Unit 1: Primitive Types

Topic 5 Lab 1: Casting & Ranges of Variables

Name:	

A casting operator applies to the expression directly to its right. For example, `(double) 5 * 2`, the `(double)` operator only affects the 5, which casts 5 to 5.0. *Then* multiplication happens resulting in 5.0 * 2, or 10.0. In `(double) (5 * 2)`, the (double) operator applies to the result of (5 * 2), since 5 * 2 is in parentheses. So 5 * 2 happens first, resulting in 10, and *then* the 10 result is cast to a double, or 10.0

Determine the **value** and **type** of each expression after it is evaluated. The first one is done for you as an example:

a) `7 / 2`

b) `7.4 / 2`

c) `(int) 7.4 / 2`

d) `(int) 7.4 / 2.0`

e) `(int) (7.4 / 2.0)`

f) `10.0 / 4.0`

g) `10 / 4.0`

h) `(double) 10 / 4.0`

j) `(double) 10 / 4`

k) `(double) (10 / 4)`

l) `(int) 5.8 + 2.4`

m) `(int) (5.8 + 2.4)`

n) `(int) 5.8 + (int) 2.4`

o) `(int) 4.5 - (double) 3`

p) `(double) 9 / 4 + (int) 5.5`

q) `(double) 6 / 4 - 3`

r) `(double) (6 / 4) - 3`

s) `(double) (6 / 4 - 3)`

t) `(double) ((int) 3.8 * 2) / 4`

	value	**type**
a)	3	int
b)		
c)		
d)		
e)		
f)		
g)		
h)		
j)		
k)		
l)		
m)		
n)		
o)		
p)		
q)		
r)		
s)		
t)		

Solutions & explanations

PREDICT: What prints after each `println` statement?

```
double num1 = 4.8;
double num2 = 5.9;
System.out.println(num1 + num2);
System.out.println((int) num1 + num2);
System.out.println(num1 + (int) num2);
System.out.println((int) num1 + (int) num2);
System.out.println((int) (num1 + num2));
```

TYPE PREDICTIONS BELOW
(the first one is done for you)

Printed value	double or int?
10.7	double

Copy/paste the code above Replit and run it; were your predictions correct? If not, what was your mistake?

Confirm answers

PREDICT: What prints after each `println` statement?

```
int num3 = 8;
int num4 = 9;
System.out.println(num3 + num4);
System.out.println((double) num3 + num4);
System.out.println(num3 + (double) num4);
System.out.println((double) num3 + (double) num4);
System.out.println((double) (num3 + num4));
```

TYPE PREDICTIONS BELOW

Printed value	double or int?

Copy/paste the code above into Replit and run it; were your predictions correct? If not, what was your mistake?

Confirm answers

Here is a code segment that declares and initializes several variables:

```
int a = 10;
int b = 15;
double y = 20.9;
double z = 25.4;
// add your code here:


System.out.println("a = " + a);
System.out.println("b = " + b);
```

← **a)** Add ***two*** *lines* of code to the code segment on the left to store the truncated `int` values of y and z into a and b, respectively, using the `(int)` casting operator.

b) Test your solution by copying all the code into a program and running it; *you should see* (without decimals):

```
a = 20
b = 25
```

Check solution

Challenge! A common issue when casting a `double` to an `int` is that the decimal is **truncated** rather than *rounded*. Fortunately, there is a cool way to use casting in order to *round a decimal to the nearest integer!*

Your challenge is to figure out what could go in place of ????? below so that this code:

```
double price = 4.85;
int roundedPrice = ?????  // something that involves casting!
System.out.println("roundedPrice = " + roundedPrice);
```

Prints out `roundedPrice` *rounded to the nearest integer* (5) rather than truncated (4):

```
roundedPrice = 5
```

Hint please!

When you have it figured out, test it a few times with different values to make it works; try setting `price` to **4.00, 4.25, 4.50, and 5.00** to make sure you get the rounded **4, 4, 5, and 5**.

Paste your solution to the right:

```
int roundedPrice =
```

Give up? (don't!)

