# Chapter 3. Number Handling I – Numerical Primitives

This chapter offers extensive coverage of the topic of number handling in Java. Your knowledge of numerical types in Java and operations thereon is exercised and enhanced by the wide-ranging set of exercises that are presented in this chapter.

Key topics on which exercises are presented in this chapter include: knowledge of the primitive numerical types, appropriate selection of numerical types, computation exercises with built-in numerical operators as well as with the methods of the class `Math`.

Beyond the topics noted above, exercises on the usage of numerical constants from the numerical wrapper classes, overflow conditions, the `Exact` methods, infinity handling, hexadecimal numbers, number base conversion, logical and bitwise operations, increment, decrement and shorthand numerical operators are also included. A robust set of exercises on the formatting of numerical output is also presented.

The topic of number handling in Java is extensive and thus is not limited to the topics in this chapter; this chapter is the first of five chapters in this book on the topic of number handling in Java.

| | | |
|---|---|---|
| 1. | Explain the concept of numerical primitives in Java. | |
| | One way to describe a numerical primitive in Java is that a numerical primitive is a number type that is natively represented and handled in the computer CPUs that run Java. Most CPUs in the world today can represent and process the types `byte`, `short`, `int`, `long`, `float` and `double` natively. Also, numerical operations such as +, -, /, * on primitive types are native operations built into CPUs. | |
| 2. | List in ascending order of size, the different integer primitives available in Java. | In order of size: `byte`, `short`, `int` and `long`. |
| 3. | Explain the concept of numerical wrapper classes in Java. | |
| | In the first exercise, we explained that numerical primitives and some operations that can be performed on them are directly represented in most computer CPUs. However, not every operation that we would like performed is so represented. To address this, a corresponding class for each primitive exists in Java. These classes serve as an appropriate collection point for all methods and matters regarding the respective primitive that they correspond to. For example, the primitive `int` has a corresponding wrapper class named `Integer`, the primitive `short` has a wrapper class called `Short`, etc. Beyond serving as a collection point, there is also certain functionality in Java which works only on objects (for example *Collections* functionality, which we will see exercises on later); objects of these wrapper classes stand in for their equivalent primitive values in such cases.<br>In this chapter we will see the use of the wrapper classes to a degree; they and their interaction with primitives is covered more directly in the chapter entitled *Number Handing II – Numeric Wrapper Classes.* | |
| 4. | List the names of the corresponding wrapper classes respectively for each of the integer primitives `byte`, `short`, `int` and `long`. | `Byte`, `Short`, `Integer`, `Long` |
| **int** | | |
| 5. | What is the minimum value supported by the Java **`int`** type? | -2,147,483,648 |
| 6. | What is the maximum value supported by the Java `int` type? | 2,147,483,647 |
| 7. | What is the default value assigned to an `int` variable that you have not yet assigned a value to *(called an uninitialized variable)*? | There is no value assigned by the compiler. You yourself have to assign a value to a primitive variable before you can use it. |
| 8. | How many bits/bytes are used to represent an `int` in Java? | 32 bits/4 bytes |
| 9. | Write a line of code to show the declaration and initialization | `int firstInteger = 5;` |

| | | |
|---|---|---|
| | of an `int` variable named `firstInteger` with the value 5. | |
| 10. | Write a program, using the built-in constants in the class `Integer`, which writes out the minimum and maximum values supported by the `int` type; also print out the size in bits that an `int` occupies.<br>*Hint: constants* `Integer.MIN_VALUE`, `Integer.MAX_VALUE`, `Integer.SIZE` | |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  System.out.println(Integer.MIN_VALUE);`<br>`  System.out.println(Integer.MAX_VALUE);`<br>`  System.out.println(Integer.SIZE);`<br>`  }`<br>`}`<br>**Note:** As can be observed in this solution, the constants that refer to the primitive `int` are stored in its related wrapper class `Integer`. Recall that in the solution to exercise 3 it was stated that the corresponding wrapper class for each primitive type contains methods and other data (constants for example) that pertain to it and its corresponding primitive type.<br>It should be pointed out that the range of values supported by a primitive and its corresponding class are the same. However, the size of an `int` (which is represented by the constant `Integer.SIZE`) is not the size of an `Integer` object, that constant only refers to the size of an `int`.<br>The same logic above applies to all of the primitive types and their respective related classes. | |

| | | |
|---|---|---|
| **byte** | | |
| 11. | Describe the **`byte`** type. In particular describe the number of bits/bytes required to contain a `byte`, as well as the range of values that an `byte` can represent in Java. | |
| | A **`byte`** in Java is an integer type which occupies a single byte (8 bits) and thus can represent a range of $2^8 = 256$ numbers; the range of values that it represents is -128 to 127. | |
| 12. | Write a program which declares and initializes a byte variable named `firstByte` to the value -115 and then prints the variable to the console, stating "The value of firstByte is <*value*>". | |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  byte firstByte = -115;`<br>`  System.out.println("The value of firstByte is " + firstByte);`<br>`  }`<br>`}` | |
| 13. | What is the minimum value supported by the `byte` type? | -128 |
| 14. | What is the maximum value supported by the `byte` type? | 127 |
| 15. | Write a program, using the appropriate built-in constants in the class `Byte`, which writes out the minimum and maximum values supported by the `byte` type; also print out the size in bits that a `byte` occupies. | |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  System.out.println(Byte.MIN_VALUE);`<br>`  System.out.println(Byte.MAX_VALUE);`<br>`  System.out.println(Byte.SIZE);`<br>`  }`<br>`}` | |
| **short** | | |
| 16. | What is the minimum value supported by the short integer (**`short`**) type? | -32,768 |
| 17. | What is the maximum value supported by the **`short`** type? | 32,767 |
| 18. | How many bits/bytes are used to represent a `short`? | 16 bits/2 bytes |
| 19. | Write a program which declares and initializes a `short` variable named `firstShort` to the value 10,000 and then prints the variable to the console, stating "The value of firstShort is <*value*>". | |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {` | |

```
    short firstShort = 10000;
    System.out.println("The value of firstShort is " + firstShort);
  }
}
```

