast isn't required, and it has no impact on the behavior of the program in Question 8 (p. 250) .

This program behaves exactly the same with the second statement in the above fragment replaced by the following statement, which does not contain a downcast.

```
myVar.test();
```

Again, you need to know when downcasting is required, when it isn't required, and to make use of that knowledge to downcast appropriately.

Back to Question 9 (p. 251)

12.6.3 Answer 8

D. A

12.6.3.1 Explanation 8

Compiles and executes successfully

This program compiles and executes successfully causing the version of the method named **test** , which is overridden in the class named **A** to be executed. That overridden method is shown in the following fragment.

```
  class A extends Base{
public void test(){
  System.out.print("A ");
}//end test()
}//end class A
```

So, what is the issue here?

The purpose of this program is to determine if you understand polymorphic behavior and the role of downcasting, as shown in the following fragment.

```
    Base myVar = new A();
((A)myVar).test();
```

This would be a simple case of polymorphic behavior were it not for the downcast shown in the above fragment.

The downcast is redundant

Actually, the downcast was placed there to see if you could determine that it is redundant. It isn't required, and it has no impact on the behavior of this program. This program would behave exactly the same if the second statement in the above fragment were replaced with the following statement, which does not contain a downcast.

```
myVar.test();
```

You need to know when downcasting is required, when it isn't required, and to make use of that knowledge to downcast appropriately.

Back to Question 8 (p. 250)

12.6.4 Answer 7

B. Runtime Error

12.6.4.1 Explanation 7

Storing a reference as a superclass type

You can store an object's reference in any reference variable whose declared type is a superclass of the actual class from which the object was instantiated.

May need to downcast later

Later on, when you attempt to make use of that reference, you may need to downcast it. Whether or not you will need to downcast will depend on what you attempt to do.

In order to call a method ...

For example, if you attempt to call a method on the reference, but that method is not defined in or inherited into the class of the reference variable, then you will need to downcast the reference in order to call the method on that reference.

Class Base defines method named test

This program defines a class named **Base** that defines a method named **test** .

Class A extends Base and overrides test

The program also defines a class named **A** that extends **Base** and overrides the method named **test** as shown in the following fragment.

```
    class Base{
  public void test(){
    System.out.print("Base ");};
}//end class Base

class A extends Base{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

A new object of the class Base

The program instantiates a new object of the class **Base** and stores a reference to that object in a reference variable of type **Base**, as shown in the following fragment.

```
        Base myVar = new Base();
    ((A)myVar).test();
```

Could call test directly on the reference

Having done this, the program could call the method named **test** directly on the reference variable using a statement such as the following, which is not part of this program.

```
        myVar.test();
```

This statement would cause the version of the method named **test** defined in the class named **Base** to be called, causing the word **Base** to appear on the standard output device.

This downcast is not allowed

However, this program attempts to cause the version of the method named **test** defined in the class named **A** to be called, by downcasting the reference to type **A** before calling the method named **test**. This is shown in the following fragment.

```
        ((A)myVar).test();
```

A runtime error occurs

This program compiles successfully. However, the downcast shown above causes the following runtime error to occur under JDK 1.3:

```
Exception in thread "main" java.lang.ClassCastException: Base
    at Worker.doIt(Ap126.java:22)
    at Ap126.main(Ap126.java:15)
```

What you can do

You can store an object's reference in a reference variable whose type is a superclass of the class from which the object was originally instantiated. Later, you can downcast the reference back to the type *(class)* from which the object was instantiated.

What you cannot do

However, you cannot downcast an object's reference to a subclass of the class from which the object was originally instantiated.

Unfortunately, the compiler is unable to detect an error of this type. The error doesn't become apparent until the exception is thrown at runtime.

Back to Question 7 (p. 249)

12.6.5 Answer 6

C. Base

12.6.5.1 Explanation 6

Totally straightforward code

This rather straightforward program instantiates an object of the class named **Base** and assigns that object's reference to a reference variable of the type **Base** as shown in the **following fragment** .

```
    Base myVar = new Base();
myVar.test();
```

Then it calls the method named **test** on the reference variable.

Class Base defines the method named test

The class named **Base** contains a concrete definition of the method named **test** as shown in the following fragment. This is the method that is called by the code shown in the above fragment (p. 259) .

```
  class Base{
public void test(){
  System.out.print("Base ");};
}//end class Base
```

Class A is just a smokescreen

The fact that the class named **A** extends the class named **Base** , and overrides the method named **test** , as shown in the following fragment, is of absolutely no consequence in the behavior of this program. Hopefully you understand why this is so. If not, then you still have a great deal of studying to do on Java inheritance.

```
  class A extends Base{
public void test(){
  System.out.print("A ");
  }//end test()
}//end class A
```

Back to Question 6 (p. 248)

12.6.6 Answer 5

A. Compiler Error

12.6.6.1 Explanation 5

Cannot instantiate an abstract class

This program defines an **abstract** class named **Base** . Then it violates one of the rules regarding **abstract** classes, by attempting to instantiate an object of the **abstract** class as shown in the following code fragment.

```
Base myVar = new Base();
```

The program produces the following compiler error under JDK 1.3:

```
Ap124.java:19: Base is abstract; cannot be instantiated
Base myVar = new Base();
```

Back to Question 5 (p. 247)

12.6.7 Answer 4

C. A

12.6.7.1 Explanation 4

An abstract class with an abstract method

This program illustrates the use of an **abstract** class containing an **abstract** method to achieve *polymorphic behavior* .

The following code fragment shows an **abstract** class named **Base** that contains an **abstract** method named **test** .

```
  abstract class Base{
abstract public void test();
}//end class Base
```

Extending abstract class and overriding abstract method

The class named **A** , shown in the following fragment extends the **abstract** class named **Base** and overrides the **abstract** method named **test** .

```
  class A extends Base{
public void test(){
  System.out.print("A ");
}//end test()
}//end class A
```

Can store a subclass reference as a superclass type

Because the class named **A** extends the class named **Base**, a reference to an object instantiated from the class named **A** can be stored in a reference variable of the declared type **Base**. No cast is required in this case.

Polymorphic behavior

Furthermore, because the class named **Base** contains the method named **test**, *(as an **abstract** method)*, when the method named **test** is called on a reference to an object of the class named **A**, stored in a reference variable of type **Base**, the *overridden* version of the method as defined in the class named **A** will actually be called. This is polymorphic behavior.

*(Note, however, that this example does little to illustrate the power of polymorphic behavior because only one class extends the class named **Base** and only one version of the abstract method named **test** exists. Thus, the system is not required to select among two or more overridden versions of the method named **test**.)*

The important code

The following code fragment shows the instantiation of an object of the class named **A** and the assignment of that object's reference to a reference variable of type **Base**. Then the fragment calls the method named **test** on the reference variable.

```
    Base myVar = new A();
myVar.test();
```

This causes the overridden version of the method named **test**, shown in the following fragment, to be called, which causes the letter **A** to be displayed on the standard output device.

```
  public void test(){
  System.out.print("A ");
}//end test()
```

Back to Question 4 (p. 246)

12.6.8 Answer 3

A. Compiler Error

12.6.8.1 Explanation 3

Classes can be final or abstract, but not both

A class in Java may be declared **final**. A class may also be declared **abstract**. A class cannot be declared both **final** and **abstract**.

Behavior of final and abstract classes

A class that is declared **final** cannot be extended. A class that is declared **abstract** cannot be instantiated. Therefore, it must be extended to be useful.

An **abstract** class is normally intended to be extended.

Methods can be final or abstract, but not both

A method in Java may be declared **final**. A method may also be declared **abstract**. However, a method cannot be declared both **final** and **abstract**.

Behavior of final and abstract methods

A method that is declared **final** cannot be overridden. A method that is declared **abstract** must be overridden to be useful.

An **abstract** method doesn't have a body.

Abstract classes and methods

A class that contains an **abstract** method must itself be declared **abstract** . However, an **abstract** class is not required to contain **abstract** methods.

Failed to declare the class abstract

In this program, the class named **Base** contains an **abstract** method named **test** , but the class is not declared **abstract** as required.

```
    class Base{
  abstract public void test();
}//end class Base
```

Therefore, the program produces the following compiler error under JDK 1.3:

```
    Ap122.java:24: Base should be declared abstract;
it does not define test in Base
class Base{
```

Back to Question 3 (p. 245)

12.6.9 Answer 2

C. A

12.6.9.1 Explanation 2

If you missed this ...

If you missed this question, you didn't pay attention to the explanation for Question 1 (p. 243) .

Define a method in a subclass

This program defines a subclass named **A** that extends a superclass named **Base** . A method named **test** is defined in the subclass named **A** but is not defined in any superclass of the class named **A** .

Store a reference as a superclass type

The program declares a reference variable of the superclass type, and stores a reference to an object of the subclass in that reference variable as shown in the following code fragment.

```
Base myVar = new A();
```

Downcast and call the method

Then the program calls the method named **test** on the reference stored as the superclass type, as shown in the following fragment.

```
((A)myVar).test();
```

Unlike the program in Question 1 (p. 243) , the reference is downcast to the true type of the object before calling the method named **test** . As a result, this program does not produce a compiler error.

Why is the cast required?

As explained in Question 1 (p. 243) , it is allowable to store a reference to a subclass object in a variable of a superclass type. Also, as explained in Question 1 (p. 243) , it is not allowable to directly call, on that superclass reference, a method of the subclass object that is not defined in or inherited into the superclass.

However, such a call is allowable if the programmer purposely downcasts the reference to the true type of the object before calling the method.

Back to Question 2 (p. 244)

12.6.10 Answer 1

A. Compiler Error

12.6.10.1 Explanation 1

Define a method in a subclass

This program defines a subclass named **A** that extends a superclass named **Base** . A method named **test** , is defined in the subclass named **A** , which is not defined in any superclass of the class named **A** .

Store a reference as superclass type

The program declares a reference variable of the superclass type, and stores a reference to an object of the subclass in that reference variable as shown in the following code fragment.

```
Base myVar = new A();
```

Note that no cast is required to store a reference to a subclass object in a reference variable of a superclass type. The required type conversion happens automatically in this case.

Call a method on the reference

Then the program attempts to call the method named **test** on the reference stored as the superclass type, as shown in the following fragment. This produces a compiler error.

```
myVar.test();
```

The reason for the error

It is allowable to store a reference to a subclass object in a variable of a superclass type. However, it is not allowable to directly call, *(on that superclass reference)* , a method of the subclass object that is not defined in or inherited into the superclass.

The following error message is produced by JDK 1.3.

