

Lab Exercise: Primitives





About Intertech

Thank you for choosing Intertech for your training. The next page of this document will get you started with your lab if you'd like to skip ahead. Below is a brief overview of Intertech as well as a promo code for you for future live Intertech trainings.

Intertech (www.intertech.com) is the largest combined software developer **training** and **consulting** firm in Minnesota. Our unique blend of training, consulting, and mentoring has empowered technology teams in small businesses, Fortune 500 corporations and government agencies since 1991.

Our training organization offers live in-classroom and online deliveries, private on-site deliveries, and on-demand options. We cover a broad spectrum of .NET, Java, Agile/Scrum, Web Development, and Mobile technologies. See more information on our training and search for courses by [clicking here](#).

We appreciate you choosing Intertech!

Lab Exercise

Primitives

Working with primitive data of a singular type is usually not too difficult. Unexpected results may be seen, however, when you start to use primitives of different types interchangeably. In particular, when working with numeric data (integers and decimals), you may have to use casting operations to get data into the right type and you need to be aware of certain precision loss when casting between types.

Specifically, in this lab you will:

- Declare a number of primitive type references (local variables).
- Explore implicit upcast between primitive types.
- Use cast operator in downcast situations.
- Use comments to include/exclude code from compiling/executing.
- Revisit static variables.
- Optionally learn how to replace some primitives with big number objects.

Scenario

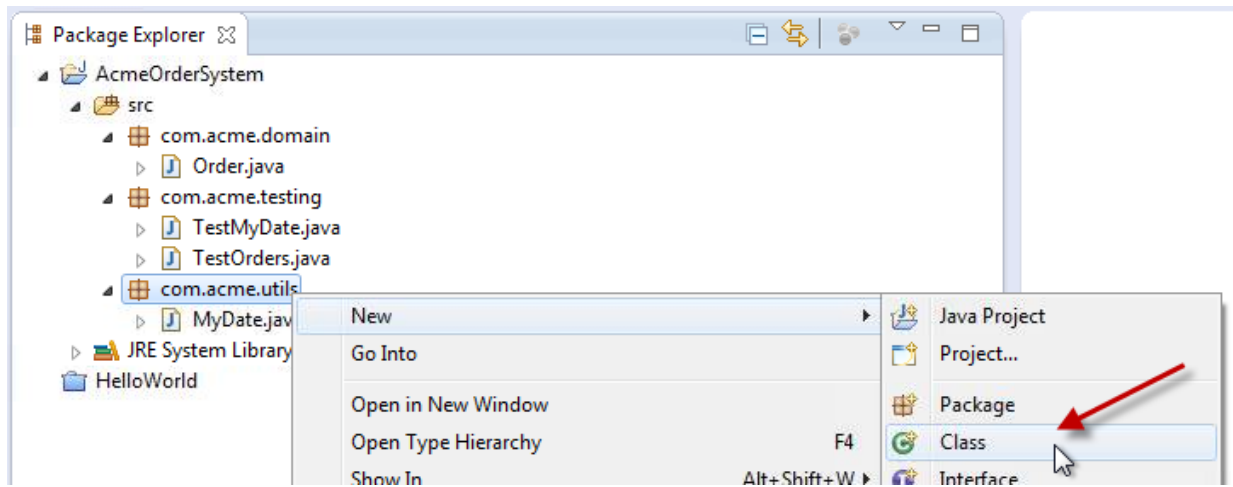
Given that Acme has so many products and works with so many different units of measure, the project team leader asks you to create a conversion service class that converts values between various weight and volume units of measure.