20. Write a program, using the built-in constants in the class `Short`, which writes out the minimum and maximum values supported by the `short` type; also, print out the size in bits that a `short` occupies.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.println(Short.MIN_VALUE);
    System.out.println(Short.MAX_VALUE);
    System.out.println(Short.SIZE);
  }
}
```

**long**

| | | |
|---|---|---|
| 21. | What is the minimum value supported by the Java long integer (**`long`**) type? | -9,223,372,036,854,775,808<br>(This is the value $-2^{63}$). |
| 22. | What is the maximum value supported by the Java `long` type? | 9,223,372,036,854,775,807 (This is $2^{63} - 1$) |
| 23. | How many bits/bytes are used to represent a `long`? | 32 bits/4 bytes |

24. Write a program which initializes a `long` variable named `firstLong` with the value 7,000,000,000 and then prints the variable to the console, stating "The value of firstLong is <*value*>".

```
public class PracticeYourJava {
  public static void main(String[] args) {
    long firstLong = 7000000000L;
    System.out.println("The value of firstLong is " + firstLong);
  }
}
```

25. Explain the reason for the postfix `L` at the end of the numerical value in the preceding exercise.

The reason for the postfix is because by default Java sees integer literals as values of type `int`; if the integer literal presented is greater than `Integer.MAX_VALUE` then Java sees it as an invalid value because in a sense it sees it as an integer that is out of bounds. However when it sees the postfix, Java evaluates the value as the type that the postfix implies.

26. What is the difference between the following declarations of the variable `firstLong`?

```
long firstLong = 7000000000L;
```
*and*
```
long firstLong = 7000_000_000L;
```
*and*
```
long firstLong = 7_000_000_000L;
```
*and*
```
long firstLong = 7_000_000___000L;
```

There is actually no difference between any of the declaration styles. Java allows you to put the underscore character in number literals as adds clarity for human viewers of the numerical value. The number of underscores does not matter either (see the 4th example). However, the underscore character cannot be put at the beginning or the end of the number.

27. Write a program, using the built-in constants in the class `Long`, which writes out the minimum and maximum values supported by the `long` type; also print out the size in bits that a `long` occupies.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.println(Long.MIN_VALUE);
    System.out.println(Long.MAX_VALUE);
    System.out.println(Long.SIZE);
  }
}
```

**floating point types (`float`, `double`)**

28. What is the minimum value supported by the **`float`** type?
    -3.4028235 x $10^{38}$

29. What is the maximum value supported by the `float` type?
    -3.4028235 x $10^{38}$

30. How many bits/bytes are used to represent a `float`?
    32 bits/4 bytes

31. What is the precision of a `float`?

    6-7 significant digits (that includes the digits both before and behind the decimal point).

    You know that integer values are discrete values with a deterministic integer distance of 1 between each point on the integer number scale. With floating points numbers it is not so; there are an infinite number of floating point numbers possible, even between two adjacent integers (for example between 0 and 1). Therefore it is impossible to write down all floating point numbers even within a small range, nor is it possible for a computer to represent the infinite number of points of even a small range on the floating point number scale. Therefore it has been decided that the `float` type will have a given maximum precision that it can represent. Any attempt to use the `float` type to represent a number with a precision higher than 6-7 digits will be unsuccesful.

    Even within the specified range of 6-7 digits, there are some numbers which cannot be represented exactly due to the way in which floating point numbers are stored in computers. Any decimal number that requires maintenance of precision (such as currency calculations) should only use the type `BigDecimal`.

    The reader is urged to investigate appropriate sources which describe how floating point numbers are represented in computers.

32. What is the smallest possible positive value that a `float` can represent?
    ~1.4 x $10^{-45}$
    Think of the number as the smallest step away from `0` that the Java `float` type can make.

33. Why is a `float` also described as a *single*?
    `float` really refers to what is called a "single precision floating point number", which is a floating point number format that is designed to occupy only 4 bytes. This nomenclature is an IEEE standard.

34. Write a program, using the built-in constants in the class `Float`, which writes out the minimum and maximum values supported by the `float` type. Also print out the size in bits that a `float` occupies.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.println("Mininum value = %f\n" + (-1 * Float.MAX_VALUE));
       //there is no constant for minimum value
    System.out.println("Maximum value = " + Float.MAX_VALUE);
    System.out.println("Bits occupied = " + Float.SIZE);
  }
}
```

35. Write a program which prints out the smallest positive `float` that Java supports.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.printf("Smallest positive float = %f", Float.MIN_VALUE);
  }
}
```

36. Why does java require that there a suffix/postfix **`F`** applied to the value assigned to a variable of type `float`?

    The reason is because Java presumes that any floating point literal entered is of type `double` *(double means "double precision floating point number", a type which we will see shortly)* unless the number is explicitly indicated to be a single precision number as indicated by the postfix **`F`**.

