AP0070: Self-assessment, Method Overloading*

Richard Baldwin

Abstract

Part of a self-assessment test designed to help you determine how much you know about method overloading in Java.

1 Table of Contents

2 Preface

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

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

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

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

3 Questions

3.1 Question 1 .

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

- A. Compiler Error
- B. Runtime Error

*Version 1.3: Dec 4, 2012 2:04 pm -0600
†http://creativecommons.org/licenses/by/3.0/

- C. 9 17.64
- D. None of the above

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
```

Answer and Explanation (p. 15)

3.2 Question 2

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

- 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
```

Answer and Explanation (p. 14)

3.3 Question 3

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

- A. Compiler Error
- B. Runtime Error
- C. 10 17.64
- D. None of the above

Listing 3: Listing for Question 3.

```
    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
```

Answer and Explanation (p. 13)

3.4 Question 4

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

- A. Compiler Error
- B. Runtime Error
- C. 9 17.64
- D. None of the above

Listing 4: Listing for Question 4.

```
    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
```

Answer and Explanation (p. 13)

3.5 Question 5

Which of the following is produced by the program shown in Listing 5 (p. 5) ?

NOTE:

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
```

Answer and Explanation (p. 11)

3.6 Question 6

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

- 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
```

Answer and Explanation (p. 10)

3.7 Question 7

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

- 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
```

Answer and Explanation (p. 10)

3.8 Question 8

What output is produced by the program shown in Listing 8 (p. 7) ?

- 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
```

Answer and Explanation (p. 9)

4 Listings

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

5 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Ap0070: Self-assessment, Method Overloading
- File: Ap0070.htm
- Originally published: 2002
- Published at cnx.org: December 4, 2012

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

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

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

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

6 Answers

6.1 Answer 8

C. SubC SuperC

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. 7)

6.2 Answer 7

C. Test Object

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. 7)

6.3 Answer 6

C. Test DumIntfc

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. 6)

6.4 Answer 5

NOTE:

```
C.  float 2.14748365E9
float 9.223372E18
double 4.2
```

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.

NOTE:

```
    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:

NOTE:

```
    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.

NOTE:

```
    long y = 9223372036854775807L;
square(y);
```

The above code fragment produced the following output:

NOTE:

```
    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.

NOTE:

```
    int x = 2147483647;
square((double)x);
long y = 9223372036854775807L;
square((double)y);
```

The above modification would cause the program to produce the following output:

NOTE:

```
    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. 5)

6.5 Answer 4

C. 9 17.64

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. 4)

6.6 Answer 3

A. Compiler Error

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:

NOTE:

```
    Ap081.java:28: square(double) is already defined
in Worker

  public double square(double y){
```

Back to Question 3 (p. 3)

6.7 Answer 2

C. float 9.0 double 17.64

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. 2)

6.8 Answer 1

C. 9 17.64

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. 1) . 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. 1)

-end-