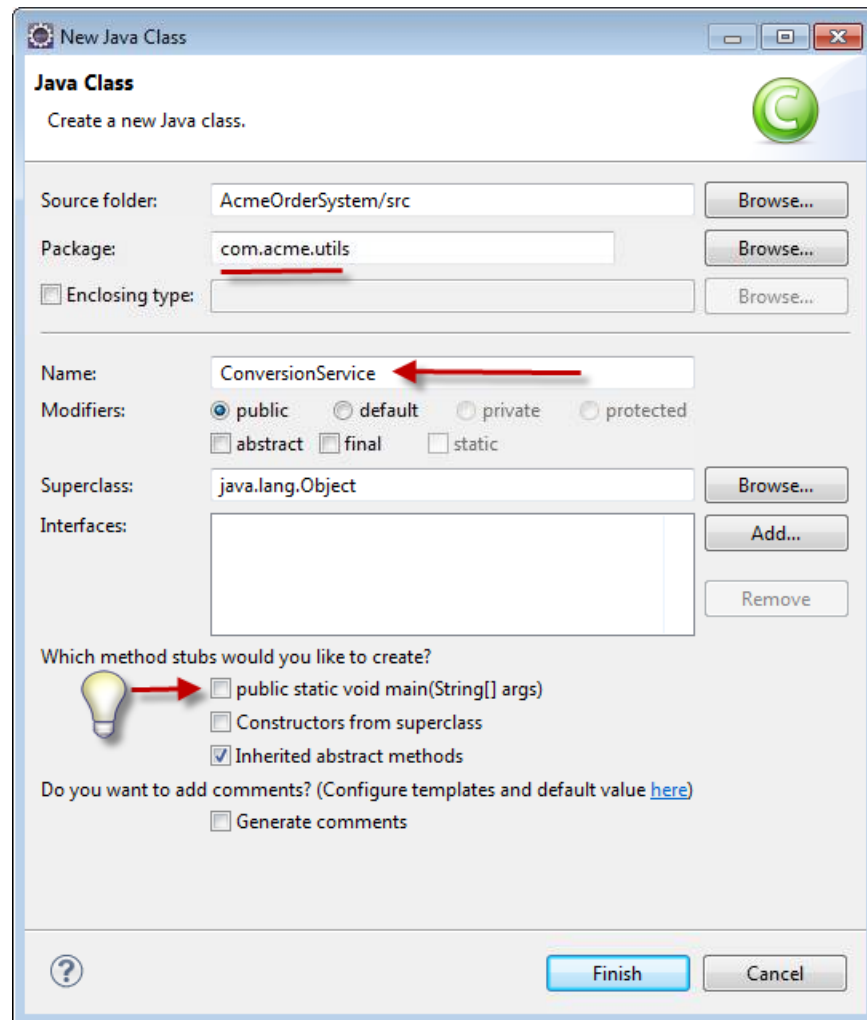
Step 1: Create the ConversionService class.

In this step, you create a class full of static variables and methods to perform conversions between kilograms and various units of weight measure. You also add static variables and methods to convert liters into various units of volume measure.

- 1.1** In the Package Explorer view, right-click on the `com.acme.utils` folder in the `AcmeOrderSystem` project and select **New > Class**.

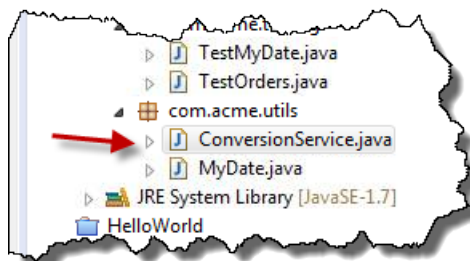


1.1.1 In the dialog window that displays, enter `ConversionService` as the name of the class, and press the *Finish* button.



Note: Don't add a main method to `ConversionService`, as it is not the application's starting point.

This should create a `ConversionService.java` file in the `com.acme.utils` folder of the `src` folder in the project and open a `ConversionService.java` editor for you to add code.



1.1.2 Add (and optionally comment) the following conversion value static variables to the ConversionService class.

```
/**
 * weight conversion rates
 */
// conversion rate for 1 kilogram to pounds
public static double kilogramToPounds = 2.2046;
// conversion rate for 1 kilogram to grams
public static int kilogramToGrams = 1000;
// conversion rate for 1 kilogram to weight ounces
public static double kilogramToOunces = 35.274;

/**
 * volume conversion rates
 */
// conversion rate for 1 liter to fluid ounces
public static float literToFluidOunce = 33.814f;
// conversion rate for 1 liter to gallons
public static float literToGallon = 0.2642f;
// conversion rate for 1 liter to pints
public static float literToPints = 2.1134f;
// conversion rate for 1 liter to milliliters
public static short literToMilliliters = 1_000;
```