37. Write a program which initializes a single precision floating point variable named `firstFloat` with the value 7.919876542 and prints its value out to the console using `System.out.println`.
    Note and explain the difference between the initialized value and the value that is printed out.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float firstFloat = 7.919876542F; // Note the F at the end
    System.out.println(firstFloat);
  }
```

}

**Observation/Explanation:** The output is `7.9198766` not the 10 digit number `7.919876542` that was input! The reason is that the `float` type officially only has a precision of 6-7 significant digits and so we should not have attempted to assign it a value of higher precision. Yes indeed there are 8 significant digits in the output, however the last digit was the result of rounding. There are times when the value of the rounded up digit is acceptable and times when it is not.
Also note that the value that was input was rounded (not truncated). There are different rounding modes available in Java; exercises on these are presented in Chapter 12, entitled *Number Handling III*.

38. Write a program which will assign the value 1768000111.77 to a variable of type `float`. Print the value out using the `%f` specifier of the `printf` method. Note the output and present your observations on it.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float f1 = 1768000111.77F; // Note the F at the end
    System.out.printf("f1 = %f\n", f1);
  }
}
```

**Output:** `f1 = 1768000128.000000`
**Observations:** The output is quite different from the input! In fact, even if the input variable was manually rounded to the nearest decimal position, the value obtained would be be closer to the original number than what the computer has determined the float to be!
As in the preceding exercise, the error was that the number entered was beyond the stated precision of a float (which is 6-7 significant digits). Java did indeed retain the integrity of at least the first 6-7 digits.
**Conclusion(s):** Do not attempt to apply to a floating point variable a value beyond its stated precision.

**double**

| | | |
|---|---|---|
| 39. | What is the minimum value supported by the type `double`? | $-1.7976931348623157 \times 10^{308}$ |
| 40. | What is the maximum value supported by the type `double`? | $1.7976931348623157 \times 10^{308}$ |
| 41. | How many bits/bytes are used to represent a `double`? | 64 bits/8 bytes |
| 42. | What is the smallest possible positive value that a `double` can represent? | $4.9 \times 10^{-324}$ |
| 43. | What is the precision of a `double`? | 15-17 significant digits.<br>(contrast this with the precision of a `float`/single). |
| 44. | Write a line of code that shows the declaration and initialization of a double precision floating point variable named `firstDouble` with the value $5.4857990943 \times 10^{-4}$. | `double firstDouble = 5.4857990943E-4;` |

45. Write a program, using the built-in constants in the class `Double`, which writes out the minimum and maximum values supported by the `double` type. Also print out the size in bits that a `double` occupies.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.println("Mininum value = " + (-1 * Double.MAX_VALUE));
                                          //there is no direct constant for this
    System.out.println("Maximum value = " + Double.MAX_VALUE);
    System.out.println("Bits occupied = " + Double.SIZE);
  }
}
```

**Assessing the Precision of type `double`**

46. Write a program which will assign the 12 digit number 1768000111.77 to a variable of type `double` and print it out using the `%f` specifier of the `printf` method and also using the method `println`. Note the output and present your observations on it.
(This input value is the same attempted with a `float` in question 38).

```
public class PracticeYourJava {
```

```
  public static void main(String[] args) {
    double d1 = 1768000111.77;
    System.out.printf("d1 = %f\n", d1);
    System.out.println(d1);
  }
}
```

**Output:**
```
d1 = 1768000111.770000
d1 = 1.76800011177E9
```

**Observations:** The output matches the input, for the precision of the input value is 12 significant digits, which is within the supported range of a `double`. We observe that `println` output the number in exponential format.

47. Write a program which will assign the 16 digit value 17680001118938.77 to a variable of type `double` and print it out using the `%f` specifier of the `printf` method. Note the output and present your observations on it.

Solution: modify the code in the preceding exercise to use the number in this exercise.
**Output:** `d1 = 17680001118938.770000`
**Observations:** The output matches the input exactly, for the precision of the input value has 15-17 significant digits which is within the supported range of a `double`.

48. Write a program which will assign the 17 precision place value 176800011189387.35 to a variable of type `double` and print it out using the `%f` specifier of the `printf` method. Note the output and present your observations on it.

Solution: modify the code in the preceding exercise to use the number in this exercise.
**Output:** `d1 = 176800011189387.340000`
**Observations:** As can be seen, the last digit of the entered number is distorted. This should be expected as the `double` supports only at best a 15-17 precision place number.
**Summary:** You should carefully consider whether or not you should use the type `double` for values with precision greater than 15 places.

49. Write a program which will assign the 18 precision place value 1234567890123456780 to a variable of type `double` and print it out using the `%f` specifier of the `printf` method. Note the output and present your observations on it.

Solution: modify the code in the preceding exercise to use the number in this exercise.
**Output:** `d1 = 1234567890123456800.000000`
**Observations:** As can be seen, digit 18 is rounded up and that affects the value of digit 17. This should be expected as the type `double` supports at best only a 15-17 precision place number.

50. **IMPORTANT**
Print out the value `Float.MIN_VALUE`, using `printf` and then `println`. Comment on your observations.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double d1 = Double.MIN_VALUE;
    System.out.printf("d1 = %f\n", d1);
    System.out.println("d1 = " + d1);
  }
}
```

**Output**
```
d1 = 0.000000
d1 = 4.9E-324
```

Clearly the 1st output, using `printf` is wrong! This occurrence is because of the *default* way in which `printf` formats floating point numbers; it will by default print at most 6 digits after the decimal point and will not by default attempt to print the value out in exponential format as `println` does.