Hmm… but what if `price` is *negative*? Does your solution work for rounding negative numbers? Try it and see. Then explain why it doesn't quite work!

Finish coding line 2 below that it will round the *negative* number -14.70 properly to **-15** (*not* -14):

```
double coldTemp = -14.70;
int roundedTemp =
System.out.println("roundedTemp = " + roundedTemp);
```

Should print: `roundedTemp = -15` (not -14!)

Hint

Paste your solution to the right:

```
int roundedTemp =
```

Give up? (don't!)

2. Use the rounding techniques above to complete the code below so that the *rounded* versions of each number are printed:

```
double num1 = 18.24;
double num2 = 212.5;
double num3 = -5.3;
double num4 = -25.77;

// complete these 4 lines of code:
int roundedNum1 =
int roundedNum2 =
int roundedNum3 =
int roundedNum4 =
```

```
System.out.println(roundedNum1);
System.out.println(roundedNum2);
System.out.println(roundedNum3);
System.out.println(roundedNum4);
```

Expected output:

```
18
213
-5
-26
```

Copy/paste the updated 4 lines of code below:

Hint, please!

Solution

3. Consider this code segment:

```
double someNum = 3 + 11 / 2;
System.out.println(someNum);
System.out.println((int) someNum);
```

a. Predict what this code will print to the screen.

b. Run the code segment in Replit to check your work; was your prediction correct? If not, *why* not? *(if you aren't sure, read the explanation!)*

a. Write your prediction here:

b. Correct? Or incorrect and why?

Explanation!

4. Consider this code segment:

```
double a = 2.0;
int b = (int) (9 / a);
double c = (double) b / 8;
int d = (int) (c + 0.5);
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
```

a. Predict what this code will print to the screen.

b. Run the code segment in Replit to check your work; was your prediction correct? If not, *why* not? (if you aren't sure, read the explanation!)

a. Write your prediction here:

```
a =
b =
c =
d =
```

b. Correct? Or incorrect and why?

Explanation

5. Consider this code segment:

```
int num1 = -7;                                    // Line 1
int num2 = 2;                                     // Line 2
int num3 = 9;                                     // Line 3
int total = num1 + num2 + num3;                   // Line 4
double average = (double) (total / 3);            // Line 5
System.out.println("The average is " + average); // Line 6
```

Which displays:

1.0

Rather than the *expected* 1.33333…

Determine ***three*** different ways you could **change** Line 5 so that it properly prints the decimal average. *Just change line 5!*

Copy/paste the updated code segment for each of your solutions:

First, explain *why* 1.0 gets displayed, rather than 1.3333…	
Solution 1:	`double average =`
Solution 2:	`double average =`

Solution 3:	`double average =`

Hint, please!

Did you get them all? Compere with these possible solutions!

Consider this *same* code segment, except in this case, let's **remove the casting from line 5:**

```
int num1 = -7;                                    // Line 1
int num2 = 2;                                     // Line 2
int num3 = 9;                                     // Line 3
int total = num1 + num2 + num3;                   // Line 4
double average = total / 3;                       // Line 5
System.out.println("The average is " + average); // Line 6
```

PREDICT: What would this code print out?	`The average is`
Copy, paste, run and see! What *was* the actual output? Were you correct?	*Explanation*

You saw that we *could* change 3 to 3.0 in line 5, but can you figure out a **fourth** approach that makes a change to one of the variables declared in lines 1 through 4, leaving line 5 as it is? *You will only need to change one variable!*

Solution 4:	

Confirm

Reflect: Which strategy of the 4 that you discovered above do you prefer and why? Do you think any one strategy is "preferred" over another? Explain!	

Fix the code!

The code below is *supposed* to output:

```
a = 2
b = 2.5
c = 5
d = 3
```

But there are several problems!