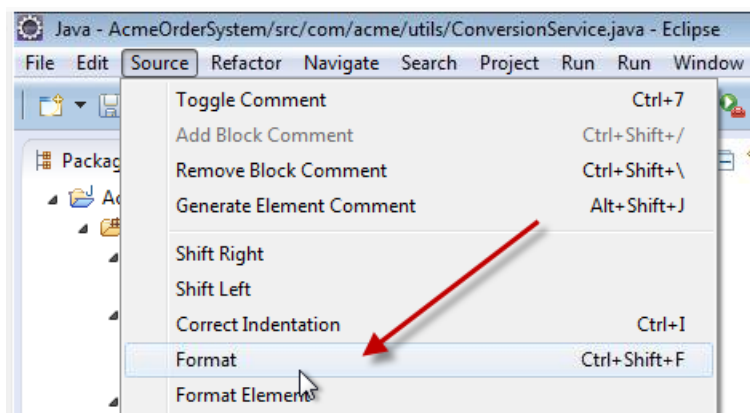
1.1.3 Now add the following conversion static methods to the ConversionService.

```
public static double pounds (double kilograms){
    return kilograms * kilogramToPounds;
}
public static int grams (int kilograms){
    return kilograms * kilogramToGrams;
}
public static double ounces (double kilograms){
    return kilograms * kilogramToOunces;
}
public static float fluidOunces(float liters){
    return liters * literToFluidOunce;
}

public static float gallons(float liters){
    return liters * literToGallon;
}
public static float pints (float liters){
    return liters * literToPints;
}
public static int millileters (int liters) {
    return liters * literToMilliliters;
}
```



Note: after entering all the code, your ConversionService.java file may not be space per Java conventions. To get your code back into good Java coding conventions format, select **Source > Format** from the Eclipse menu bar (or just hit Control-Shift-F) as shown below.



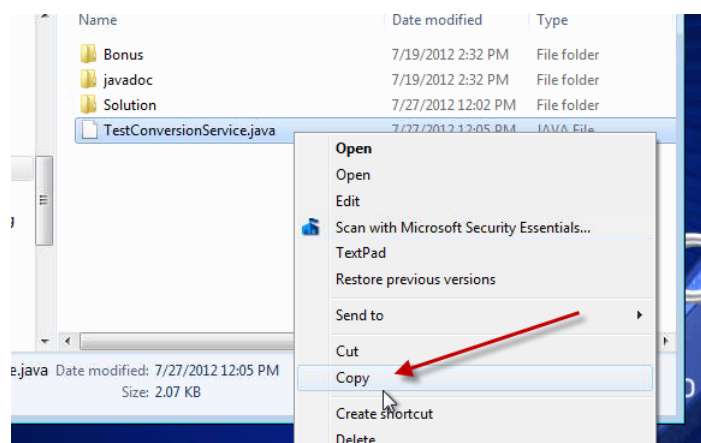
- 1.2 Save your file, and fix any compiler errors in the ConversionService class before moving to the next step.

Step 2: Run TestConversionService

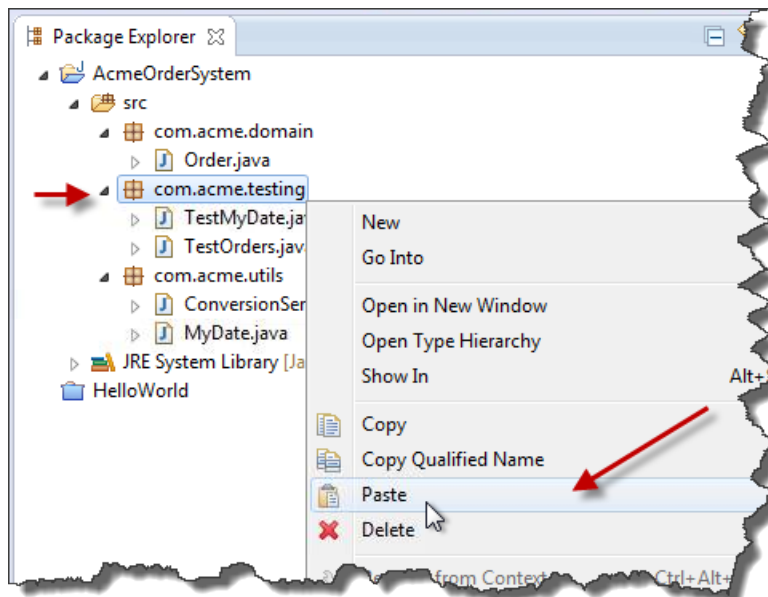
A program is provided to test your new class. You don't need to write this program, as it is available in the lab folder. In this step, you bring the test class into Eclipse and use it to test your new ConversionService class.