Later in this chapter, exercises on formatting of numerical output are presented.

For further experimentation along this line, print out the `double` value -1.0011596521859 using `printf` and `println`.

51. **IMPORTANT**
Print out the value to at least 15 places of the result of the computation 1/10. Comment on the output.

```
public class PracticeYourJava {
```

```
  public static void main(String[] args) {
    double x = 1.0f/10.0f;
    System.out.printf("%.15f\n", x);
  }
}
```

**Output**

```
z = 0.100000001490116
```

The value is not exactly `0.1`! The computation 1/10 is supposed to yield a very specific rational result. This error occurs in some floating point calculations because of the way that computers handle/store floating point numbers. Clearly, this is not a good thing when we need exact precision; imagine scenarios where we have a financial application which computes values ranging into the billions of currency units; this error will definitely show up in such sensitive computations. Another example where precision is required is in scientific calculations; this imprecision in handling would also show up then. We see in the next section and in the chapter *Number Handling III* how to achieve the desired precision for floating point numbers by using the type `BigDecimal`.

**The numerical types `BigInteger` & `BigDecimal`: A brief note**

**`BigInteger`**: What do you use to represent an integer value whose value is larger that the type `long` or even longer than an unsigned long (we see the unsigned long type in a later chapter)? The type `BigInteger`. This type can accept and operate on arbitrarily large integer values.

**`BigDecimal:`** In our exercises with floating point values, you've observed that the exact value of floating point numbers is not always correct, this due to the way that floating numbers are represented in computers. However, Java presents a different type, the type `BigDecimal`, which represents floating point numbers with fidelity. Numbers represented using the type `BigDecimal` are not represented in the same internal format that `float` and `double` are. Due to its fidelity, for any financial computation that involves decimal values, or for any decimal value whose fidelity we utterly must preserve, the **only** acceptable choice in Java for such numbers is the type `BigDecimal`.

We have presented these two types here briefly as we refer to their use in a few upcoming exercises. They are covered directly in the chapter entitled *Number Handing III – Big Integer, Big Decimal.*

| | |
|---|---|
| 52. | I have the integer value 424,312,343,211,431,231,234,312,431,234,122,343,123,131,312,313,123,436. What type is best to represent it in Java? |
| | The type `BigInteger`. This type handles arbitrarily long integer values. We see this type in a later chapter. |

**Numerical suffixes**

| | | |
|---|---|---|
| 53. | What are the suffixes to apply to numerical values to indicate what type they are? In particular list the suffixes for each of, `long`, `float` and `double`. | `long` `L`<br>`float` (also known as single) `F`<br>`double` `D`<br>**Note:** These suffixes are case insensitive. |

**Decide which number type best represents the kind of number indicated in the question.**

| | | |
|---|---|---|
| 54. | Which type would be best to represent the salary of an individual? | The type `BigDecimal`. The reasons is because monetary values are always best represented by variables of type `BigDecimal` as `BigDecimal` represents decimal numbers accurately. |
| 55. | Which type would be best to represent the aggregate salary over 20 years of the salaries of a company which has 50,000 employees? | The type `BigDecimal` would be best because we are representing a monetary value which requires precise representation. |
| 56. | Which type is sufficient to handle the age of a person? | The type `byte` is sufficient and safe (unless you are interested in fractions of years as well). The type `byte` has |

| | | |
|---|---|---|
| | | a maximum of 127 which is about as high as people live these days. |
| 57. | The population of the world is currently ~7.2 billion people. Which type should be used to represent this? | The type `long` is an appropriate type in this case. The reasons for this are:<br>1. Human beings are integral units.<br>2. This is the closest type that can contain at least that number of people. |
| 58. | To count the number of siblings you have, which type would be the most likely candidate? | A `byte` will do. The reasons for this are:<br>1. Human beings are integral units.<br>2. Chances are no one has up to 127 siblings! |
| 59. | Which type is best to represent the Planck constant which has a value of $6.626\ 068\ 9633 \times 10^{-34}$ ? | The type `BigDecimal` because although this number has only 11 significant digits which can be represented in a `double` we would be safe with the precision of the `BigDecimal` type when using this constant in calculations. |
| 60. | The speed of light is precisely defined as 299 792 458 m/s. Which type would be sufficient to represent it? | An `int`, because this number is an integer of value approximately ~300 million which is less than the ~2.1 billion that an `int` supports. |
| 61. | The Rydberg constant has a value of $1.0973731568539 \times 10^{7}$. Propose which type is best to represent it. | The type `BigDecimal` because although this number has only 14 significant digits, we would be safe with the precision of the `BigDecimal` type when using this constant. |
| 62. | The electron magnetic moment to Bohr magneton ratio has a value of -1.001 159 652 1859. Which type best represents it? | The type `BigDecimal`. |
| 63. | The electron g factor is -2.002 319 304 3718. Which type best represents it? | The type `BigDecimal`. |
| 64. | Electron g factor uncertainty is 0.000 000 000 0075. Which type best represents it? | The type `BigDecimal`. |
| 65. | The value of Coulomb's constant is $8.9875517873681764 \times 10^{9}$. Which type best represents it? | A `BigDecimal`. Not only for its accuracy, but also because this number has 17 significant digits. |
| A general note on the choice of `BigDecimal` for exercises 59 to 64: Really, a `double` may suffice in these cases for general computation scenarios (perhaps, for example, in writing examinations). | | |
| **Casting** | | |
| 66. | Explain the concept of *casting*. | |
| | Casting is a language feature wherewith you can force the compiler to take a copy of the contents of a variable and convert the copy to the format of another type.<br>For example, I can do the following:<br>`int a = 50;`<br>`short b = (short)a;`<br>The second line of which means, *give me a copy of* `a` *that has been converted into the internal format and space of a short and put it into the short variable* `b` *(note that* `a` *itself is not changed).*<br>We elucidate further using `short` and `int`; a `short` is represented in 2 bytes and an `int` in 4 bytes. It is reasonable to say that any `int` value which is less than or equal to the width of a `short` (i.e. an `int` whose actual value is between -32,768 and 32,767) can legally fit into a `short`. However, we have to explicitly inform the compiler that we do indeed want to copy the contents of the `int` in question into a `short`.<br>Note that you can actually still cast the value of a higher width type whose contents are greater than the width of a lower width type into the lower width type; for example you can cast an `int` whose value is greater than 32,767 into a `short`; while indeed the value in question cannot actually fit into the space of a `short`, the compiler will still do its best to cast the value, however the value will be distorted, i.e. wrong, since you cannot fit something of a higher actual width into a smaller space properly. | |