Use the **comments** to help you understand what each line is *supposed* to do:

```
public class Main
```

```
{
   public static void main(String[] args)
   {
       int a = 2;                      // a is 2
       double b = (double) (5 / a);    // divide 5 by 2 and store 2.5 in b
       int c = b * 2;                  // multiply 2.5 by 2 then store as int (5)
       int d = (int) b + 0.5;          // round b to the nearest int (3)

       System.out.println("a = " + a);
       System.out.println("b = " + b);
       System.out.println("c = " + c);
       System.out.println("d = " + d);
   }
}
```

a. See how many problems you can spot **before** copy/pasting and running it; what issues do you notice?

b. When you think you have identified all issues **then** copy/paste the segment above into Replit. Make all the changes necessary to print out the following **exactly**:

```
a = 2
b = 2.5
c = 5
d = 3
```

*(**free hint:** three lines of code need adjustments!)*

c. Paste your final fixed code below (make sure it prints the desired output exactly!):

Sample solution

LAB CONTINUES ON NEXT PAGE!

Range of `int`

Important Ideas!

The **maximum** value that can be stored in a 32-bit **`int`** variable is (2^32 - 1) = **2,147,483,647** (technically *not* 2,147,483,648, it's a long story why we subtract 1 here), and the **minimum** value is -(2^32) = **-2,147,483,648.** The "**range of `int`**" is -2,147,483,648 to 2,147,483,647

You don't have to memorize these values since they are stored in Java as **constants:** **`Integer.MAX_VALUE`** and **`Integer.MIN_VALUE`**

Note how the names use ALL_CAPS_WITH_UNDERSCORES, as per *naming conventions for constants* -- we will talk about where the **`Integer`** part comes from later (it's a class).

Copy/paste the following code:

```
int maxInt = Integer.MAX_VALUE;
int minInt = Integer.MIN_VALUE;
System.out.println("max int = " + maxInt);
System.out.println("min int = " + minInt);
```

RUN IT and you *should* see:

```
max int = 2147483647
min int = -2147483648
```

Now, add the following code below:

```
int maxInt = Integer.MAX_VALUE;
int minInt = Integer.MIN_VALUE;
System.out.println("max int = " + maxInt);
System.out.println("min int = " + minInt);
// add the following code:
int someBigPosNum = 2147483600;   // 47 less than max
int someBigNegNum = -2147483600;  // 48 greater than min
System.out.println("big pos num = " + someBigPosNum);
System.out.println("big neg num = " + someBigNegNum);
```

RUN IT and you should see:

```
max int = 2147483647
min int = -2147483648
big pos num = 2147483600
big neg num = -2147483600
```

Lastly, add the following code below but don't run it yet!

```java
int maxInt = Integer.MAX_VALUE;
int minInt = Integer.MIN_VALUE;
System.out.println("max int = " + maxInt);
System.out.println("min int = " + minInt);
int someBigPosNum = 2147483600;   // 47 less than max
int someBigNegNum = -2147483600;  // 48 greater than min
System.out.println("big pos num = " + someBigPosNum);
System.out.println("big neg num = " + someBigNegNum);
// add the following code:
someBigPosNum += 100;  // takes value above max
someBigNegNum -= 100;  // takes value below min
System.out.println("updated big pos num = " + someBigPosNum);
System.out.println("updated big neg num = " + someBigNegNum);
```

PREDICT what will happen when you run this code, in which two variables go outside the range of int:

- Will there be a **syntax** error? (i.e. Replit displays red squiggles before you even run it)
- Will the program begin running but terminate in an **exception** (crash)?
- Will the code run to completion, but with unpredictable results?
- Or, will this work out just fine with no problem?

Capture your **prediction** here:	
Now, **run** the code to see what actually happens! *Look at the output very closely…* **Describe what happens!** Is it what you expected to happen? Note the values of the two variables… did the math operations result in expected values?	*Confirm what you see!*

Find out what's going on on the next page!

Range of `int`

In Java, if an `int` variable is assigned a value that is either *larger* than **`Integer.MAX_VALUE`** or *smaller* (more negative) than **`Integer.MIN_VALUE`**, what happens is called an **overflow error.** This is a runtime error that *doesn't* lead to the program crashing or anything *seemingly* problematic, since there are *no errors in the console!*

```
max int = 2147483647
min int = -2147483648
big pos num = 2147483600
big neg num = -2147483600
UPDATED big pos num = -2147483596
UPDATED big neg num = 2147483596
➜
```

However, looking closely at the values, you can see that this led to **unexpected results** (the positive number became negative when adding to it, and the negative number became positive when subtracting -- this is **not** expected behavior). What actually happens when an "overflow" occurs is that the numbers "wrap around" -- which is why you suddenly have a positive number where you expected a negative number, and vice versa.