- 2.1 Import TestConversionService.java from the IntertechLearnJava zip (found in the Resources section for the "HelloWorld Lab").

2.1.1 Right-click on the file, and request a copy of the file.



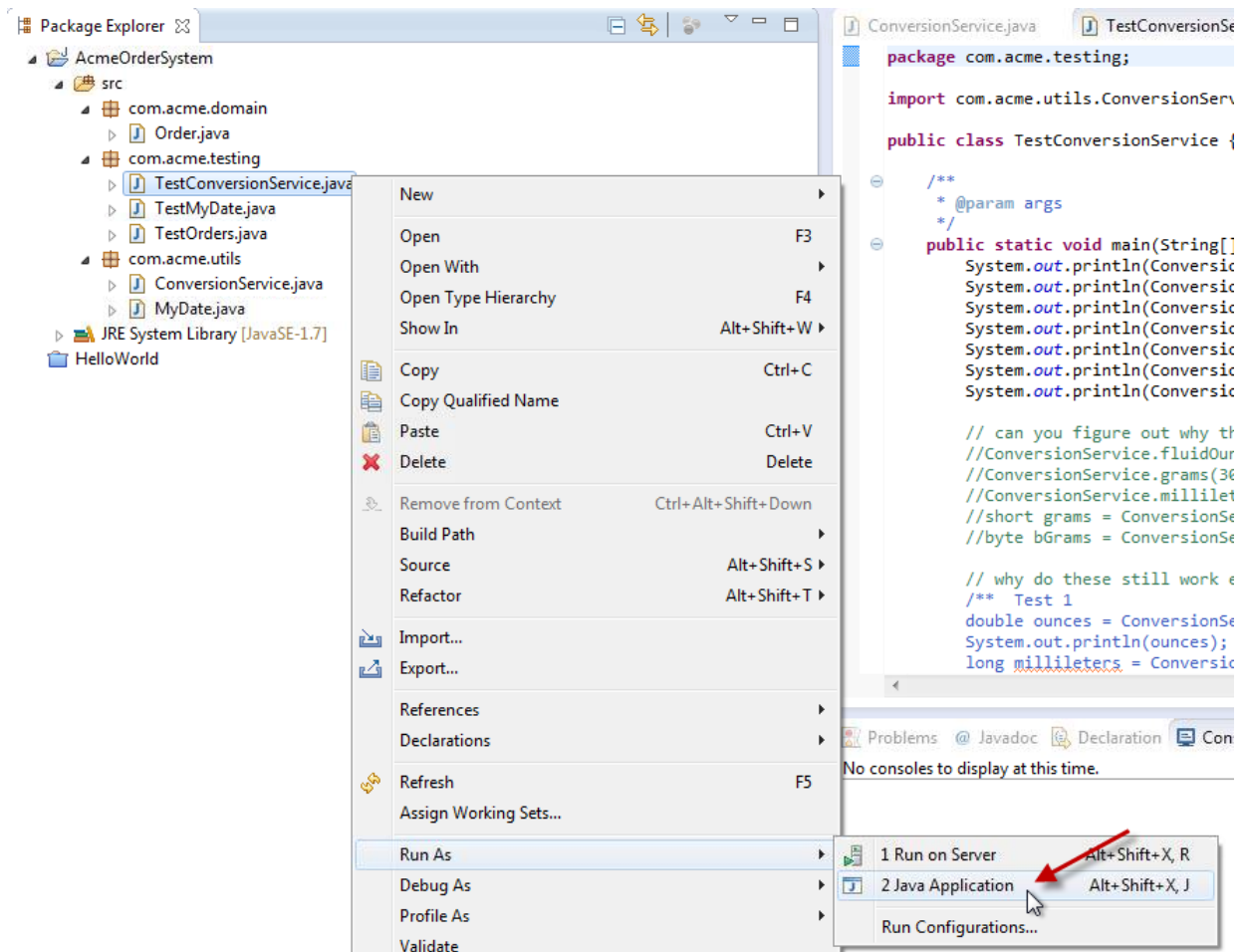
- 2.1.2 Back in the Eclipse IDE, right-click on the com.acme.testing package in the src folder of the AcmeOrderSystem project, and request to paste the copied file into the project.