```
    Ap120.java:18: cannot resolve symbol
symbol  : method test  ()
location: class Base
    myVar.test();
```

The solution is ...

This error can be avoided by casting the reference to type **A** before calling the method as shown below:

```
((A)myVar).test();
```

Back to Question 1 (p. 243)
-end-

Chapter 13

Ap0120: Self-assessment, Interfaces and polymorphic behavior[1]

13.1 Table of Contents

13.2 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. 280) to easily find and view the listings while you are reading about them.

13.3 Questions

13.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 266) ?

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

[1] This content is available online at <http://cnx.org/content/m45303/1.5/>.

Listing 1 . Listing for Question 1.

```
    public class Ap131{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap131

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    X myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Table 13.1

Answer and Explanation (p. 295)

13.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 267) ?

- A. Compiler Error

- B. Runtime Error
- C. A-inherMethod A-intfcMethod
- D. None of the above.

Listing 2 . Listing for Question 2.

```
    public class Ap132{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap132

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    Base myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Table 13.2

Answer and Explanation (p. 294)

13.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 268) ?

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

Listing 3 . Listing for Question 3.

```
    public class Ap133{
  public static void main(
                         String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap133

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    A myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Table 13.3

Answer and Explanation (p. 292)

13.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 270) ?

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

Listing 4 . Listing for Question 4.

```
    public class Ap134{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap134

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.inherMethod();
    X myVar2 = new A();
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Table 13.4

Answer and Explanation (p. 291)

13.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 271) ?

- A. Compiler Error

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

Listing 5 . Listing for Question 5.

```
    public class Ap135{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap135

class Worker{
  void doIt(){
    X myVar1 = new A();
    myVar1.intfcMethodX();
    X myVar2 = new B();
    myVar2.intfcMethodX();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "B-intfcMethodX ");
  }//end intfcMethodX()
```

```
}//end class B

interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 288)

13.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 272) ?

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

Listing 6 . Listing for Question 6.

```
    public class Ap136{
  public static void main(
                       String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap136

class Worker{
  void doIt(){
    Object[] myArray = new Object[2];
    myArray[0] = new A();
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      myArray[i].intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
```

```
    System.out.print(
                    "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B

interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 285)

13.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 273) ?

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

Listing 7 . Listing for Question 7.

```
    public class Ap137{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap137

class Worker{
  void doIt(){
    Object[] myArray = new Object[2];
    myArray[0] = new A();
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      ((X)myArray[i]).intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
```

```
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                    "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B

interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 284)

13.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 274) ?

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

Listing 8 . Listing for Question 8.

```
  public class Ap138{
public static void main(
                      String args[]){
  new Worker().doIt();
```

```
  }//end main()
}//end class Ap138

class Worker{
  void doIt(){
    X[] myArray = new X[2];
    myArray[0] = new A();
    myArray[1] = new B();

    for(int i=0;i<myArray.length;i++){
      myArray[i].intfcMethodX();
    }//end for loop

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "A-intfcMethodX ");
  }//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                   "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B

interface X{
  public void intfcMethodX();
}//end X
```

Answer and Explanation (p. 283)

13.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 277) ?

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

Listing 9 . Listing for Question 9.

```
    public class Ap139{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap139

class Worker{
  void doIt(){
    Base myVar = new Base();
    myVar.test();
    myVar = new A();
    myVar.test();
    myVar = new B();
    myVar.test();
    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");
  }//end test()
}//end class Base

class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base implements X,Y{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B

interface X{
  public void test();
}//end X

interface Y{
  public void test();
}//end Y
```

Table 13.5

Answer and Explanation (p. 282)

13.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 279) ?

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

Listing 10 . Listing for Question 10.

```
    public class Ap140{
  public static void main(
                        String args[]){
        new Worker().doIt();
  }//end main()
}//end class Ap140

class Worker{
  void doIt(){
    Base myVar1 = new Base();
    myVar1.test();
    myVar1 = new A();
    myVar1.test();
    myVar1 = new B();
    myVar1.test();

    X myVar2 = (X)myVar1;
    myVar2.test();

    System.out.println("");
  }//end doIt()
}// end class Worker

class Base{
  public void test(){
    System.out.print("Base ");
  }//end test()
}//end class Base

class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A

class B extends Base implements X,Y{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B

interface X{
  public void test();
}//end X

interface Y{
  public void test();
}//end Y
```

Table 13.6

Answer and Explanation (p. 280)

13.4 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.

13.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0120: Self-assessment, Interfaces and polymorphic behavior
- File: Ap0120.htm
- Originally published: 2004
- Published at cnx.org: 12/08/12
- Revised: 02/07/16

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.

13.6 Answers

13.6.1 Answer 10

C. Base A B B

13.6.1.1 Explanation 10

Expansion of the program from Question 9 (p. 276)

The class and interface definitions for the classes and interfaces named **Base** , **A** , **B** , **X** , and **Y** are the same as in Question 9 (p. 276) .

Call the test method differently

However, the call of the method named **test** in the object instantiated from the class named **B** is somewhat different. The difference is identified by the code in the following fragment.

```
  void doIt(){
Base myVar1 = new Base();
myVar1.test();
myVar1 = new A();
myVar1.test();
myVar1 = new B();
myVar1.test();

X myVar2 = (X)myVar1;
myVar2.test();

System.out.println("");
}//end doIt()
```

Calling test method on Base-type reference

In Question 9 (p. 276) , and in the above code fragment as well, the method named **test** was called on each of the objects using a reference stored in a reference variable of type **Base** .

Calling the overridden version of test method

This might be thought of as calling the overridden version of the method, through polymorphism, without regard for anything having to do with the interfaces.

Calling test method on interface-type reference

Then the code shown above calls the same method named **test** on one of the same objects using a reference variable of the interface type **X** .

Only one test method in each object

Keep in mind that each object defines only one method named **test** . This single method serves the dual purpose of overriding the method having the same signature from the superclass, and implementing a method with the same signature declared in each of the interfaces.

Implementing the interface method

Perhaps when the same method is called using a reference variable of the interface type, it might be thought of as implementing the interface method rather than overriding the method defined in the superclass. You can be the judge of that.

The same method is called regardless of reference type

In any event, in this program, the same method is called whether it is called using a reference variable of the superclass type, or using a reference variable of the interface type.

Illustrates the behavior of signature collisions

The purpose of this and Question 9 (p. 276) is not necessarily to illustrate a useful inheritance and implementation construct. Rather, these two questions are intended to illustrate the behavior of Java for the case of duplicated superclass and interface method signatures.

Back to Question 10 (p. 278)

13.6.2 Answer 9

C. Base A B

13.6.2.1 Explanation 9

A question regarding signature collisions

The question often arises in my classroom as to what will happen if a class inherits a method with a given signature and also implements one or more interfaces that declare a method with an identical signature.

The answer

The answer is that nothing bad happens, as long as the class provides a concrete definition for a method having that signature.

Only one method definition is allowed

Of course, only one definition can be provided for any given method signature, so that definition must satisfy the needs of overriding the inherited method as well as the needs of implementing the interfaces.

An example of signature collisions

The following fragment defines a class named **Base** that defines a method named **test** . The code also defines two interfaces named **X** and **Y** , each of which declares a method named **test** with an identical signature.

```
  class Base{
public void test(){
  System.out.print("Base ");
}//end test()
}//end class Base

interface X{
  public void test();
}//end X

interface Y{
  public void test();
}//end Y

class A extends Base implements X,Y{
  public void test(){
    System.out.print("A ");
  }//end test()
}//end class A
```

Classes A and B extend Base and implement X and Y

The code in the following fragment defines two classes, named **A** and **B** , each of which extends **Base** , and each of which implements both interfaces **X** and **Y** . Each class provides a concrete definition for the method named **test** , with each class providing a different definition.

```
  class A extends Base implements X,Y{
public void test(){
  System.out.print("A ");
}//end test()
```

```
}//end class A

class B extends Base implements X,Y{
  public void test(){
    System.out.print("B ");
  }//end test()
}//end class B
```

Override inherited method and define interface method

Each of the methods named **test** in the above fragment serves not only to override the method inherited from the class named **Base** , but also to satisfy the requirement to define the methods declared in the implemented interfaces named **X** and **Y** . *(This can also be thought of as overriding an inherited abstract method from an interface.)*

Store object's references as type Base and call test method

Finally, the code in the following fragment declares a reference variable of the type **Base** . Objects respectively of the classes **Base** , **A** , and **B** are instantiated and stored in the reference variable. Then the method named **test** is called on each of the references in turn.

```
    void doIt(){
  Base myVar = new Base();
  myVar.test();
  myVar = new A();
  myVar.test();
  myVar = new B();
  myVar.test();
  System.out.println("");
}//end doIt()
}// end class Worker
```

As you probably expected, this causes the following text to appear on the screen:

```
Base A B
```

Back to Question 9 (p. 276)

13.6.3 Answer 8

C. A-intfcMethodX B-intfcMethodX

13.6.3.1 Explanation 8

Similar to previous two programs

This program is very similar to the programs in Question 6 (p. 272) and Question 7 (p. 273) . The program is Question 6 (p. 272) exposed a specific type mismatch problem. The program in Question 7 (p. 273) provided one solution to the problem.

A different solution

The following fragment illustrates a different solution to the problem.

```
  void doIt(){
X[] myArray = new X[2];
myArray[0] = new A();
myArray[1] = new B();

for(int i=0;i<myArray.length;i++){
  myArray[i].intfcMethodX();
}//end for loop

System.out.println("");
}//end doIt()
```

An array object of the interface type

In this case, rather than to declare the array object to be of type **Object**, the array is declared to be of the interface type **X**.

This is a less generic container than the one declared to be of type **Object**. Only references to objects instantiated from classes that implement the **X** interface, or objects instantiated from subclasses of those classes can be stored in the container. However, this is often adequate.

What methods can be called?

Since the references are stored as the interface type, any method declared in or inherited into the interface can be called on the references stored in the container. Of course, the objects referred to by those references must provide concrete definitions of those methods or the program won't compile.

(Although it isn't implicitly obvious, it is also possible to call any of the eleven methods defined in the ***Object*** *class on an object's reference being stored as an interface type. Those eleven methods can be called on any object, including array objects, regardless of how the references are stored.)*

Not the standard approach

If you are defining your own container, this is a satisfactory approach to implementation of the observer design pattern. However, you cannot use this approach when using containers from the standard collections framework, because those containers are designed to always store references as the generic type **Object**. In those cases, the casting solution of Question 7 (p. 273) *(or the use of generics)* is required.

Back to Question 8 (p. 274)

13.6.4 Answer 7

C. A-intfcMethodX B-intfcMethodX

13.6.4.1 Explanation 7

The correct use of an interface

This program illustrates the correct use of an interface. It uses a cast of the interface type in the following fragment to resolve the problem that was discussed at length in Question 6 (p. 272) earlier.

```
  void doIt(){
Object[] myArray = new Object[2];
myArray[0] = new A();
myArray[1] = new B();

for(int i=0;i<myArray.length;i++){
  ((X)myArray[i]).intfcMethodX();
}//end for loop
```

```
    System.out.println("");
  }//end doIt()
```

Back to Question 7 (p. 273)

13.6.5 Answer 6

A. Compiler Error

13.6.5.1 Explanation 6

What is a container?