Casting also pertains to objects; you can cast an object of a given class into a variable of any of its ancestor classes. This ability is facilitated by the concept of polymorphism. We see examples of this in the chapter on classes.

The concepts of *boxing* and *unboxing* can be seen as a special type of casting. We address the topic of boxing and unboxing in a later chapter.

67. Why would you want to cast variables?

You would want to cast variables in scenarios where you want the contents of a cast*ed* variable to be put into a variable of a different type.

For example, if I have an `int a=10` and given how small its value is, I want to put it into a variable of type `byte` (which supports values from 0 to 255), I would cast it as follows:

```
byte c=(byte)a;
```

This line is saying *take a copy of the contents of* `int a` *and stuff them into* the format of a `byte` *as best as you can.*

If for example I have to add an `int` and a `long` and want the resultant value to be put into an `int`, due to the fact that the target variable has a smaller width than at least one of the input variables, the input variables that are larger than the output variable have to be cast to the width of the target variable. See the following example:

```
long a=10; int b=50;
int result = (int)a + b;
```

Some casts are more dramatic. For example, the `char` type (which represents characters) can be cast into an integer representation. For example:

```
char letter = 'A';
int integerEquivalent = (int)letter;
```

With this cast, you are now able to facilitate the sorting of characters because they have an explicit numerical representation, which lends itself to ordering by magnitude.

You can also cast objects. For example, if I have the following object of a class named `Road`:

```
Road x = new Road();
```

I can, if I desire, cast the object `x` to an object of the ultimate ancestor class `Object` as follows:

```
Object y = (Object)x;
```

(The casting of an object to any of its ancestor classes is made feasible by the concept of polymorphism)

I can cast `y` back to an object of class `Road` as follows:

```
Road z = (Road)y;
```

As previously stated, for numerical values you *will* lose data in casting if you try to cast a variable which contains a higher value than what the type of the target variable can support. For example if we have the following scenario:

```
int var01 = 75000;
short var02 = (short)var01;
```

Because the value of the variable `var01` exceeds the maximum for a `short`, `var02` definitely will not contain the value 75,000, but rather a distorted value.

68. What is implicit casting?

Implicit casting is a scenario where, given the operation at hand, the compiler sees it fit to silently cast the contents of a variable of a given type to a more appropriate type that it is safe to cast the given type to.

For example, if I wanted to add an `int` to a `long` and put the result into a `long`, the code would look like the following:

```
long a=10; int b = 50;
long result = a + b;
```

In order to perform this computation, the compiler has to make the contents of both `a` and `b` the same type; in this case the compiler will simply silently cast `b` to the wider type `long` (note that it doesn't change `b` itself it just

takes a *casted* value of `b`) and adds it to `a` to obtain `result`.

There will not be any ill-effects since an `int` is of smaller range than the target output variable (a `long` in this case).

Now, if I had the following scenario where `result` is an `int`:

```
long a=10; int b=50;
int result = a + b;
```

The compiler will not implicitly cast the variable `a` to an `int`. Why not? Because casting a given type (`long` in this case) to a smaller type can potentially result in the loss of data. Therefore, in this case, the compiler will insist that you do the casting of the variable `a` yourself if that is what you really want, requiring you to explicitly write the following:

```
int result = (int)a + b;
```

69. Explain the result of compiling and running the program below.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    short a = 5;
    short b = 2;
    short result;

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

It will *not* compile. The reason for the non-compilation is that in Java, computations done with respect to `short` or `int` values result in an `int`. Therefore Java holds the value of `a + b` as an `int`, even though both input values are of type `short`. Us trying to put the resulting `int` value into a `short` is illegal unless we explicitly cast the value.

To get this to compile, you have to do either of the following:

(1) `result` should be declared as an `int`

*or*

(2) `result = (short)(a+b);` that is, you have to cast the result of the computation to the desired type (in this case, a `short`).

70. Write a program to add the following two `short` values (`i=250; j=3`). Put the result into a `short` and print the result out to the console.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    short i = 250;
    short j = 3;
    short result;

    result = (short)(i + j);
    System.out.println("result = " + result);
  }
}
```

71. Write a program which multiplies the following two `short` values (`i=250; j=300`), puts the result into a `short` and prints the result out to the console.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    short i = 250;
    short j = 300;
    short result;

    result =(short)(i * j);
    System.out.println("result = " + result);
  }
}
```

**Note:** This will print out a value of `550` instead of `75000`! The reason is because the resultant value is greater than `Short.Max` (which = `32767`) and thus the result of casting a 4 byte integer value of `75000` to a `short` (which has a

width of 2 bytes) yields `550` which is not want we want at all. We have two options to avoid such behavior. These are:

1. Always use the appropriately sized type for your calculations.
   *or*
2. Use the ***`Exact`*** methods of class `Math` to ensure that an exception is triggered when a computation goes out of bounds. We will look at the ***`Exact`*** methods later in this chapter.