Moral of the story: `int` *does* have limits, so when dealing with very big numbers (positive or negative), you will want to be careful when using them inside `int` variables!

Lab continues on the next page!

Coding Challenge!

Jackson wrote the following program to get two integers from the user, then display the quotient as a *decimal* and the sum as an *integer* (with no decimal part printed). But it doesn't quite work. Help him fix his code so that he gets the expected results. For example, entering 7 and 3 *should* print 2.3333.. (with a decimal part) and 10 (an int with no decimal, i.e. *not* 10.0).

Expected output for 7 entered first and 3 entered second:

```
Enter first integer: 7
Enter second integer: 3
The quotient is 2.3333333333333335
The sum is 10
```

Actual (buggy) output:

```
Enter first integer: 7
Enter second integer: 3
The quotient is 2.0
The sum is 10
```

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);

    System.out.print("Enter first integer: ");
    int num1 = scan.nextInt();

    System.out.print("Enter second integer: ");
    int num2 = scan.nextInt();

    double quotient = num1 / num2;                // Line A
    int sum = num1 + num2;                        // Line B

    System.out.println("The quotient is " + quotient);
    System.out.println("The sum is " + sum);
  }
}
```

Figure out one way to fix this that involves changing something in Line A and/or Line B *only*.
Copy/paste this first solution below:

Sample solution

Jackson then had the idea to maybe use `nextDouble` instead of `nextInt`, so he made the changes below in **red**, but it ***still*** wouldn't work!

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);

    System.out.print("Enter first integer: ");
    double num1 = scan.nextDouble();

    System.out.print("Enter second integer: ");
    double num2 = scan.nextDouble();

    double quotient = num1 / num2;                // Line A
    int sum = num1 + num2;                        // Line B

    System.out.println("The quotient is " + quotient);
    System.out.println("The sum is " + sum);
  }
}
```

Figure out a way to fix this version of Jackson's code that involves making a change to Line A or Line B *only*.
Copy/paste this second solution below:

Sample solution

Done!
Submit in Google Classroom:

Turn in

HINTS

Hint for Problem 2 (back):

Use the following casting strategy to round a *positive* `double` value to the nearest `int`:

```
(int) (value + 0.5)
```

…or to round a *negative* `double` value to the nearest `int`:

```
(int) (value - 0.5)
```

Hint for Problem 5 (back):

- Currently, the casting is being done on the result of `int/int` division due to the parentheses around `total / 3`. You *instead* want to have the division to be either **double / int** or **int / double**…

Hints for Programming Challenge #3 (back):

- Use `scan.nextDouble()` to accept the user's value and store it in a `double` variable **instead** of `nextInt()` because if you try to store a value that is too big immediately as an `int`, you will have overflow errors and results will be unexpected, and a `double` can store *much bigger* numbers than `int`.

- Once you have the input stored as a `double`, then test it using if statements against Integer.MAX_VALUE and Integer.MIN_VALUE.

- You might want to use *nested* selection statements, for example:

```
int x = 3;

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

if (x > 4)
{
   System.out.println(x + " is greater than 4!");
}
else
{
   if (x > 0)
   {
      System.out.println(x + " is not greater than 4 but it is positive");
   }
   else
   {
      System.out.println(x + " is not greater than 0");
   }
}

System.out.println("Goodbye!");
```