The word container is often used in Java, with at least two different meaning. One meaning is to refer to the type of an object that is instantiated from a subclass of the class named **Container** . In that case, the object can be considered to be of type **Container** , and typically appears in a graphical user interface *(GUI)* . That is not the usage of the word in the explanation of this program.

A more generic meaning

In this explanation, the word container has a more generic meaning. It is common to store a collection of object references in some sort of Java container, such as an array object or a **Vector** object. In fact, there is a complete collections framework provided to facilitate that sort of thing *(**Vector** is one of the concrete classes in the Java Collections Framework)* .

Storing references as type Object

It is also common to declare the type of references stored in the container to be of the class **Object** . Because **Object** is a completely generic type, this means that a reference to any object instantiated from any class *(or any array object)* can be stored in the container. The standard containers such as **Vector** and **Hashtable** take this approach.

(Note that this topic became a little more complicated with the release of generics in jdk version 1.5.)

A class named Base and an interface named X

In a manner similar to several previous programs, this program defines a class named **Base** and an interface named **X** as shown in the following fragment.

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

interface X{
  public void intfcMethodX();
}//end X
```

Classes A and B extend Base and implement X

Also similar to previous programs, this program defines two classes named **A** and **B** . Each of these classes extends the class named **Base** and implements the interface named **X** , as shown in the next fragment.

```
  class A extends Base implements X{
public void inherMethod(){
  System.out.print(
                  " A-inherMethod ");
}//end inherMethod()

public void intfcMethodX(){
  System.out.print(
                  "A-intfcMethodX ");
}//end intfcMethodX()
}//end class A

class B extends Base implements X{
  public void inherMethod(){
    System.out.print(
                  " B-inherMethod ");
  }//end inherMethod()

  public void intfcMethodX(){
    System.out.print(
                  "B-intfcMethodX ");
  }//end intfcMethodX()
}//end class B
```

Concrete definitions of the interface method

As before, these methods provide concrete definitions of the method named **intfcMethodX**, which is declared in the interface named **X**.

An array of references of type *Object*

The interesting portion of this program begins in the following fragment, which instantiates and populates a two-element array object *(container)* of type **Object**. *(In the sense of this discussion, an array object is a container, albeit a very simple one.)*

```
  void doIt(){
Object[] myArray = new Object[2];
myArray[0] = new A();
myArray[1] = new B();
```

Store object references of type *A* and B as type *Object*

Because the container is declared to be of type **Object**, references to objects instantiated from any class can be stored in the container. The code in the above fragment instantiates two objects, *(one of class **A** and the other of class **B**)*, and stores the two object's references in the container.

Cannot call interface method as type Object

The code in the **for** loop in the next fragment attempts to call the method named **intfcMethodX** on each of the two objects whose references are stored in the elements of the array.

```
  for(int i=0;i<myArray.length;i++){
myArray[i].intfcMethodX();
}//end for loop
```

```
      System.out.println("");
    }//end doIt()
```

This produces the following compiler error under JDK 1.3:

```
    Ap136.java:24: cannot resolve symbol
symbol  : method intfcMethodX  ()
location: class java.lang.Object

myArray[i].intfcMethodX();
```

What methods can you call as type Object?

It is allowable to store the reference to an object instantiated from any class in a container of the type **Object** . However, the only methods that can be directly called *(without a cast and not using generics)* on that reference are the following eleven methods. These methods are defined in the class named **Object** :

- **clone** ()
- equals(Object obj)
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- wait()
- wait(long timeout)
- wait(long timeout,int nanos)

Overridden methods

Some, *(but not all)* , of the methods in the above list are defined with default behavior in the **Object** class, and are meant to be overridden in new classes that you define. This includes the methods named **equals** and **toString** .

Some of the methods in the above list, such as **getClass** , are simply utility methods, which are not meant to be overridden.

Polymorphic behavior applies

If you call one of these methods on an object's reference *(being stored as type Object)* , polymorphic behavior will apply. The version of the method overridden in, or inherited into, the class from which the object was instantiated will be identified and executed.

Otherwise, a cast is required

In order to call any method other than one of the eleven methods in the above list (p. 287) , *(on an object's reference being stored as type* ***Object*** *without using generics)* , you must cast the reference to some other type.

Casting to an interface type

The exact manner in which you write the cast will differ from one situation to the next. In this case, the problem can be resolved by rewriting the program using the interface cast shown in the following fragment.

```
    void doIt(){
  Object[] myArray = new Object[2];
  myArray[0] = new A();
  myArray[1] = new B();

  for(int i=0;i<myArray.length;i++){
    ((X)myArray[i]).intfcMethodX();
  }//end for loop

  System.out.println("");
}//end doIt()
```

The observer design pattern

By implementing an interface, and using a cast such as this, you can store references to many different objects, of many different actual types, each of which implements the same interface, but which have no required superclass-subclass relationship, in the same container. Then, when needed, you can call the interface methods on any of the objects whose references are stored in the container.

This is a commonly used design pattern in Java, often referred to as the observer design pattern.

Registration of observers

With this design pattern, none, one, or more observer objects, *(which implement a common observer interface)* are registered on an observable object. This means references to the observer objects are stored in a container by the observable object.

Making a callback

When the observable object determines that some interesting event has occurred, the observable object calls a specific interface method on each of the observer objects whose references are stored in the container.

The observer objects execute whatever behavior they were designed to execute as a result of having been notified of the event.

The model-view-control (MVC) paradigm

In fact, there is a class named **Observable** and an interface named **Observer** in the standard Java library. The purpose of these class and interface definitions is to make it easy to implement the observer design pattern.

(The Observer interface and the Observable class are often used to implement a programming style commonly referred to as the MVC paradigm.)

Delegation event model, bound properties of Beans, etc.

Java also provides other tools for implementing the observer design pattern under more specific circumstances, such as the Delegation Event Model, and in conjunction with bound and constrained properties in JavaBeans Components.

Back to Question 6 (p. 272)

13.6.6 Answer 5

C. A-intfcMethodX B-intfcMethodX

13.6.6.1 Explanation 5

More substantive use of an interface

This program illustrates a more substantive use of the interface than was the case in the previous programs.

The class named Base

The program defines a class named **Base** as shown in the following fragment.

```
  class Base{
public void inherMethod(){
  System.out.print("Base ");
  }//end inherMethod()
}//end class Base
```

The interface named X

The program also defines an interface named **X** as shown in the next fragment. Note that this interface declares a method named **intfcMethodX** .

```
  interface X{
public void intfcMethodX();
}//end X
```

Class A extends Base and implements X

The next fragment shows the definition of a class named **A** that extends **Base** and implements **X** .

```
  class A extends Base implements X{
public void inherMethod(){
  System.out.print(
                  " A-inherMethod ");
}//end inherMethod()

public void intfcMethodX(){
  System.out.print(
                  "A-intfcMethodX ");
}//end intfcMethodX()
}//end class A
```

Defining interface method

Because the class named **A** implements the interface named **X** , it must provide a concrete definition of all the methods declared in **X** .

In this case, there is only one such method. That method is named **intfcMethodX** . A concrete definition for the method is provided in the class named **A** .

Class B also extends Base and implements X

The next fragment shows the definition of another class *(named **B**)*, which also extends **Base** and implements **X** .

```
  class B extends Base implements X{
public void inherMethod(){
  System.out.print(
                  " B-inherMethod ");
}//end inherMethod()

public void intfcMethodX(){
  System.out.print(
                  "B-intfcMethodX ");
}//end intfcMethodX()
}//end class B
```

Defining the interface method

Because this class also implements **X**, it must also provide a concrete definition of the method named **intfcMethodX**.

Different behavior for interface method

However *(and this is extremely important)*, there is no requirement for this definition of the method to match the definition in the class named **A**, or to match the definition in any other class that implements **X**.

Only the method signature for the method named **intfcMethodX** is necessarily common among all the classes that implement the interface.

The definition of the method named **intfcMethodX** in the class named **A** is different from the definition of the method having the same name in the class named **B**.

The interesting behavior

The interesting behavior of this program is illustrated by the code in the following fragment.

```
  void doIt(){
X myVar1 = new A();
myVar1.intfcMethodX();
X myVar2 = new B();
myVar2.intfcMethodX();

System.out.println("");
}//end doIt()
```

Store object's references as interface type X

The code in the above fragment causes one object to be instantiated from the class named **A**, and another object to be instantiated from the class named **B**.

The two object's references are stored in two different reference variables, each declared to be of the type of the interface **X**.

Call the interface method on each reference

A method named **intfcMethodX** is called on each of the reference variables. Despite the fact that both object's references are stored as type **X**, the system selects and calls the appropriate method, *(as defined by the class from which each object was instantiated)*, on each of the objects. This causes the following text to appear on the screen:

```
A-intfcMethodX B-intfcMethodX
```

No subclass-superclass relationship exists

Thus, the use of an interface makes it possible to call methods having the same signatures on objects instantiated from different classes, without any requirement for a subclass-superclass relationship to exist among the classes involved.

In this case, the only subclass-superclass relationship between the classes named **A** and **B** was that they were both subclasses of the same superclass. Even that relationship was established for convenience, and was not a requirement.

Different behavior of interface methods

The methods having the same signature, *(declared in the common interface, and defined in the classes)*, need not have any similarity in terms of behavior.

A new interface relationship

The fact that both classes implemented the interface named **X** created a new relationship among the classes, which is not based on class inheritance.

Back to Question 5 (p. 270)

13.6.7 Answer 4

C. Base A-intfcMethod

13.6.7.1 Explanation 4

Illustrates the use of an interface as a type

The program defines a class named **Base** , and a class named **A** , which extends **Base** , and implements an interface named **X** , as shown below.

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Implementing interfaces

A class may implement none, one, or more interfaces.

The cardinal rule on interfaces

If a class implements one or more interfaces, that class must either be declared abstract, or it must provide concrete definitions of all methods declared in and inherited into all of the interfaces that it implements. If the class is declared abstract, its subclasses must provide concrete definitions of the interface methods.

A concrete definition of an interface method

The interface named **X** in this program declares a method named **intfcMethod** . The class named **A** provides a concrete definition of that method.

(The minimum requirement for a concrete definition is a method that matches the method signature and has an empty body.)

Storing object's reference as an interface type

The interesting part of the program is shown in the following code fragment.