72. Write a program to multiply the following two `short` variables (`i=250; j=300`). Put the result into an `int` and print the result out to the console.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    short i = 250;
    short j = 300;
    int result;

    result = i * j;
    System.out.println("result = " + result);
  }
}
```

**constants**

73. What is a constant (**`final`**) in Java?

A constant is a variable with a hardcoded and immutable value. For example, in the class `Math`, there are the constants `PI` and `E`.

You define constants using the keyword **`final`** (indicating that this is the "final value" of this variable). A variable labeled as being `final` can only be assigned to once! That assignment might be hardcoded in the code itself, or for constants in individual objects or in methods it can be at any time during the running of the code, only that it can only be done only once.

Again, a constant can be declared at the class level, or at the instance level, or inside a method. The declaration for a class level constant has the following structure:

```
class <class name>
{
   <access level> static final <field type> <field name> = <value>;
}
```

In the class `Math` for example, the constants `PI` and `E` are class level constants and are declared as:

```
class Math
{
  public static final double PI = 3.141592653589793; // See the use of the keyword final
  public static final double E =  2.718281828459045;
}
```

(The keyword `static` is what makes these constants *class constants*, thus making them directly accessible by any object that can access the fields of this class; for example you can simply access the field `PI` in any of your code directly as `Math.PI`. We will look closer at the keyword `static` the chapter entitled *Classes I*).

The declaration for an instance/object level constant is as follows:

```
class <class name>
{
   <access level> final <field type> <field name> = <value>; // No keyword static
}
```

Method constants are declared within methods in the following manner (using `main` as an example):

```
public static void main(String[] args){
  final double ten       = 10.0;
  final int DaysInWeek   = 7;
  final int MonthsInYear = 12;
  final String MetricOrImperial; // We can assign to this later in our code, perhaps it is
                                 // a field we get from the user.
 }
```

**Accessing constants**
Class constants are accessed using the class name as the prefix to the constant name. Therefore, for example, you access the constant `PI` of the class `Math` as `Math.PI` in your code.
Instance constants are accessed using the name of the object as the prefix to the constant name.
Method constants are accessed using only their name (since a method constant is only accessed from within the method in which it is declared).

74. What is a blank final?

A blank final is a `final` variable that is not assigned at compile time.

75. Write a program to print out the value of the built-in mathematical constant π.
*Hint: `Math.PI`*

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.println(Math.PI);
  }
}
```

76. Write a program to print out the value of the built-in mathematical constant **e**.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    System.out.println(Math.E);
  }
}
```

77. Write a line of code to show the implementation of the Rydberg constant as a `public` *class* constant.

```
public static final double rydberg = 1.0973731568539e7;
```

78. Write a line of code to show the implementation of the Rydberg constant as a `public` *instance* constant.

```
public final double rydberg = 1.0973731568539e7;
```

**Built-in mathematical operators and methods**
In this section, exercises on the usage in Java of standard mathematical operators, constants and methods of the class `Math` are presented.

79. Write a program which adds the following two `int` variables; `i=5`; `j=9`; and prints the result out to the console.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    int i = 5; int j = 9;
    int result = i + j;
    System.out.println("The sum is: " + result);
  }
}
```

80. Given two `int` variables `i=5`; `j=9`; write a program which subtracts the latter from the former and prints the result out to the console.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    int i = 5; int j = 9;
    int result = i - j;
    System.out.println("The sum is: " + result);
  }
}
```

81. Write a program to calculate and print out the remainder of 55 divided by 22.
*Hint: Use the `%` operator*

```
public class PracticeYourJava {
  public static void main(String[] args) {
    int rem = 55 % 22;
    System.out.println("The remainder is: " + rem);
  }
}
```

82. Write a program to calculate and print out the square root of 81.39.
*Hint: `Math.sqrt`*

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float x = 81.39f;
    double result;

    result = Math.sqrt(x);
    System.out.println("The square root of " + x + " is: " + result);
  }
}
```

83. Print out the value of 79.34 raised to the power of 12.

*Hint:* `Math.pow`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float x = 79.34f;
    double result;

    result = Math.pow(x, 12);
    System.out.println(x + " to the power of " + 12 + " is: " + result);
  }
}
```

84. Print out the cube root of 89.
*Hint:* `Math.cbrt`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float x = 89.0f;
    double result;

    result = Math.cbrt(89);
    System.out.println("The cube root of " + x + " is: " + result);
  }
}
```

85. Print out the fifth root of 89.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float x = 89.0f;
    double result;

    result = Math.pow(x,(1.0/5.0));
    System.out.println("The 5th root of " + x + " is: " + result);
  }
}
```

86. Print out the $7^{th}$ root of 121.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float x = 121.0f;
    double result;

    result = Math.pow(x,(1.0/7.0));
    System.out.println("The 7th root of " + x + " is: " + result);
  }
}
```