2.2 Fix any errors in `ConversionService.java` or the `TestConversionService.java` file. The most likely place for an error is in the `ConversionService` type. If you didn't follow the directions exactly, the test program can fail to compile.

2.3 Run the `TestConversionService` class.

2.3.1 Right-click on `TestConversionService` and select *Run As > Java Application*.



2.3.2 When you run the test, the output in the Console view should look like the following. If it looks different, go back and try to fix it. Ask for help if you get stuck.

```
37.1954
0.58124
30000
40000
1763.7
13.94844
16.97542
```


Step 3: Understanding Downcast and Upcast

With your service and test classes working, now it is time to try out some work with numeric primitives in order to understand downcast and upcast. In particular, you want to know when/where implicit and explicit casting is required. You also want to know why some casting can cause precision problems.

- 3.1** Use the cast operator. Note that the following chunk of code is commented out in the middle of the `TestConversionService` `main()` method.

```
// can you figure out why these do not compile
//ConversionService.fluidOunces(1.1);
//ConversionService.grams(30L);
//ConversionService.millileters(4.0);
//short grams = ConversionService.grams(30);
//byte byteGrams = ConversionService.grams(30);
```

- 3.1.1** Uncomment each of the lines of code in this section (there are five lines of code to uncomment). When you do, you will notice Eclipse presents you with a compiler error.



```
System.out.println(ConversionService.pints(6.6f));
System.out.println(ConversionService.pounds(7.7));

// can you figure out why these do not compile
ConversionService.fluidOunces(1.1);
//ConversionService.grams(30L);
//ConversionService.millileters(4.0);
//short grams = ConversionService.grams(30);
//byte byteGrams = ConversionService.grams(30);
```

- 3.1.2** Can you determine why each of these lines causes a compiler error? Fix each line of code so that it compiles using an appropriate cast operation. Here is the first one to get you started.

```
ConversionService.fluidOunces((float)1.1);
```

- 3.2** Once you have uncommented all five lines of code and fixed the compiler errors, run you `TestConversionService` again to make sure everything still

works. You may want to add two lines of code (as shown below) to see grams and byteGrams results in the Console view.

```
System.out.println(grams);
System.out.println(byteGrams);
```

3.3 Why does some code work without casting? Note that another chunk of code in the main() method of TestConversionService is also commented out.

3.3.1 Uncomment the code in this section by removing the block comment around the code.

```
// why do these still work even though the types are different
/*
System.out.println("-----");
double ounces = ConversionService.fluidOunces(1.1f);
System.out.println(ounces);
long millileters = ConversionService.millileters(40);
System.out.println(millileters);
double decimalMillileters = ConversionService.millileters(40);
System.out.println(decimalMillileters);
short s = 30;
System.out.println(ConversionService.grams(s));
byte b = 4;
System.out.println(ConversionService.millileters(b));
char z = 'z';
System.out.println(ConversionService.millileters(z));
System.out.println(ConversionService.gallons(200));
System.out.println(ConversionService.ounces(50.5f));
System.out.println(ConversionService.pints(6L));
System.out.println(ConversionService.pounds(7L));
*/
```

3.3.2 In each call to a static method on ConversionService in this section of code the type of the input is not the type of the actual parameter in the ConversionService method. However, when you uncomment this code, it compiles. When you run TestConversionService, it executes and should produce results like you see below.

```
37.1954
```

```
0.58124
30000
40000
1763.7
13.94844
16.97542
-----
37.19540023803711
40000
40000.0
30000
4000
122000
52.84
1781.337
12.6804
15.432200000000002
```

3.3.3 How does this work? Can you identify the principal at work here?

3.4 See that not all casting is safe. Finally, note the last set of commented code in the `main()` method of `TestConversionService`.