Prints:

```
Hello!
3 is not greater than 4 but it is positive!
Goodbye!
```

Question 1 (back)

	expression	value	type	explanation
a)	`7 / 2`	**3**	**int**	int / int = int
b)	`7.4 / 2`	**3.7**	**double**	double / int = double
c)	`(int) 7.4 / 2`	**3**	**int**	the (int) casts 7.4 to an int of 7, and 7 / 2 is int / int which is an int value of 3
d)	`(int) 7.4 / 2.0`	**3.5**	**double**	the (int) casts 7.4 to an int of 7, and 7 / 2.0 is int / double which is a double value of 3.5
e)	`(int) (7.4 / 2.0)`	**3**	**int**	the expression (7.4 / 2.0) is evaluated first since it's in ( ), and the (int) casts the resulting double value of 3.7 to the int value of 3 (truncates)
f)	`10.0 / 4`	**2.5**	**double**	double / int = double
g)	`10 / 4.0`	**2.5**	**double**	int / double = double
h)	`(double) 10 / 4.0`	**2.5**	**double**	the (double) casts 10 to a double of 10.0, and 10.0 / 4.0 is double / double which is a double value of 2.5
j)	`(double) 10 / 4`	**2.5**	**double**	the (double) casts 10 to a double of 10.0, and 10.0 / 4 is double / int which is a double value of 2.5
k)	`(double) (10 / 4)`	**2.0**	**double**	the expression (10 / 4) is evaluated first since it's in ( ), and the (double) casts the resulting int value of 2 to the double value of 2.0
l)	`(int) 5.8 + 2.4`	**7.4**	**double**	the (int) casts 5.8 to an int of 5, and 5 + 2.4 is int + double which is a double value of 7.4
m)	`(int) (5.8 + 2.4)`	**8**	**int**	the expression (5.8 + 2.4) is evaluated first since it's in ( ), and the (int) casts the resulting double value of 8.2 to the int value of 8
n)	`(int) 5.8 + (int) 2.4`	**7**	**int**	The first (int) casts 5.8 to an int of 5, and the second (int) casts 2.4 to an int value of 2, and 5 + 2 is the int value 7
o)	`(int) 4.3 - (double) 3`	**1.0**	**double**	The (int) casts 4.5 to an int of 4, and the (double) casts 3 to 3.0, and 4 - 3.0 is int - double which is the double value of 1.0

p)	`(double) 9 / 4 + (int) 5.5`	**7.25**	**double**	The (double) casts 9 to 9.0, and the (int) casts 5.5 to 5, and 9.0 / 4 + 5 is double / int + int which becomes 2.25 + 5 = 7.25
q)	`(double) 6 / 4 - 3`	**-1.5**	**double**	The (double) casts 6 to a double of 6.0, and then 6.0 / 4 is 1.5, and then 1.5 (a double) minus 3 (an int) = -1.5 (a double)
r)	`(double) (6 / 4) - 3`	**-2.0**	**double**	The (double) casts the result of the (6 / 4), which is an int of 1, to a double of 1.0, and then 1.0 (a double) minus 3 (an int) = -2.0 (a double)
s)	`(double) (6 / 4 - 3)`	**-2.0**	**double**	The entire expression (6 / 4 - 3) evaluates first because of ( ) and since these are all ints, you get 6 / 4 is 1 and 1 - 3 is -2, then the (double) casts the -2 to -2.0
t)	`(double) ((int) 3.8 * 2) / 4`	**1.5**	**double**	This part evaluates first: `((int) 3.8 * 2)` in which (int) casts 3.8 to 3, and then 3 * 2 = 6 The expression then simplifies to: `(double) 6 / 4` and the (double) casts 6 to 6.0 and this becomes 6.0 / 4 = 1.5

(back)

Question 2 (back)

```java
double num1 = 18.24;
double num2 = 212.5;
double num3 = -5.3;
double num4 = -25.77;

int roundedNum1 = (int) (num1 + 0.5);
int roundedNum2 = (int) (num2 + 0.5);
int roundedNum3 = (int) (num3 - 0.5);
int roundedNum4 = (int) (num4 - 0.5);

System.out.println(roundedNum1);
System.out.println(roundedNum2);
System.out.println(roundedNum3);
System.out.println(roundedNum4);
```

Question 3 (back)

```
double someNum = 3 + 11 / 2;            // Line 1
System.out.println(someNum);            // Line 2
System.out.println((int) someNum);      // Line 3
```