```
  void doIt(){
Base myVar1 = new Base();
myVar1.inherMethod();
X myVar2 = new A();
```

```
    myVar2.intfcMethod();

    System.out.println("");
  }//end doIt()
```

The above fragment instantiates a new object of the class named **A** , and saves a reference to that object in a reference variable of the declared type **X** .

How many ways can you save an object's reference?

Recall that a reference to an object can be held by a reference variable whose type matches any of the following:

- The class from which the object was instantiated.
- Any superclass of the class from which the object was instantiated.
- Any interface implemented by the class from which the object was instantiated.
- Any interface implemented by any superclass of the class from which the object was instantiated.
- Any superinterface of the interfaces mentioned above.

Save object's reference as implemented interface type

In this program, the type of the reference variable matches the interface named **X** , which is implemented by the class named **A** .

What does this allow you to do?

When a reference to an object is held by a reference variable whose type matches an interface implemented by the class from which the object was instantiated, that reference can be used to call any method declared in or inherited into that interface.

(That reference cannot be used to call methods not declared in or not inherited into that interface.)

In this simple case ...

The method named **intfcMethod** is declared in the interface named **X** and implemented in the class named **A** .

Therefore, the method named **intfcMethod** can be called on an object instantiated from the class named **A** when the reference to the object is held in a reference variable of the interface type.

*(The method could also be called if the reference is being held in a reference variable of declared type **A** .)*

The call to the method named **intfcMethod** causes the text **A-intfcMethod** to appear on the screen.

Back to Question 4 (p. 269)

13.6.8 Answer 3

C. Base A-intfcMethod

13.6.8.1 Explanation 3

What is runtime polymorphic behavior?

One way to describe runtime polymorphic behavior is:

The runtime system selects among two or more methods having the same signature, not on the basis of the type of the reference variable in which an object's reference is stored, but rather on the basis of the class from which the object was originally instantiated.

Illustrates simple class and interface inheritance

The program defines a class named **Base** , and a class named **A** , which extends **Base** , and implements the interface named **X** , as shown in the following fragment.

```
    class Base{
  public void inherMethod(){
    System.out.print("Base ");
  }//end inherMethod()
}//end class Base

class A extends Base implements X{
  public void inherMethod(){
    System.out.print(
                    " A-inherMethod ");
  }//end inherMethod()

  public void intfcMethod(){
    System.out.print("A-intfcMethod ");
  }//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Define an interface method

The interface named **X** declares a method named **intfcMethod** . A concrete definition of that method is defined in the class named **A** .

A new object of type Base

The code in the following fragment instantiates a new object of the class **Base** and calls its **inherMethod** . This causes the word **Base** to appear on the output screen. There is nothing special about this. This is a simple example of the use of an object's reference to call one of its instance methods.

```
  void doIt(){
Base myVar1 = new Base();
myVar1.inherMethod();
```

A new object of type A

The following fragment instantiates a new object of the class **A** and calls its **intfcMethod** . This causes the text **A-intfcMethod** to appear on the output screen. There is also nothing special about this. This is also a simple example of the use of an object's reference to call one of its instance methods.

```
    A myVar2 = new A();
myVar2.intfcMethod();

System.out.println("");
}//end doIt()
```

Not polymorphic behavior

The fact that the class named **A** implements the interface named **X** does not indicate polymorphic behavior in this case. Rather, this program is an example of simple class and interface inheritance.

Interface type is not used

The program makes no use of the interface as a type, and exhibits no polymorphic behavior *(no decision among methods having the same signature is required)* .

The class named **A** inherits an abstract method named **intfcMethod** from the interface and must define it. *(Otherwise, it would be necessary to declare the class named* ***A*** *abstract.)*

The interface is not a particularly important player in this program.

Back to Question 3 (p. 268)

13.6.9 Answer 2

A. Compiler Error

13.6.9.1 Explanation 2

Simple hierarchical polymorphic behavior

This program is designed to test your knowledge of simple hierarchical polymorphic behavior.

Implement the interface named X

This program defines a class named **A** that extends a class named **Base** , and implements an interface named **X** , as shown in the following code fragment.

```
  class A extends Base implements X{
public void inherMethod(){
  System.out.print(
                  " A-inherMethod ");
}//end inherMethod()

public void intfcMethod(){
  System.out.print("A-intfcMethod ");
}//end intfcMethod()
}//end class A

interface X{
  public void intfcMethod();
}//end X
```

Override and define some methods

The class named **A** overrides the method named **inherMethod** , which it inherits from the class named **Base** . It also provides a concrete definition of the method named **intfcMethod** , which is declared in the interface named **X** .

Store object's reference as superclass type

The program instantiates an object of the class named **A** and assigns that object's reference to a reference variable of type **Base** , as shown in the following code fragment.

```
Base myVar2 = new A();
```

Oops! Cannot call this method

So far, so good. However, the next fragment shows where the program turns sour. It attempts to call the method named **intfcMethod** on the object's reference, which was stored as type **Base** .

```
myVar2.intfcMethod();
```

Polymorphic behavior doesn't apply here

Because the class named **Base** does not define the method named **intfcMethod**, hierarchical polymorphic behavior does not apply. Therefore a reference to the object being stored as type **Base** cannot be used to directly call the method named **intfcMethod**, and the program produces a compiler error.

What is the solution?

Hierarchical polymorphic behavior is possible only when the class defining the type of the reference *(or some superclass of that class)* contains a definition for the method that is called on the reference.

There are a couple of ways that downcasting could be used to solve the problem in this case.

Back to Question 2 (p. 266)

13.6.10 Answer 1

A. Compiler Error

13.6.10.1 Explanation 1

I put this question in here just to see if you are still awake.

Can store reference as interface type

A reference to an object instantiated from a class can be assigned to any reference variable whose declared type is the name of an interface implemented by the class from which the object was instantiated, or implemented by any superclass of that class.

Define two classes and an interface

This program defines a class named **A** that extends a class named **Base**. The class named **Base** extends **Object** by default.

The program also defines an interface named **X**.

Instantiate an object

The following statement instantiates an object of the class named **A**, and attempts to assign that object's reference to a reference variable whose type is the interface type named **X**.

```
X myVar2 = new A();
```

Interface **X** is defined but not implemented

None of the classes named **A**, **Base**, and **Object** implement the interface named **X**. Therefore, it is not allowable to assign a reference to an object of the class named **A** to a reference variable whose declared type is **X**. Therefore, the program produces the following compiler error under JDK 1.3:

```
    Ap131.java:20: incompatible types
found   : A
required: X
    X myVar2 = new A();
```

Back to Question 1 (p. 265)

-end-

Chapter 14

Ap0130: Self-assessment, Comparing objects, packages, import directives, and some common exceptions[1]

14.1 Table of Contents

14.2 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. 306) to easily find and view the listings while you are reading about them.

14.3 Questions

14.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 298) ?

- A. Compiler Error
- B. Runtime Error
- C. Joe Joe false

[1] This content is available online at <http://cnx.org/content/m45310/1.6/>.

- D. Joe Joe true
- E. None of the above.

Listing 1 . Listing for Question 1.

```
    public class Ap141{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap141

class Worker{
  void doIt(){
    char[] anArray = {'J','o','e'};
    String Str1 = new String(anArray);
    String Str2 = new String(anArray);

    System.out.println(
              Str1 + " " + Str2 + " " +
              (Str1 == Str2));
  }//end doIt()
}// end class Worker
```

Table 14.1

Answer and Explanation (p. 316)

14.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 299) ?

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

Listing 2 . Listing for Question 2.

```
    public class Ap142{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap142

class Worker{
  void doIt(){
    char[] anArray = {'J','o','e'};
    String Str1 = new String(anArray);
    String Str2 = new String(anArray);

    System.out.println(
              Str1 + " " + Str2 + " " +
              Str1.equals(Str2));
  }//end doIt()
}// end class Worker
```

Table 14.2

Answer and Explanation (p. 315)

14.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 300) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 3 . Listing for Question 3.

```
    public class Ap143{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap143

class Worker{
  void doIt(){
    java.util.ArrayList ref =
            new java.util.ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                   (String)ref.get(0) +
                   (String)ref.get(1) +
                   (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

Table 14.3

Answer and Explanation (p. 313)

14.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 301) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 4 . Listing for Question 4.

```
    public class Ap144{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap144

class Worker{
  void doIt(){
    ArrayList ref =
            new ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                   (String)ref.get(0) +
                   (String)ref.get(1) +
                   (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

Table 14.4

Answer and Explanation (p. 312)

14.3.5 Question 5

What output is produced by the program shown in Listing 5 (p. 302) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 5 . Listing for Question 5.

```
    import java.util.ArrayList;

public class Ap145{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap145

class Worker{
  void doIt(){
    ArrayList ref = null;
    ref = new ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                   (String)ref.get(0) +
                   (String)ref.get(1) +
                   (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

Table 14.5

Answer and Explanation (p. 311)

14.3.6 Question 6

What output is produced by the program shown in Listing 6 (p. 303) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 6 . Listing for Question 6.

```
    import java.util.ArrayList;

public class Ap146{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap146

class Worker{
  void doIt(){
    ArrayList ref = null;
    ref.add("ABC ");
    ref.add("DEF ");
    ref.add("GHI");

    System.out.println(
                  (String)ref.get(0) +
                  (String)ref.get(1) +
                  (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

Table 14.6

Answer and Explanation (p. 310)

14.3.7 Question 7

What output is produced by the program shown in Listing 7 (p. 304) ?

- A. Compiler Error
- B. Runtime Error
- C. ABC DEF GHI
- D. None of the above.

Listing 7 . Listing for Question 7.

```
    import java.util.ArrayList;

public class Ap147{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap147

class Worker{
  void doIt(){
    ArrayList ref = null;
    ref = new ArrayList(1);
    ref.add("ABC ");
    ref.add("DEF ");

    System.out.println(
                  (String)ref.get(0) +
                  (String)ref.get(1) +
                  (String)ref.get(2));
  }//end doIt()
}// end class Worker
```

Table 14.7

Answer and Explanation (p. 309)

14.3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 305) ?

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

Listing 8 . Listing for Question 8.

```
    public class Ap148{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap148

class Worker{
  void doIt(){
    System.out.println(1.0/0);
  }//end doIt()
}// end class Worker
```

Table 14.8

Answer and Explanation (p. 309)

14.3.9 Question 9

What output is produced by the program shown in Listing 9 (p. 305) ?

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

Listing 9 . Listing for Question 9.

```
    public class Ap149{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap149

class Worker{
  void doIt(){
    System.out.println(1/0);
  }//end doIt()
}// end class Worker
```

Table 14.9

Answer and Explanation (p. 308)

14.3.10 Question 10

What output is produced by the program shown in Listing 10 (p. 306) ?

- A. Compiler Error
- B. Runtime Error
- C. AB CD EF
- D. None of the above.