87. Write a program to calculate and print out the cosine of $180^{o}$.
*Hint: Methods* `Math.Cos` *and* `Math.toRadians`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double degrees = 180;
    double radians = Math.toRadians(degrees);
    double cosine = Math.cos(radians);
    System.out.printf("The cosine of %f degrees = %f\n",degrees,cosine);
  }
}
```

88. Write a program to calculate and print out the tangent of $30^{o}$.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double degrees = 180;
    double radians = Math.toRadians(degrees);
    double tangent = Math.tangent(radians);
    System.out.printf("The tangent of %f degrees = %f\n",degrees,tangent);
  }
}
```

89. Write a program to calculate and print out the tangent of 60°.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double degrees = 60;
    double radians = Math.toRadians(degrees);
    double tangent = Math.tangent(radians);
    System.out.printf("The tangent of %f degrees = %f\n",degrees,tangent);
  }
}
```

90. Write a program to calculate and print out the sine of 75°.
*Hint:* `Math.toRadians`, `Math.sin`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double degrees = 75;
    double radians = Math.toRadians(degrees);
    double sine = Math.sin(radians);
    System.out.printf("The sine of %f degrees = %f\n",degrees,sine);
  }
}
```

91. The sine of a given angle is .8660254037. Write a program to calculate and print out in degrees *and* in radians the angle that this value corresponds to.
*Hint:* `Math.asin`, `Math.toDegrees`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double sine = .8660254037;
    double degrees, radians;

    radians  = Math.asin(sine);
    degrees  = Math.toDegrees(radians);
    System.out.printf("%f is the sine of %f degrees (or %f radians).\n", sine, degrees, radians);
  }
}
```

92. The cosine of a given angle is .8660254037. Write a program to calculate and print out in degrees and in radians the angle that this value corresponds to.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double cosine = .8660254037;
    double degrees, radians;

    radians  = Math.acos(cosine);
    degrees  = Math.toDegrees(radians);
    System.out.printf("%f is the cosine of %f degrees (or %f radians).\n", cosine, degrees,
radians);
  }
}
```

93. The tangent of a given angle is .8660254037. Write a program to calculate and print out in degrees and in radians the angle that this value corresponds to.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double tangent = .8660254037;
    double degrees, radians;

    radians  = Math.atan(tangent);
    degrees  = Math.toDegrees(radians);
    System.out.printf("%f is the tangent of %f degrees (or %f radians).\n", tangent, degrees,
                      radians);
  }
}
```

94. Write a program to print out the area of a circle of radius 7.

```
public class PracticeYourJava {
```

```
  public static void main(String[] args) {
    double radius = 7.0;
    double area;

    area = Math.PI * Math.pow(7,2);
    System.out.printf("The area of a circle of radius %f = %f.\n", radius, area);
  }
}
```

95. Write a program to print out the absolute value of the following values: -743.8, 525.3 and 1501.
*Hint:* `Math.abs`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double value01 = -743.8; double abs01;
    double value02 = 525.3;  double abs02;
    int value03    = 1501;   int abs03;

    abs01 = Math.abs(value01);
    abs02 = Math.abs(value02);
    abs03 = Math.abs(value03);

    System.out.printf("The absolute value of %f is %f\n", value01, abs01);
    System.out.printf("The absolute value of %f is %f\n", value02, abs02);
    System.out.printf("The absolute value of %d is %d\n", value03, abs03);
  }
}
```

96. Calculate the logarithm to base 10 of 7 and the logarithm to base 10 of 22. Add the resulting values, putting the sum thereof into a variable `x`. Then compute and print out the value of $10^x$.
*Hint:* `Math.Log10, Math.pow`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double d01 = 7;
    double d02 = 22;

    double logOfd01 = Math.log10(d01);
    double logOfd02 = Math.log10(d02);

    double x = logOfd01 + logOfd02;
    double result = Math.pow(10, x);
    System.out.printf("result = %f\n", result);
  }
}
```

97. Calculate the logarithm to base $e$ of 7 and the logarithm to base $e$ of 22. Add the resulting values, putting the sum thereof into a variable `x`. Then compute and print out the value of $e^x$.
*Hint:* `Math.Log, Math.exp`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double d01 = 7;
    double d02 = 22;

    double logOfd01 = Math.log(d01);
    double logOfd02 = Math.log(d02);

    double x = logOfd01 + logOfd02;
    double result = Math.exp(x);
    System.out.printf("result = %f\n", result);
  }
}
```

98. Pythagoras' theorem states that for a given right-angled triangle with sides $a$, $b$ and $c$ where $c$ is the hypotenuse, if you have the values $a$ and $b$, the value for $c$ can be determined as follows:
$c = \sqrt{(a^2 + b^2)}$.
So given a triangle where $a$=4 and $b$=3, write a program to calculate `c` and print it out.
*Hint:* `Math.hypot`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    float a = 3;
    float b = 4;
    double c; // this is for the hypotenuse

    c = Math.hypot(a, b);
    System.out.printf("hypotenuse = %f\n", c);
  }
}
```

99. Given two numbers, 55 and 70, write a program to determine and print out which is the larger of the two.

*Hint:* `Math.max`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int bigger = Math.max(55, 70);
    System.out.printf("The bigger number is %f\n", bigger);
  }
}
```

100. Given two numbers, 320 and -2.95 of type `double`, write a program to determine and print out which is the smaller of the two numbers.