This outputs:

```
8.0
8
```

*note that it's 8.0 -- **NOT** 8.5*

Explanation:

Line 1 sets `someNum` to the `double` value of `3 + 11 / 2`, and by order of operations, 11 / 2 gets evaluated first. But since `11 / 2` is an `int/int` operation, it evaluates to an `int` value of 5 (**not** a `double` value of 5.5, *despite the fact that* `someNum` *is declared as a double*). 3 is then added to 5, which becomes 8, because 3 + 5 is two `ints` being added, but after the entire expression is simplified to 8 (an `int`), Java *then* converts 8 to 8.0 behind the scenes and assigns it to `someNum` since `someNum` is a `double`.

Line 2 then prints the `double` value of 8.0.

Line 3 prints that value *cast* as the `int` value of 8 (reminder that casting to an `int` **truncates** the decimal -- it *doesn't* round it -- which is why it's 8 and not 9).

Question 4 (back)

```
double a = 2.0;                    // Line 1
int b = (int) (9 / a);             // Line 2
double c = (double) b / 8;         // Line 3
int d = (int) (c + 0.5);           // Line 4
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
```

This outputs:

CONSOLE

```
a = 2.0
b = 4
c = 0.5
d = 1
```

Explanation:
Line 2 sets `b` to the expression `9 / a` after it is evaluated and casted as an `int`. The expression `9 / a` gets evaluated *before* the casting because it is in parentheses: `(9 / a)` Since `a` is a `double`, this evaluates to a `double` value of 9/2.0, or 4.5, then cast to an `int` value of 4 (truncated). `b` is now 4.

Line 3 sets `c` to the expression `b / 8` with `b` casted to a `double`. Since `b / 8` is *not* in parentheses, the `b` gets casted to a `double` *before* the division. And so `c` is set equal to the value of the expression 4.0 / 8, which is 0.5 (a `double`, because this is `double/int`).

Lastly, line 4 performs the "rounding" operation by *first* adding 0.5 to the value of `c` (because `c + 0.5` is inside parentheses) and then casting the result to an `int`. The result here is 0.5 + 0.5 = 1.0, which when cast to an `int`, is 1, so `d` is 1.

Question 5 (back)

First, explain *why* 1.0 gets displayed, rather than 1.3333…	The problem is that the `(double)` doesn't cast until *after* `total/3` is evaluated because the `total/3` expression is in parentheses, and `total/3` is `int/int` division resulting in truncation.
Solution 1:	`double average = (double) total / 3;`
Solution 2:	`double average = total / (double) 3;`
Solution 3:	`double average = total / 3.0;`

Solution 1 Explanation:

Remove the parentheses around the `total/3` expression to ensure the casting occurs on the `total` variable *only* rather than the `int` result of (`total/3`); this ensures that the division is properly occurring between an `double` and an `int`, which gives a `double`:

```
int total = num1 + num2 + num3;                  // Line 4
double average = (double) total / 3;             // Line 5
System.out.print("The average is " + average); // Line 6
```

Important! In this approach, the casting operator takes precedence over division, so `total` is casted first to a double *before* the division happens.

Solution 2 Explanation:

Move the (`double`) cast so that it casts the 3 in the denominator to a `double` value of 3.0, which has the same effect as the first approach since the division is occurring now between and `int` and a `double`, which gives a `double`:

```
int total = num1 + num2 + num3;                  // Line 4
double average = total / (double) 3;             // Line 5
System.out.print("The average is " + average); // Line 6
```

Solution 3 Explanation:

You also have *removed* the `(double)` casting operator in line 5, and changed the 3 to 3.0, thus ensuring `int / double` division; this has the same effect as casting 3 to a double as done in solution 2:

```
int total = num1 + num2 + num3;                  // Line 4
double average = total / 3.0;                    // Line 5
System.out.println("The average is " + average); // Line 6
```

Solution to CS Awesome problem (back):