Listing 10 . Listing for Question 10.

```
    public class Ap150{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap150

class Worker{
  void doIt(){
    String[] ref = {"AB ","CD ","EF "};
    for(int i = 0; i <= 3; i++){
      System.out.print(ref[i]);
    }//end forloop
    System.out.println("");
  }//end doIt()
}// end class Worker
```

Table 14.10

Answer and Explanation (p. 307)

14.4 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.

14.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0130: Self-assessment, Comparing objects, packages, import directives, and some common exceptions
- File: Ap0130.htm
- Originally published: 2004
- Published at cnx.org: 12/18/12
- Revised: 12/03/14

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.

14.6 Answers

14.6.1 Answer 10

Both of the following occur.
C. AB CD EF
B. Runtime Error

14.6.1.1 Explanation 10

Another index out of bounds

This is another example of a program that throws an index out of bounds exception. In this case, since the container is an array object, the name of the exception is **ArrayIndexOutOfBoundsException** .

Populate a three-element array object

The code in the following fragment creates and populates a three-element array object containing reference to three **String** objects.

```
  void doIt(){
String[] ref = {"AB ","CD ","EF "};
```

Access an out-of-bounds element

The next fragment attempts to access elements at indices 0 through 3 inclusive.

```
    for(int i = 0; i <= 3; i++){
  System.out.print(ref[i]);
}//end forloop
```

Since index value 3 is outside the bounds of the array, the program throws the following exception and aborts:

```
   AB CD EF
java.lang.ArrayIndexOutOfBoundsException
 at Worker.doIt(Ap150.java:22)
 at Ap150.main(Ap150.java:14)
```

Note however that the program displays the contents of the three **String** objects referred to by the contents of the first three elements in the array before the problem occurs.

That's the way it often is with runtime errors. Often, a program will partially complete its task before getting into trouble and aborting with a runtime error.

Back to Question 10 (p. 306)

14.6.2 Answer 9

B. Runtime Error

14.6.2.1 Explanation 9

A setup

If you feel like you've been had, chances are you have been had. The purpose for Question 8 (p. 304) was to set you up for this question.

Division by zero for integer types

This program deals with the process of dividing by zero for **int** types. The code in the following fragment divides the **int** value 1 by the **int** value 0.

```
  void doIt(){
  System.out.println(1/0);
}//end doIt()
```

Not the same as *double* divide by zero

However, unlike with type **double**, this process doesn't return a very large value and continue running. Rather, for type **int**, attempting to divide by zero will result in a runtime error of type **ArithmeticException** that looks something like the following under JDK 1.3:

```
   java.lang.ArithmeticException: / by zero
at Worker.doIt(Ap149.java:20)
at Ap149.main(Ap149.java:14)
```

An exercise for the student

I won't attempt to explain the difference in behavior for essentially the same problem between type **int** and type **double**. As the old saying goes, I'll leave that as an exercise for the student.

Back to Question 9 (p. 305)

14.6.3 Answer 8

C. Infinity

14.6.3.1 Explanation 8

A double *divide by zero* operation

This program deals with the process of dividing by zero for floating values of type **double** .

The following code fragment attempts to divide the double value 1.0 by the double value 0.

```
  void doIt(){
System.out.println(1.0/0);
}//end doIt()
```

The program runs successfully, producing the output **Infinity** .

What is Infinity?

Suffice it to say that Infinity is a very large number.

(Any value divided by zero is a very large number.)

At this point, I'm not going to explain it further. If you are interested in learning what you can do with **Infinity** , see the language specifications.

Back to Question 8 (p. 304)

14.6.4 Answer 7

B. Runtime Error

14.6.4.1 Explanation 7

This program illustrates an **IndexOutOfBounds** exception.

Instantiate and populate an ArrayList object

By now, you will be familiar with the kind of container object that you get when you instantiate the **ArrayList** class.

The code in the following fragment instantiates such a container, having an initial capacity of one element.

Then it adds two elements to the container. Each element is a reference to an object of the class **String** .

```
  void doIt(){
ArrayList ref = null;
ref = new ArrayList(1);
ref.add("ABC ");
ref.add("DEF ");
```

Increase capacity automatically

Because two elements were successfully added to a container having an initial capacity of only one element, the container was forced to increase its capacity automatically.

Following execution of the code in the above fragment, **String** object references were stored at index locations 0 and 1 in the **ArrayList** object.

Get reference at index location 2

The next fragment attempts to use the **get** method to fetch an element from the container at index value 2.

Index values in an **ArrayList** object begin with zero. Therefore, since only two elements were added to the container in the earlier fragment, there is no element at index value 2.

```
System.out.println(
          (String)ref.get(0) +
          (String)ref.get(1) +
          (String)ref.get(2));
```

An IndexOutOfBounds exception

As a result, the program throws an **IndexOutOfBounds** exception. The error produced under JDK 1.3 looks something like the following:

```
    Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index: 2, Size: 2
at java.util.ArrayList.RangeCheck
  (Unknown Source)
at java.util.ArrayList.get
  (Unknown Source)
at Worker.doIt(Ap147.java:27)
at Ap147.main(Ap147.java:16)
```

Attempting to access an element with a negative index value would produce the same result.

An ArrayIndexOutOfBounds exception

A similar result occurs if you attempt to access an element in an ordinary array object outside the bounds of the index values determined by the size of the array. However, in that case, the name of the exception is **ArrayIndexOutOfBounds** .

Back to Question 7 (p. 303)

14.6.5 Answer 6

B. Runtime Error

14.6.5.1 Explanation 6

The infamous *NullPointerException*

Interestingly, one of the first things that you read when you start reading Java books, is that there are *no pointers in Java* . It is likely that shortly thereafter when you begin writing, compiling, and executing simple Java programs, one of your programs will abort with an error message looking something like that **shown below** :

```
    Exception in thread "main" java.lang.NullPointerException
        at
Worker.doIt(Ap146.java:23)
        at
Ap146.main(Ap146.java:16)
```

What is a NullPointerException?

Stated simply, a **NullPointerException** occurs when you attempt to perform some operation on an object using a reference that doesn't refer to an object.

That is the case in this program

The following code fragment declares a local reference variable and initializes its value to ***null*** .

```
  void doIt(){
ArrayList ref = null;
```

(A reference variable in Java must either refer to a valid object, or specifically refer to no object (null). Unlike a pointer in C and C++, a Java reference variable cannot refer to something arbitrary.)

In this case, null means that the reference variable doesn't refer to a valid object.

No ArrayList object

Note that the code in the above fragment does not instantiate an object of the class **ArrayList** and assign that object's reference to the reference variable.

(The reference variable doesn't contain a reference to an object instantiated from the class named ***ArrayList*** *, or an object instantiated from any class for that matter.)*

Call a method on the reference

However, the code in the next fragment attempts to add a **String** object's reference to a nonexistent **ArrayList** object by calling the **add** method on the reference containing null.

```
ref.add("ABC ");
```

This results in the **NullPointerException** shown earlier (p. 310) .

What can you do with a null reference?

The only operation that you can perform on a reference variable containing null is to assign an object's reference to the variable. Any other attempted operation will result in a **NullPointerException** .

Back to Question 6 (p. 302)

14.6.6 Answer 5

C. ABC DEF GHI

14.6.6.1 Explanation 5

The purpose of this program is to

- Continue to illustrate the use of java packages, and
- Illustrate the use of the Java import directive.

Program contains an import directive

This program is the same as the program in Question 4 (p. 300) with a major exception. Specifically, the program contains the *import directive* shown in the following fragment.

```
import java.util.ArrayList;
```

A shortcut

The designers of Java recognized that having to type a fully-qualified name for every reference to a class in a Java program can become burdensome. Therefore, they provided us with a shortcut that can be used, so long as we don't need to refer to two or more class files having the same name.

Import directives

The shortcut is called an import directive.

As can be seen above, the import directive consists of the word *import* followed by the fully-qualified name of a class file that will be used in the program.

A program may have more than one import directive, with each import directive specifying the location of a different class file.

The import directive(s) must appear before any class or interface definitions in the source code.

The alternative wild-card syntax

An alternative form of the import directive replaces the name of the class with an asterisk.

The asterisk behaves as a wild-card character. It tells the compiler to use any class file that it finds in that package that matches a class reference in the source code.

The wild-card form should be used with care, because it can sometimes cause the compiler to use a class file that is different from the one that you intended to use *(if it finds the wrong one first)* .

Class file name collisions

If your source code refers to two different class files having the same name, you must forego the use of the import directive and provide fully-qualified names for those class files.

Back to Question 5 (p. 301)

14.6.7 Answer 4

A. Compiler Error

14.6.7.1 Explanation 4

The purpose of this program is to continue to illustrate the use of java packages.

No fully-qualified class names

This program is the same as the program in Question 3 (p. 299) with a major exception. Neither of the references to the **ArrayList** class use fully-qualified names in this program. Rather, the references are as shown in the following fragment.

```
ArrayList ref =
    new ArrayList(1);
```

Compiler errors

As a result, the JDK 1.3 compiler produces two error messages similar to the following:

```
    Ap144.java:20: cannot resolve symbol
symbol  : class ArrayList
location: class Worker
    ArrayList ref =
```

Doesn't know how to find the class file

This error message indicates that the compiler didn't know where to look on the disk to find the file named **ArrayList.class**

Back to Question 4 (p. 300)

14.6.8 Answer 3

C. ABC DEF GHI

14.6.8.1 Explanation 3

Illustrate the use of java packages

Since it was necessary to make use of a class to illustrate packages, this program also previews the use of the **ArrayList** class. We will be very interested in this class later when we study Java data containers.

What is an ArrayList object?

Some of this terminology may not make much sense to you at this point, but I'll go ahead and tell you anyway, just as a preview.

According to Sun, the **ArrayList** class provides a

*"Resizable-array implementation of the **List** interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the **List** interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to **Vector**, except that it is unsynchronized.)"*

Stated more simply ...

Stated more simply, an object of the **ArrayList** class can be used as a replacement for an array object. An **ArrayList** object knows how to increase its capacity on demand, whereas the capacity of a simple array object cannot change once it is instantiated.

An ArrayList object

The following statement instantiates a new object of the **ArrayList** class, with an initial capacity for one element. The initial capacity is determined by the **int** value passed to the constructor when the object is instantiated.

```
java.util.ArrayList ref =
    new java.util.ArrayList(1);
```

Back to the primary purpose ...

Getting back to the primary purpose of this program, what is the meaning of the term ***java.util*** that appears ahead of the name of the class, **ArrayList** ?

Avoiding name conflicts

One of the age-old problems in computer programming has to do with the potential for name conflicts. The advent of OOP and reusable code didn't cause that problem to go away. If anything, it made the problem worse.