3.4.1 Uncomment this code by removing the code blocks around the two comparison tests.

```
// compare these results. Can you tell why they are different?
/*
System.out.println("-----");
float bigGallons =
ConversionService.gallons(123456789123456789L);
System.out.println(bigGallons);
double bigGallons2 = 123456789123456789L * 0.2642;
System.out.println(bigGallons2);
*/

/*
System.out.println("-----");
int bigGrams = ConversionService.grams(1234567890);
System.out.println(bigGrams);
long bigGrams2 = 1234567890L * 1000L;
System.out.println(bigGrams2);
*/
```

3.4.2 Run `TestConversionService` again. The output should look like that shown below. Importantly, note the last two sets of numbers.

```
37.1954
0.58124
30000
40000
1763.7
13.94844
16.97542
-----
37.19540023803711
40000
40000.0
30000
4000
122000
52.84
1781.337
12.6804
15.4322000000000002
-----
3.26172852E16
3.261728368641728E16
-----
1912276048
1234567890000
```

3.4.3 Why are the results of these calculations different? Notice that these two chunks of code each call one of the ConversionService methods and store it in a local variable (bigGallons and bigGrams). The same calculation is also performed using hard coded numbers in the next lines of code. Therefore, these calculations should return the same values, but they do not. Why? Can you explain what went wrong?

```
1781.337  
12.6804  
15.4322000000000002
```

```
-----  
3.26172852E16  
3.261728368641728E16  
-----
```

```
1912276048  
12345678900000
```

← these numbers should be the same

← these numbers should be the same

Lab Solution

ConversionService.java

```
package com.acme.utils;
public class ConversionService {
    /**
     * weight conversion rates
     */
    // conversion rate for 1 kilogram to pounds
    public static double kilogramToPounds = 2.2046;
    // conversion rate for 1 kilogram to grams
    public static int kilogramToGrams = 1000;
    // conversion rate for 1 kilogram to weight ounces
    public static double kilogramToOunces = 35.274;
    /**
     * volume conversion rates
     */
    // conversion rate for 1 liter to fluid ounces
    public static float literToFluidOunce = 33.814f;
    // conversion rate for 1 liter to gallons
    public static float literToGallon = 0.2642f;
    // conversion rate for 1 liter to pints
    public static float literToPints = 2.1134f;
    // conversion rate for 1 liter to milliliters
    public static short literToMilliliters = 1_000;
    public static double pounds(double kilograms) {
        return kilograms * kilogramToPounds;
    }
    public static int grams(int kilograms) {
        return kilograms * kilogramToGrams;
    }
    public static double ounces(double kilograms) {
        return kilograms * kilogramToOunces;
    }
    public static float fluidOunces(float liters) {
        return liters * literToFluidOunce;
    }
    public static float gallons(float liters) {
        return liters * literToGallon;
    }
    public static float pints(float liters) {
        return liters * literToPints;
    }
    public static int millileters(int liters) {
        return liters * literToMilliliters;
    }
}
```

TestConversionService.java

```
package com.acme.testing;

import com.acme.utils.ConversionService;

public class TestConversionService {

    public static void main(String[] args) {
        System.out.println(ConversionService.fluidOunces(1.1f));
        System.out.println(ConversionService.gallons(2.2f));
        System.out.println(ConversionService.grams(30));
        System.out.println(ConversionService.millileters(40));
        System.out.println(ConversionService.ounces(50));
        System.out.println(ConversionService.pints(6.6f));
        System.out.println(ConversionService.pounds(7.7));

        // can you figure out why these do not compile
        ConversionService.fluidOunces((float)1.1);
        ConversionService.grams((int)30L);
        ConversionService.millileters((int)4.0);
        short grams = (short)ConversionService.grams(30);
        byte byteGrams = (byte)ConversionService.grams(30);

        // why do these still work even though the types are
        different
        System.out.println("-----");
    };

    double ounces = ConversionService.fluidOunces(1.1f);
    System.out.println(ounces);
    long millileters = ConversionService.millileters(40);
    System.out.println(millileters);
    double decimalMillileters =
ConversionService.millileters(40);
    System.out.println(decimalMillileters);
    short s = 30;
    System.out.println(ConversionService.grams(s));
    byte b = 4;
    System.out.println(ConversionService.millileters(b));
    char z = 'z';
    System.out.println(ConversionService.millileters(z));
    System.out.println(ConversionService.gallons(200));
    System.out.println(ConversionService.ounces(50.5f));
    System.out.println(ConversionService.pints(6L));
    System.out.println(ConversionService.pounds(7L));

    // compare these results. Can you tell why they are
    different?
```

```
        System.out.println("-----");
    );
    float bigGallons =
ConversionService.gallons(123456789123456789L);
    System.out.println(bigGallons);
    double bigGallons2 = 123456789123456789L * 0.2642;
    System.out.println(bigGallons2);
    System.out.println("-----");
    );
    int bigGrams = ConversionService.grams(1234567890);
    System.out.println(bigGrams);
    long bigGrams2 = 1234567890L * 1000L;
    System.out.println(bigGrams2);
    }
}
```