Question 1 Solution

Original Code:

```
public class FirstClass
{
   public static void main(String[] args)
   {
       int a = 2;
       double b = (double) (5 / a);
       int c = b * 2;
       int d = (int) b + 0.5;

       System.out.println("a = " + a);
       System.out.println("b = " + b);
       System.out.println("c = " + c);
       System.out.println("d = " + d);
   }
}
```

Fixed Code (changes in red):

```
public class FirstClass
{
   public static void main(String[] args)
   {
       int a = 2;
       double b = (double) 5 / a;
       int c = (int) (b * 2);
       int d = (int) (b + 0.5);

       System.out.println("a = " + a);
       System.out.println("b = " + b);
       System.out.println("c = " + c);
       System.out.println("d = " + d);
   }
}
```

Note: The answers to the other CS Awesome questions are provided in the feedback when you choose correct or incorrect.

Answers (back)

PREDICT: What prints after each `println` statement?

```
double num1 = 4.8;
double num2 = 5.9;
System.out.println(num1 + num2);
System.out.println((int) num1 + num2);
System.out.println(num1 + (int) num2);
System.out.println((int) num1 + (int) num2);
System.out.println((int) (num1 + num2));
```

TYPE PREDICTIONS BELOW
(the first one is done for you)

Printed value	double or int?
10.7	double
9.9	double
9.8	double
9	int
10	int

Here is what gets printed when you run this code:

CONSOLE

```
10.7
9.9
9.8
9
10
```

Answers (back)

PREDICT: What prints after each `println` statement?

```
int num3 = 8;
int num4 = 9;
System.out.println(num3 + num4);
System.out.println((double) num3 + num4);
System.out.println(num3 + (double) num4);
System.out.println((double) num3 + (double) num4);
System.out.println((double) (num3 + num4));
```

TYPE PREDICTIONS BELOW

Printed value	double or int?
17	int
17.0	double
17.0	double
17.0	double
17.0	double

Here is what gets printed when you run this code:

CONSOLE

```
17
17.0
17.0
17.0
17.0
```

Solution (back)

Here is a code segment that declares and initializes several variables:

```
int a = 10;
int b = 15;
double y = 20.9;
double z = 25.4;

a = (int) y;
b = (int) z;

System.out.println("a = " + a);
System.out.println("b = " + b);
```

← **a)** Add ***two*** *lines* of code to the code segment on the left to store the truncated `int` values of y and z into a and b, respectively, using the `(int)` casting operator.

b) Test your solution by copying all the code into a program and running it; *you should see* (without decimals):

```
a = 20
b = 25
```

Hint (back)

It involves adding 0.5

Solution (back)

```
int roundedPrice = (int) (price + 0.5);
```

This adds 0.5 to price *first* since the addition is inside parentheses (thus taking 4.85 to 5.35), *then* performs the `(int)` casting operation which truncates 5.35 to 5 -- which is 4.85 rounded to the nearest integer!

Cool right?!

Hint (back)

It still involves 0.5, just not *adding* 0.5…

Solution (back)

Subtract 0.5 instead of adding 0.5 to round *negative* numbers!

```
int roundedTemp = (int) (coldTemp - 0.5);
```

We should get this on the console:

```
roundedTemp = -15
```

Sample solution (back)

Here are some changes that accomplish this (there may be other ways!)

BEFORE

```
int a = 2;
double b = (double) (5 / a);
int c = b * 2;
int d = (int) b + 0.5;

System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
```

AFTER

```
int a = 2;
double b = (double) 5 / a;
int c = (int) (b * 2);
int d = (int) (b + 0.5);

System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
```

What changed?

```
int a = 2;
double b = (double) 5 / a;
int c = (int) (b * 2);
int d = (int) (b + 0.5);

System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
```

Remove ( ) around 5 / a so the casting applies *just* to 5 (making it 5.0) rather than to the result (5 / 2) which would be 2 (because of int division) casted to 2.0. Recall that 5.0 / 2 = 2.5 , but 5 / 2 = 2 (and we want 2.5).