For example, you and I may work as programmers for separate companies named X and Y. A company named Z may purchase our two companies and attempt to merge the software that we have written separately. Given that there are only a finite number of meaningful class names, there is a good possibility that you and I may have defined different classes with the same names. Furthermore, it may prove useful to use both of the class definitions in a new program.

Put class files in different directories

Sun's solution to the problem is to cause compiled class files to reside in different directories. Simplifying things somewhat, if your compiled file for a class named **Joe** is placed in a directory named **X**, and my compiled file for a different class named **Joe** is placed in a directory named **Y**, then source code in the same Java program can refer to those two class files as **X.Joe** and **Y.Joe**. This scheme makes it possible for the Java compiler and the Java virtual machine to distinguish between the two files having the name **Joe.class**.

The java and util directories

Again, simplifying things slightly, the code in the above fragment refers to a file named **ArrayList.class**, which is stored in a directory named **util**, which is a subdirectory of a directory named **java**.

The directory named java is the root of a directory tree containing a very large number of standard Java class files.

(As an aside, there is another directory named javax, which forms the root of another directory tree containing class files considered to be extensions to the standard class library.)

Many directories *(packages)*

Stated simply, a Java package is nothing more or less than a directory containing class files.

The standard and extended Java class libraries are scattered among a fairly large number of directories or packages *(a quick count of the packages in the JDK 1.3 documentation indicates that there are approximately 65 standard and extended packages)* .

A fully-qualified class name

With one exception, whenever you refer to a class in a Java program, you must provide a fully-qualified name for the class, including the path through the directory tree culminating in the name of the class. Thus, the following is the fully-qualified name for the class whose name is **ArrayList** .

java.util.ArrayList

(Later we will see another way to accomplish this that requires less typing effort.)

The exception

The one exception to the rule is the use of classes in the *java.lang* package, *(such as* ***Boolean*** *,* ***Class*** *, and* ***Double*** *)* . Your source code can refer to classes in the **java.lang** package without the requirement to provide a fully-qualified class name.

An ArrayList object

Now back to the use of the object previously instantiated from the class named **ArrayList** . This is the kind of object that is often referred to as a container.

(A container in this sense is an object that is used to store references to other objects.)

Many methods available

An object of the **ArrayList** class provides a variety of methods that can be used to store object references and to fetch the references that it contains.

The *add* method

One of those methods is the method named **add** .

The following code fragment instantiates three objects of the **String** class, and stores them in the **ArrayList** object instantiated earlier.

(Note that since the initial capacity of the ***ArrayList*** *object was adequate to store only a single reference, the following code causes the object to automatically increase its capacity to at least three.)*

```
    ref.add("ABC ");
ref.add("DEF ");
ref.add("GHI");
```

The get() method

The references stored in an object of the **ArrayList** class can be fetched by calling the **get** method on a reference to the object passing a parameter of type **int** .

The code in the following fragment calls the **get** method to fetch the references stored in index locations 0, 1, and 2. These references are passed to the **println** method, where the contents of the **String** objects referred to by those references are concatenated and displayed on the computer screen.

```
System.out.println(
          (String)ref.get(0) +
          (String)ref.get(1) +
          (String)ref.get(2));
```

The output

This results in the following being displayed:

ABC DEF GHI

Summary

The above discussion gave you a preview into the use of containers in general, and the **ArrayList** container in particular.

However, the primary purpose of this program was to help you to understand the use of packages in Java.

The **ArrayList** class was simply used as an example of a class file that is stored in a standard Java package.

Back to Question 3 (p. 299)

14.6.9 Answer 2

D. Joe Joe true

14.6.9.1 Explanation 2

Two String objects with identical contents

As in Question 1 (p. 297) , the program instantiates two **String** objects containing identical character strings, as shown in the following code fragment.

```
    char[] anArray = {'J','o','e'};
String Str1 = new String(anArray);
String Str2 = new String(anArray);
```

Compare objects for equality

Also, as in Question 1 (p. 297) , this program compares the two objects for equality and displays the result as shown by the call to the **equals** method in the following fragment.

```
System.out.println(
      Str1 + " " + Str2 + " " +
      Str1.equals(Str2));
```

Compare using overridden equals method

The == operator is not used to compare the two objects in this program. Instead, the objects are compared using an overridden version of the **equals** method. In this case, the **equals** method returns true, indicating that the objects are of the same type and contain the same data values.

The equals method

The **equals** method is defined in the **Object** class, and can be overridden in subclasses of **Object** . It is the responsibility of the author of the subclass to override the method so as to implement that author's concept of "equal" insofar as objects of the class are concerned.

The overridden *equals* method

The reason that the **equals** method returned true in this case was that the author of the **String** class provided an overridden version of the **equals** method.

The default *equals* method

If the author of the class does not override the **equals** method, and the default version of the **equals** method inherited from **Object** is called on an object of the class, then according to Sun:

"for any reference values x and y, this method returns true if and only if x and y refer to the same object (x==y has the value true)"

In other words, the default version of the **equals** method inherited from the class **Object** provides the same behavior as the == operator when applied to object references.

Back to Question 2 (p. 298)

14.6.10 Answer 1

C. Joe Joe false

14.6.10.1 Explanation 1

The identity operator

This program illustrates the behavior of the == operator *(sometimes referred to as the identity operator)* when used to compare references to objects.

Two String objects with identical contents

As shown in the following fragment, this program instantiates two objects of the **String** class containing identical character strings.

```
  class Worker{
void doIt(){
  char[] anArray = {'J','o','e'};
  String Str1 = new String(anArray);
  String Str2 = new String(anArray);
```

The fact that the two **String** objects contain identical character strings is confirmed by:

- Both objects are instantiated using the same array object of type **char** as input.
- When the **toString** representations of the two objects are displayed later, the display of each object produces Joe on the computer screen.

Compare object references using identity (==)

The references to the two **String** objects are compared using the == operator, and the result of that comparison is displayed. This comparison will produce either true or false. The code to accomplish this comparison is shown in the following fragment.

```
System.out.println(
      Str1 + " " + Str2 + " " +
      (Str1 == Str2));
```

The statement in the above fragment produces the following display:

Joe Joe false

How can this be false?

We know that the two objects are of the same type (**String**) and that they contain the same character strings. Why does the == operator return false?

Doesn't compare the objects

The answer lies in the fact that the above statement doesn't really compare the two objects at all. Rather, it compares the values stored in the reference variables referring to the two objects. That is not the same as comparing the objects.

References are not equal

Even though the objects are of the same type and contain the same character string, they are two different objects, located in different parts of memory. Therefore, the contents of the two reference variables containing references to the two objects are not equal.

The correct answer is *false*

The == operator returns ***false*** as it should. The only way that the == operator could return ***true*** is if both reference variables refer to the same object, *(which is not the case)* .

The bottom line is ...

The == operator cannot be used to compare two objects for equality. However, it can be used to determine if two reference variables refer to the same object.

Back to Question 1 (p. 297)

-end-

Chapter 15

Ap0140: Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references[1]

15.1 Table of Contents

15.2 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. 329) to easily find and view the listings while you are reading about them.

15.3 Questions

15.3.1 Question 1 .

What output is produced by the program shown in Listing 1 (p. 320) ?

[1] This content is available online at <http://cnx.org/content/m45302/1.5/>.

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

Listing 1 . Listing for Question 1.

```
    public class Ap151{
  public static void main(
                        String args[]){
        new Worker().doIt();
  }//end main()
}//end class Ap151

class Worker{
  void doIt(){
    Object refA = new MyClassA();
    Object refB =
              (Object)(new MyClassB());
    System.out.print(refA);
    System.out.print(refB);
    System.out.println("");
  }//end doIt()
}// end class Worker

class MyClassA{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassA

class MyClassB{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassB
```

Table 15.1

Answer and Explanation (p. 339)

15.3.2 Question 2

What output is produced by the program shown in Listing 2 (p. 321) ?

- A. Compiler Error
- B. Runtime Error
- C. OK OK

- D. OK
- E. None of the above.

Listing 2 . Listing for Question 2.

```
    public class Ap152{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap152

class Worker{
  void doIt(){
    Object ref1 = new MyClassA();
    Object ref2 = new MyClassB();
    System.out.print(ref1);

    MyClassB ref3 = (MyClassB)ref1;
    System.out.print(ref3);
    System.out.println("");
  }//end doIt()
}// end class Worker

class MyClassA{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassA

class MyClassB{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassB
```

Table 15.2

Answer and Explanation (p. 338)

15.3.3 Question 3

What output is produced by the program shown in Listing 3 (p. 322) ?

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

Listing 3 . Listing for Question 3.

```
    import java.util.Random;
import java.util.Date;

public class Ap153{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap153

class Worker{
  void doIt(){
    Random ref = new Random(
                 new Date().getTime());
    if(ref.nextBoolean()){
      throw new IllegalStateException();
    }else{
      System.out.println("OK");
    }//end else
  }//end doIt()
}// end class Worker
```

Table 15.3

Answer and Explanation (p. 336)

15.3.4 Question 4

What output is produced by the program shown in Listing 4 (p. 323) ?

- A. Compiler Error
- B. Runtime Error
- C. 5 10 15
- D. None of the above.

Listing 4 . Listing for Question 4.

```
    import java.util.NoSuchElementException;
public class Ap154{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap154

class Worker{
  void doIt(){
    MyContainer ref =
                      new MyContainer();
    ref.put(0,5);
    ref.put(1,10);
    ref.put(2,15);

    System.out.print(ref.get(0)+" ");
    System.out.print(ref.get(1)+" ");
    System.out.print(ref.get(2)+" ");
    System.out.print(ref.get(3)+" ");

  }//end doIt()
}// end class Worker

class MyContainer{
  private int[] array = new int[3];

  public void put(int idx, int data){
    if(idx > (array.length-1)){
      throw new
               NoSuchElementException();
    }else{
      array[idx] = data;
    }//end else
  }//end put()

  public int get(int idx){
    if(idx > (array.length-1)){
      throw new
               NoSuchElementException();
    }else{
      return array[idx];
    }//end else
  }//end put()

}//end class MyContainer
```

Table 15.4

Answer and Explanation (p. 334)

15.3.5 Question 5

The source code in Listing 5 (p. 324) is contained in a single file named Ap155.java

What output is produced by the program?

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

Listing 5 . Listing for Question 5.

```
    public class Ap155{
  public static void main(
                        String args[]){
    new Ap155a().doIt();
  }//end main()
}//end class Ap155

public class Ap155a{
  void doIt(){
    System.out.println("OK");
  }//end doIt()
}// end class Ap155a
```

Table 15.5

Answer and Explanation (p. 334)

15.3.6 Question 6

A Java application consists of the two source files shown in Listing 6 (p. 324) and Listing 7 (p. 325) having names of AP156.java and AP156a.java

What output is produced by this program?