Answers to questions

Step 3.1.2 – why are there compiler errors

Cannot do a downcast, double to float, implicitly

```
ConversionService.fluidOunces(1.1);
```

Cannot do a downcast, long to int, implicitly

```
ConversionService.grams(30L);
```

Need an explicit cast to go from double to int

```
ConversionService.millileters(4.0);
```

Cannot downcast return int to a short implicitly

```
short grams = ConversionService.grams(30);
```

Cannot downcast return int to a byte implicitly



Primitives

```
byte bGrams = ConversionService.grams(30);
```

Step 3.3.3 – all of these method calls are performing an implicit upcast

```
double ounces = ConversionService.fluidOunces(1.1f); // upcast
from returning float to double is implicit

System.out.println(ounces);
long millileters = ConversionService.millileters(40); // upcast
from returning int to long is implicit

System.out.println(millileters);
double decimalMillileters = ConversionService.millileters(40);
// upcast from returning int to double is implicit

System.out.println(decimalMillileters);
short s = 30;
System.out.println(ConversionService.grams(s)); // upcast from
short to int is implicit

byte b = 4;
System.out.println(ConversionService.millileters(b)); // upcast
from byte to int is implicit

char z = 'z';
System.out.println(ConversionService.millileters(z)); // cast
from char to int is implicit

System.out.println(ConversionService.gallons(200)); // cast
from int to float is allowed and is implicit (but may lose
precision)

System.out.println(ConversionService.ounces(50.5f)); // upcast
from float to double is implicit

System.out.println(ConversionService.pints(6L)); // cast from
long to float is allowed and is implicit (but may lose
precision)

System.out.println(ConversionService.pounds(7L)); // cast from
long to double is allowed and is implicit (but may lose
precision)
```

Step 3.4.3 – why don't the results match

```
float bigGallons =
ConversionService.gallons(123456789123456789L);
System.out.println(bigGallons);
// precision is lost due to working large numbers and float type
double bigGallons2 = 123456789123456789L * 0.2642;
```



```
System.out.println(bigGallons2);  
// precision is retained due to working with doubles and large  
numbers  
  
int bigGrams = ConversionService.grams(1234567890);  
System.out.println(bigGrams);  
// precision is lost due to multiplying 2 ints  
long bigGrams2 = 1234567890L * 1000L;  
System.out.println(bigGrams2);  
// precision is maintained using longs
```

Bonus Lab

Step 4: Use BigInteger and BigDecimal classes.

If the conversion service had to work with really big quantities, primitives (even 64 bit primitives) would not suffice. Change a portion of the Conversion service and test classes to use the big number classes.

4.1 Change the weight conversion rates to use BigDecimal and BigInteger.

4.1.1 Locate the kilogramToPounds, kilogramToGrams, and kilogramToOunces static variables in the ConversionService class. Modify the type of these variables to the appropriate big number type. The first one is shown below.

```
public static BigDecimal kilogramToPounds = new
BigDecimal(2.2046);
```

4.1.2 Change the corresponding pound(), grams(), and ounce() methods to take big number parameters and return big number results – in addition to using the new big number statics you defined above. The first method is shown below.

```
public static BigDecimal pounds(BigDecimal kilograms) {
    return kilograms.multiply(kilogramToPounds);
}
```

4.1.3 Update the TestConversionService class to use the new pound(), grams(), and ounce() methods that require the use of big number objects to call versus primitives.

4.2 After changing both the ConversionService and TestConversionService classes, run the TestConversionService. The results of the test should still match the results shown in step 3.4.2.

Give Intertech a call at **1.800.866.9884** or visit Intertech's website.