Add ( ) around b * 2, *then* cast result with (int) so that the multiplication between 2.5 and 2 occurs *first* to give 5.0, *then* 5.0 is cast to 5. Note that adding (int) next to b *without* enclosing b * 2 in ( ) would not be sufficient, since the casting would only apply to b, thus casting 2.5 to 2 before multiplying by 2 (which would incorrectly print 4)

Adding ( ) around b + 0.5 since we want to round b + 0.5 to the nearest int, and need the addition of 0.5 to occur *before* casting to an int.

Explanation (back)

Consider this *same* code segment, except in this case, let's **remove the casting from line 5:**

```
int num1 = -7;                                    // Line 1
int num2 = 2;                                     // Line 2
int num3 = 9;                                     // Line 3
int total = num1 + num2 + num3;                   // Line 4
double average = total / 3;                       // Line 5
System.out.println("The average is " + average); // Line 6
```

Copy, paste, run and see! What *was* the actual output? Were you correct?

```
The average is 1.0
```

What happens here is that `total` is an `int`, and assigned the value -7 + 2 + 9 = 4 in Line 4

Line 5 performs *integer division* between `total` (an `int`) and `3` (also an `int`). 4 / 3 = 1 (truncated)

Lastly, the value 1 gets converted automatically by Java to a `double` since `average` is declared as a `double`. So 1 → 1.0

Answer (back)

You can do this by declaring `total` to be a `double` rather than an `int`:

```
int num1 = -7;                                    // Line 1
int num2 = 2;                                     // Line 2
int num3 = 9;                                     // Line 3
double total = num1 + num2 + num3;                // Line 4
double average = total / 3;                       // Line 5
System.out.println("The average is " + average); // Line 6
```

Even though all the numbers are still integers, when they get summed up to 4, the 4 gets converted to 4.0 when assigned to `total` since `total` is a `double`.

Then in Line 5, when `total` is divided by 3 (an `int`), `total` is *already* a `double` 4.0 because it was declared that way in Line 4, so it isn't necessary to cast it to a double here. Thus, 4.0 / 3 is `double/int` which gives the correct `double` result of 1.3333333

Compare (back)

Now, **run** the code to see what actually happens! *Look at the output very closely…*
 	
Describe what happens! Is it what you expected to happen? Note the values of the two variables… did the math operations result in correct values?
	You should see this:
	`max int = 2147483647`
	`min int = -2147483648`
	`big pos num = 2147483600`
	`big neg num = -2147483600`
	`UPDATED big pos num = -2147483596`
	`UPDATED big neg num = 2147483596`
	
	Notably, the code **DID** run through to completion *without* syntax errors or exceptions (crashes).
	
	But the result for the two variables is **NOT** what you would expect! Notably, `bigPosNum` is now *negative* all of a sudden, and `bigNegNum` is now *positive*! **What?! Pure craziness.** Read on in the lab to find out why!

Sample solution (back)

Cast `num1` to a `double`:

```
Scanner scan = new Scanner(System.in);

System.out.print("Enter first integer: ");
int num1 = scan.nextInt();

System.out.print("Enter second integer: ");
int num2 = scan.nextInt();

double quotient = (double) num1 / num2;
int sum = num1 + num2;

System.out.println("The quotient is " + quotient);
System.out.println("The sum is " + sum);
```

Note that `num1` gets casted to a double *before* the division takes place.

OR cast `num2` to a `double`:

```
Scanner scan = new Scanner(System.in);

System.out.print("Enter first integer: ");
int num1 = scan.nextInt();

System.out.print("Enter second integer: ");
int num2 = scan.nextInt();

double quotient = num1 / (double) num2;
int sum = num1 + num2;

System.out.println("The quotient is " + quotient);
System.out.println("The sum is " + sum);
```

Similarly, note that `num2` gets casted to a double *before* the division takes place.

Sample solution (back)

Cast the *sum* of `num1` and `num2` as an `int`:

```
Scanner scan = new Scanner(System.in);

System.out.print("Enter first integer: ");
double num1 = scan.nextDouble();

System.out.print("Enter second integer: ");
double num2 = scan.nextDouble();

double quotient = num1 / num2;
int sum = (int) (num1 + num2);

System.out.println("The quotient is " + quotient);
System.out.println("The sum is " + sum);
```