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

Listing 6 . Listing for Question 6.

continued on next page

```
    public class Ap156{
  public static void main(
                        String args[]){
    new Ap156a().doIt();
  }//end main()
}//end class Ap156
```

Table 15.6

Listing 7 . Listing for Question 6.

```
    public class Ap156a{
  void doIt(){
    System.out.println("OK");
  }//end doIt()
}// end class Ap156a
```

Table 15.7

Answer and Explanation (p. 333)

15.3.7 Question 7

Explain the purpose of the terms @param and @return in Listing 8 (p. 326) . Also explain any of the other terms that make sense to you.

Listing 8 . Listing for Question 7.

```
    public class Ap157{

/**
* Returns the character at the
* specified index. An index ranges from
* <code>0</code> to
* <code>length() - 1</code>.
*
* @param  index  index of desired
* character.
* @return  the desired character.
*/
  public char charAt(int index) {
    //Note, this method is not intended
    // to be operational.  Rather, it
    // ...
    return 'a';//return dummy char
  }//end charAt method
}//end class
```

Table 15.8

Answer and Explanation (p. 332)

15.3.8 Question 8

What output is produced by the program shown in Listing 9 (p. 327) ?

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

Listing 9 . Listing for Question 8.

```
    public class Ap158{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap158

class Worker{
  void doIt(){
    char[] ref;
    System.out.print(ref);
    System.out.print(" ");
    ref[0] = 'T';
    ref[1] = 'o';
    ref[2] = 'm';
    System.out.println(ref);
  }//end doIt()
}// end class Worker
```

Table 15.9

Answer and Explanation (p. 331)

15.3.9 Question 9

What output is produced by the program shown in Listing 10 (p. 328) ?

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

Listing 10 . Listing for Question 9.

```
    public class Ap159{
  public static void main(
                        String args[]){
    new Worker().doIt();
  }//end main()
}//end class Ap159

class Worker{
  void doIt(){
    char[] ref = null;
    System.out.print(ref);
    System.out.print(" ");
    ref[0] = 'T';
    ref[1] = 'o';
    ref[2] = 'm';
    System.out.println(ref);
  }//end doIt()
}// end class Worker
```

Table 15.10

Answer and Explanation (p. 331)

15.3.10 Question 10

What output is produced by the program shown in Listing 11 (p. 329) ?

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

Listing 11 . Listing for Question 10.

```
    public class Ap160{
  public static void main(
                        String args[]){
        new Worker().doIt();
  }//end main()
}//end class Ap160

class Worker{
  void doIt(){
    char[] ref = {'J','o','e'};
    System.out.print(ref);
    System.out.print(" ");
    ref[0] = 'T';
    ref[1] = 'o';
    ref[2] = 'm';
    System.out.println(ref);
  }//end doIt()
}// end class Worker
```

Table 15.11

Answer and Explanation (p. 330)

15.4 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.

15.5 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Ap0140: Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references

- File: Ap0140.htm
- Originally published: 2004
- Published at cnx.org: 12/18/12
- Revised: 12/03/14

15.6 Answers

15.6.1 Answer 10

C. Joe Tom

15.6.1.1 Explanation 10

This is an upgrade to the program from Question 9 (p. 327) .

Success at last

The code in the following fragment resolves the compilation problem from Question 8 and the runtime problem from Question 9 (p. 327) .

```
  void doIt(){
char[] ref = {'J','o','e'};
System.out.print(ref);
System.out.print(" ");
ref[0] = 'T';
ref[1] = 'o';
ref[2] = 'm';
System.out.println(ref);
}//end doIt()
```

Simply initializing the local reference variable named **ref** satisfies the compiler, making it possible to compile the program.

Initializing the local reference variable named **ref** with a reference to a valid array object eliminates the NullPointerException that was experienced in Question 9 (p. 327) .

Printing the contents of the array object

The print statement passes the reference variable to the **print** method. The **print** method finds that the reference variable refers to a valid object *(instead of containing null as was the case in Question 9* (p. 327)) and behaves accordingly.

The **print** statement causes the initialized contents of the array object to be displayed. Then those contents are replaced with a new set of characters. The **println** statement causes the new characters to be displayed.

Back to Question 10 (p. 328)

15.6.2 Answer 9

B. Runtime Error

15.6.2.1 Explanation 9

Purposely initializing a local variable

This is an update to the program from Question 8 (p. 326) . The code in the following fragment solves the compilation problem identified in Question 8 (p. 326) .

```
  void doIt(){
char[] ref = null;
```

In particular, initializing the value of the reference variable named **ref** satisfies the compiler and makes it possible to compile the program.

A *NullPointerException*

However, there is still a problem, and that problem causes a runtime error.

The following statement attempts to use the reference variable named **ref** to print something on the screen. This results, among other things, in an attempt to call the **toString** method on the reference. However, the reference doesn't refer to an object. Rather, it contains the value ***null*** .

```
System.out.print(ref);
```

The result is a runtime error with the following infamous **NullPointerException** message appearing on the screen:

```
   java.lang.NullPointerException
at java.io.Writer.write(Writer.java:107)
at java.io.PrintStream.write(PrintStream.java:245)
at java.io.PrintStream.print(PrintStream.java:396)
at Worker.doIt(Ap159.java:22)
at Ap159.main(Ap159.java:15)
```

Back to Question 9 (p. 327)

15.6.3 Answer 8

A. Compiler Error

15.6.3.1 Explanation 8

Garbage in, garbage out

Earlier programming languages, notably C and C++ allowed you to inadvertently write programs that process the garbage left in memory by previous programs running there. This happens when the C or C++ programmer fails to properly initialize variables, allowing them to contain left-over garbage from memory.

Member variables are automatically initialized to default values

That is not possible in Java. All member variables in a Java object are automatically initialized to a default value if you don't write the code to initialize them to some other value.

Local variables are not automatically initialized

Local variables are not automatically initialized. However, your program will not compile if you write code that attempts to fetch and use a value in a local variable that hasn't been initialized or had a value assigned to it.

Print an uninitialized local variable

The statement in the following code fragment attempts to fetch and print a value using the uninitialized local variable named **ref** .

```
  void doIt(){
char[] ref;
System.out.print(ref);
```

As a result, the program refuses to compile, displaying the following error message under JDK 1.3.

```
Ap158.java:23: variable ref might not have been initialized
System.out.print(ref);
```

Back to Question 8 (p. 326)

15.6.4 Answer 7

See explanation below.

15.6.4.1 Explanation 7

The javadoc.exe program

When you download the JDK from Oracle, you receive a program named **javadoc.exe** in addition to several other programs.

The purpose of the **javadoc** program is to help you document the Java programs that you write. You create the documentation by running the **javadoc** program and specifying your source file or files as a command-line parameter. For example, you can generate documentation for this program by entering the following at the command line.

```
javadoc Ap157.java
```

Produces HTML files as output

This will produce a large number of related HTML files containing documentation for the class named **Ap157** . The primary HTML file is named **Ap157.html** . A file named **index.html** is also created.

This file can be opened in a browser to provide a viewer for all of the information contained in the many related HTML files.

(As a labor saving device, you can also specify a group of input files to the javadoc program, using wildcard characters as appropriate, to cause the program to produce documentation files for each of the input files in a single run.)

Special documentation comments and directives

If you include comments in your source code that begin with

/**

and end with

*/

they will be picked up by the javadoc program and become part of the documentation.

In addition to comments, you can also enter a variety of special directives to the javadoc program as shown in the following program.

```
    public class Ap157{

/**
* Returns the character at the
* specified index. An index ranges from
* <code>0</code> to
* <code>length() - 1</code>.
*
* @param  index  index of desired
* character.
* @return  the desired character.
*/
  public char charAt(int index) {
    //Note, this method is not intended
    // to be operational.  Rather, it
    // is intended solely to illustrate
    // the generation of javadoc
    // documentation for the parameter
    // and the return value.
    return 'a';//return dummy char
  }//end charAt method
}//end class
```

The @param and @return directives

The **@param** and **@return** directives in the source code shown above are used by the **javadoc** program for documenting information about parameters passed to and information returned from the method named **charAt** . The method definition follows the special javadoc comment.

Back to Question 7 (p. 325)

15.6.5 Answer 6

C. OK

15.6.5.1 Explanation 6

Public classes in separate files

This program meets the requirement identified in Question 5 (p. 324) . In particular, this program defines two public classes. The source code for each public class is stored in a separate file. Thus, the program compiles and executes successfully, producing the text OK on the screen.

Back to Question 6 (p. 324)

15.6.6 Answer 5

A. Compiler Error

15.6.6.1 Explanation 5

Public classes in separate files

Java requires that the source code for every public class be contained in a separate file. In this case, the source code for two public classes was contained in a single file. The following compiler error was produced by JDK 1.3:

```
    Ap155.java:18: class Ap155a is public, should be declared in a file
named Ap155a.java
public class Ap155a{
```

Back to Question 5 (p. 324)

15.6.7 Answer 4

This program produces both of the following:

- C. 5 10 15
- B. Runtime Error

15.6.7.1 Explanation 4

The NoSuchElementException

This program defines, creates, and uses a very simple container object for the purpose of illustrating the **NoSuchElementException** .

The code in the following fragment shows the beginning of a class named **MyContainer** from which the container object is instantiated.

```
  class MyContainer{
private int[] array = new int[3];

public void put(int idx, int data){
  if(idx > (array.length-1)){
    throw new
            NoSuchElementException();
  }else{
    array[idx] = data;
  }//end else
}//end put()
```

A wrapper for an array object

This class is essentially a wrapper for a simple array object of type **int** . An object of the class provides a method named **put** , which can be used to store an **int** value into the array. The **put** method receives two parameters. The first parameter specifies the index of the element where the value of the second parameter is to be stored.

Throw NoSuchElementException on index out of bounds

The **put** method tests to confirm that the specified index is within the positive bounds of the array. If not, it uses the **throw** keyword to throw an exception of the type **NoSuchElementException** . Otherwise, it stores the incoming data value in the specified index position in the array.

(Note that a negative index will cause an ArrayIndexOutOfBoundsException instead of a NoSuchElementException to be thrown.)

The get method

An object of the **MyContainer** class also provides a **get** method that can be used to retrieve the value stored in a specified index.

```
  public int get(int idx){
if(idx > (array.length-1)){
  throw new
          NoSuchElementException();
}else{
  return array[idx];
}//end else
}//end put()
```

The **get** method also tests to confirm that the specified index is within the positive bounds of the array. If not, it throws an exception of the type **NoSuchElementException** . Otherwise, it returns the value stored in the specified index of the array.

(As noted earlier, a negative index will cause an ArrayIndexOutOfBoundsException instead of a NoSuchElementException to be thrown.)

The NoSuchElementException

Thus, this container class illustrates the general intended purpose of the **NoSuchElementException** .

Instantiate and populate a container

The remainder of the program simply exercises the container. The code in the following fragment instantiates a new container, and uses the **put** method to populate each of its three available elements with the values 5, 10, and 15.

```
  void doIt(){
MyContainer ref =
                new MyContainer();
ref.put(0,5);
ref.put(1,10);
ref.put(2,15);
```

Get and display the data in the container

Then the code in the next fragment uses the **get** method to get and display the values in each of the three elements, causing the following text to appear on the screen:

5 10 15

```
        System.out.print(ref.get(0)+" ");
    System.out.print(ref.get(1)+" ");
    System.out.print(ref.get(2)+" ");
```

One step too far

Finally, the code in the next fragment goes one step too far and attempts to get a value from index 3, which is outside the bounds of the container.

```
        System.out.print(ref.get(3)+" ");
```

This causes the **get** method of the container object to throw a **NoSuchElementException** . The program was not designed to handle this exception, so this causes the program to abort with the following text showing on the screen:

```
   5 10 15 java.util.NoSuchElementException
at MyContainer.get(Ap154.java:49)
at Worker.doIt(Ap154.java:30)
at Ap154.main(Ap154.java:15)
```

(Note that the values of 5, 10, and 15 were displayed on the screen before the program aborted and displayed the error message.)

Back to Question 4 (p. 322)

15.6.8 Answer 3

This program can produce either of the following depending on the value produced by a random boolean value generator:

- B. Runtime Error
- C. OK

15.6.8.1 Explanation 3

Throwing an exception

This program illustrates the use of the **throw** keyword to throw an exception.

*(Note that the **throw** keyword is different from the **throws** keyword.)*

Throw an exception if random boolean value is true

A random **boolean** value is obtained. If the value is true, the program throws an **IllegalStateException** and aborts with the following message on the screen:

```
   java.lang.IllegalStateException
at Worker.doIt(Ap153.java:29)
at Ap153.main(Ap153.java:20)
```

If the random **boolean** value is false, the program runs to completion, displaying the text OK on the screen.

Instantiate a Random object

The following code fragment instantiates a new object of the **Random** class and stores the object's reference in a reference variable named **ref** .

```
  void doIt(){
Random ref = new Random(
             new Date().getTime());
```

I'm not going to go into a lot of detail about the **Random** class. Suffice it to say that an object of this class provides methods that will return a pseudo random sequence of values upon successive calls. You might think of this object as a random value generator.

Seeding the random generator

The constructor for the class accepts a **long** integer as the seed for the sequence.

*(Two **Random** objects instantiated using the same seed will produce the same sequence of values.)*

In this case, I obtained the time in milliseconds, relative to January 1, 1970, as a **long** integer, and provided that value as the seed. Thus, if you run the program two times in succession, with a time delay of at least one millisecond in between, the random sequences will be different.

Get a random boolean value

The code in the next fragment calls the **nextBoolean** method on the **Random** object to obtain a random boolean value. *(Think of this as tossing a coin with true on one side and false on the other side.)*

```
  if(ref.nextBoolean()){
throw new IllegalStateException();
```

Throw an exception

If the **boolean** value obtained in the above fragment is true, the code instantiates a new object of the **IllegalStateException class** , and uses the **throw** keyword to throw an exception of this type.

Program aborts

The program was not designed to gracefully handle such an exception. Therefore the program aborts, displaying the error message shown earlier.

Don't throw an exception

The code in the next fragment shows that if the **boolean** value tested above is false, the program will display the text OK and run successfully to completion.

```
  }else{
System.out.println("OK");
}//end else
}//end doIt()
```

You may need to run the program several times to see both possibilities.

Back to Question 3 (p. 321)

15.6.9 Answer 2

The answer is both of the following:

- D. OK
- B. Runtime Error

15.6.9.1 Explanation 2

One cast is allowable ...

It is allowable, but not necessary, to cast the type of an object's reference toward the root of the inheritance hierarchy.

It is also allowable to cast the type of an object's reference along the inheritance hierarchy toward the actual class from which the object was instantiated.

Another cast is not allowable ...

However, *(excluding interface type casts)* , it is not allowable to cast the type of an object's reference in ways that are not related in a subclass-superclass inheritance sense. For example, you cannot cast the type of an object's reference to the type of a sibling of that object.

Two sibling classes

The code in the following fragment defines two simple classes named **MyClassA** and **MyClassB** . By default, each of these classes extends the class named **Object** . Therefore, neither is a superclass of the other. Rather, they are siblings.

```
  class MyClassA{
public String toString(){
  return "OK ";
}//end test()
}//end class MyClassA

class MyClassB{
  public String toString(){
    return "OK ";
  }//end test()
}//end class MyClassB
```

Instantiate one object from each sibling class

The code in the next fragment instantiates one object from each of the above classes, and stores references to those objects in reference variables of type Object.

Then the code causes the overridden **toString** method of one of the objects to be called by passing that object's reference to the **print** method.

```
  void doIt(){
Object ref1 = new MyClassA();
Object ref2 = new MyClassB();
System.out.print(ref1);
```

The code in the above fragment causes the text OK to appear on the screen.

Try to cast to a sibling class type

At this point, the reference variable named **ref1** holds a reference to an object of type **MyClassA** . The reference is being held as type **Object** .

The statement in the next fragment attempts to cast that reference to type **MyClassB** , which is a sibling of the class named **MyClassA** .

```
MyClassB ref3 = (MyClassB)ref1;
```

A ClassCastException

The above statement causes a **ClassCastException** to be thrown, which in turn causes the program to abort. The screen output is shown below:

```
   OK java.lang.ClassCastException:MyClassA
at Worker.doIt(Ap152.java:24)
at Ap152.main(Ap152.java:14)
```

(Note that the text OK appeared on the screen before the program aborted and displayed diagnostic information on the screen.)

Back to Question 2 (p. 320)

15.6.10 Answer 1

C. OK OK

15.6.10.1 Explanation 1

Type conversion

This program illustrates type conversion up the inheritance hierarchy, both with and without a cast.

Store object's reference as type Object

The following fragment instantiates a new object of the class named **MyClassA** , and stores that object's reference in a reference variable of type **Object** . This demonstrates that you can store an object's reference in a reference variable whose type is a superclass of the class from which the object was instantiated, with no cast required.

```
  class Worker{
void doIt(){
  Object refA = new MyClassA();
```

Cast object's reference to type Object

The code in the next fragment instantiates an object of the class named **MyClassB** , and stores the object's reference in a reference variable of type **Object** , after first casting the reference to type **Object** . This, and the previous fragment demonstrate that while it is allowable to cast a reference to the superclass type before storing it in a superclass reference variable, such a cast is not required.

```
Object refB =
      (Object)(new MyClassB());
```

Type conversion and assignment compatibility

This is part of a larger overall topic commonly referred to as type conversion. It also touches the fringes of something that is commonly referred to as assignment-compatibility.

Automatic type conversions

Some kinds of type conversions happen automatically. For example, you can assign a value of type **byte** to a variable of type **int** and the type conversion will take place automatically.

Cast is required for narrowing conversions

However, if you attempt to assign a value of type **int** to a variable of type **byte** , the assignment will not take place automatically. Rather, the compiler requires you to provide a cast to confirm that you accept responsibility for the conversion, which in the case of **int** to byte could result in the corruption of data.

Automatic conversions up the inheritance hierarchy

When working with objects, type conversion takes place automatically for conversions toward the root of the inheritance hierarchy. Therefore, conversion from any class type to type **Object** happen automatically. However, conversions in the direction away from the root require a cast.

(Conversion from any class type to any superclass of that class also happens automatically.)

Polymorphic behavior

The code in the next fragment uses polymorphic behavior to display the contents of the two String objects.

```
    System.out.print(refA);
System.out.print(refB);
```

No cast required

This works without the use of a cast because the **print** method calls the **toString** method on any object's reference that it receives as an incoming parameter. The **toString** method is defined in the **Object** class, and overridden in the **String** class. Polymorphic behavior dictates that in such a situation, the version of the method belonging to the object will be called regardless of the type of the reference variable holding the reference to the object.

When would a cast be required?

Had the program attempted to call a method on the reference that is not defined in the **Object** , class, it would have been necessary to cast the reference down the inheritance hierarchy in order to successfully call the method.

Back to Question 1 (p. 319)

-end-

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Java OOP Self-Assessment*
Edited by: R.G. (Dick) Baldwin
URL: http://cnx.org/content/col11987/1.1/
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0005: Preface to OOP Self-Assessment"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45252/1.8/
Pages: 1-2
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0010: Self-assessment, Primitive Types"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45284/1.9/
Pages: 3-25
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0020: Self-assessment, Assignment and Arithmetic Operators"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45286/1.6/
Pages: 27-55
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0030: Self-assessment, Relational Operators, Increment Operator, and Control Structures"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45287/1.4/
Pages: 57-82
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0040: Self-assessment, Logical Operations, Numeric Casting, String Concatenation, and the toString Method"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45260/1.6/
Pages: 83-103
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0050: Self-assessment, Escape Character Sequences and Arrays"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45280/1.8/
Pages: 105-133
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Ap0060: Self-assessment, More on Arrays"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45264/1.5/
Pages: 135-160

Module: "Ap0070: Self-assessment, Method Overloading"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45276/1.5/
Pages: 161-178

Module: "Ap0080: Self-assessment, Classes, Constructors, and Accessor Methods"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45279/1.5/
Pages: 179-198

Module: "Ap0090: Self-assessment, the super keyword, final keyword, and static methods"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45270/1.5/
Pages: 199-217

Module: "Ap0100: Self-assessment, The this keyword, static final variables, and initialization of instance variables"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45296/1.4/
Pages: 219-241

Module: "Ap0110: Self-assessment, Extending classes, overriding methods, and polymorphic behavior"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45308/1.5/
Pages: 243-264

Module: "Ap0120: Self-assessment, Interfaces and polymorphic behavior"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45303/1.5/
Pages: 265-295

Module: "Ap0130: Self-assessment, Comparing objects, packages, import directives, and some common exceptions"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45310/1.6/
Pages: 297-317

Module: "Ap0140: Self-assessment, Type conversion, casting, common exceptions, public class files, javadoc comments and directives, and null references"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45302/1.5/
Pages: 319-340

Java OOP Self-Assessment

Welcome to my book titled OOP Self-Assessment. This is a self-assessment test designed to help you determine how much you know about object-oriented programming (OOP) using Java. In addition to being a self-assessment test, it is also a major learning tool. Each module consists of about ten to twenty questions with answers and explanations on two or three specific topics. In many cases, the explanations are extensive. You may find those explanations to be very educational in your journey towards understanding OOP using Java.

About OpenStax-CNX

Rhaptos is a web-based collaborative publishing system for educational material.