*Hint:* `Math.min`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    double smaller = Math.min(320, -2.95);
    System.out.printf("The smaller number is %f\n", smaller);
  }
}
```

**Note:** to do comparisons using `Math.min` or `Math.max` the numbers being compared must be of the same type.

101. Round the value 12777.899 to the nearest whole number `double` and print the result out.

*Hint:* `Math.rint`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    double x = 12777.899;
    double rounded = Math.rint(x);
    System.out.printf("The rounded value is %f\n", rounded);
  }
}
```

102. Round the value 12777.899 to the nearest `long` and print the result out.

*Hint:* `Math.round`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    double x = 12777.899;
    long rounded = Math.round(x);
    System.out.printf("The rounded value is %d\n", rounded);
  }
}
```

103. Write a program to print out the ceiling of 5.799 and 583.6 respectively.

*Hint:* `Math.ceil`

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double d01 = 5.799;
    double d02 = 583.6;

    double ceil01 = Math.ceil(d01);
    double ceil02 = Math.ceil(d02);

    System.out.printf("The ceiling of %f is %f\n",d01,ceil01);
    System.out.printf("The ceiling of %f is %f\n",d02,ceil02);
  }
}
```

104. Write a program to print out the floor of 5.799 and 583.6 respectively.

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double d01 = 5.799;
    double d02 = 583.6;

    double ceil01 = Math.floor(d01);
    double ceil02 = Math.floor(d02);

    System.out.printf("The floor of %f is %f\n",d01,ceil01);
    System.out.printf("The floor of %f is %f\n",d02,ceil02);
  }
}
```

105. Give the signum of each of the following `double` values:

1. -55.6
2. 0.0004
3. 0
4. 79

```
public class PracticeYourJava {
  public static void main(String[] args) {
    double num01=-55.6, num02=0.0004, num03=0.0;
    int  num04=79;

    double signum01 = Math.signum(num01);
    double signum02 = Math.signum(num02);
    double signum03 = Math.signum(num03);
    double signum04 = Math.signum(num04);
```

```
    System.out.printf("signum of %f is %f\n", num01, signum01);
    System.out.printf("signum of %f is %f\n", num02, signum02);
    System.out.printf("signum of %f is %f\n", num03, signum03);
    System.out.printf("signum of %d is %f\n", num04, signum04);
  }
}
```

106. I have two double variables `x` and `y`. Write a program that returns a number with the magnitude of `x` but the sign of `y`.
Test the program with the following pairs *(x, y)* of values:

- -89.5, 200
- 400, -300

*Hint:* `Math.copySign`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    double set1_num1=-89.5, set1_num2=200;
    double set2_num1=400,   set2_num2=-300;

    double result1 = Math.copySign(set1_num1, set1_num2);
    double result2 = Math.copySign(set2_num1, set2_num2);

    System.out.printf("First pair result = %f \n", result1);
    System.out.printf("First pair result = %f \n", result2);
  }
}
```

107. **IMPORTANT:** Print out the result of the division of the following calculation: `a/b`, where `a` and `b` are both integers of value 1 and 4 respectively. Put the result into a `float`.
Explain the result.

An initial attempt at writing the code might yield the following:

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a=1, b=4;
    float result = a/b;  //  WRONG !!!

    System.out.println(result);
  }
}
```

**Explanation**: The result of this code is `0`! The issue here is that on seeing integer values (1 and 4 in this case), Java automatically performs an integer calculation (irrespective of the type of the target output variable). If you want the correct results, then ensure that you cast the input variables appropriately. In this case we would write the following calculation for `result`:

```
float result = (float)a/(float)b;
```

108. What is the expected output of this code fragment?

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = 5;
    int b = 0;
    System.out.printf("%d/%d=%f",a,b,(a/b));
  }
}
```

The program will stop running and it will throw the exception '`java.lang.ArithmeticException`' for this integer divide by zero situation.

**Infinity/-Infinity/NaN**

109. Write a program which determines and prints out the result of the following computations given the noted variable values:

```
float a=0, b=0, c=-1, d=1;
```

1. `a/b`
2. `c/a`
3. `d/a`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    float a = 0, b=0, c=-1, d=1;
    System.out.printf("%f/%f = %f\n", a, b, a / b);
    System.out.printf("%f/%f = %f\n", c, a, c / a);
    System.out.printf("%f/%f = %f\n", d, a, d / a);
  }
}
```

**Results**

1. `0/0  = NaN` (which means Not a Number)
2. `-1/0 = -Infinity`
3. `1/0  = Infinity`

110. Write a program which determines and prints out the result of each of the following calculations:

1. -∞ + ∞
2. ∞ / ∞
3. -∞ / ∞
4. ∞ / 0
5. -∞ / 0

*Hint: The constants* `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    float a = Float.POSITIVE_INFINITY;
    float b = Float.NEGATIVE_INFINITY;
    System.out.printf("%f+%f = %f\n", b, a, b + a);
    System.out.printf("%f/%f = %f\n", b, a, b / a);
    System.out.printf("%f/%f = %f\n", b, b, b / b);
    System.out.println(a+ "/0 = " + (a/0));
    System.out.println(b+ "/0 = " + (b/0));
    //System.out.printf("%f/%f=%f\n", a, 0, a / 0);
    //System.out.printf("%f/%f=%f\n", b, 0, b / 0);
  }
}
```

**Results**

```
1.  -Infinity +  Infinity   =  NaN
2.  -Infinity /  Infinity   =  NaN
3.  -Infinity / -Infinity   =  NaN
4.  Infinity  / 0           =  Infinity
5.  -Infinity / 0           = -Infinity
```

**Complex Numbers**

111. We have the complex number 10 + 5i, with each of its numerical values expressed in a `double` variable. Express it in polar form, i.e. (`r`, `θ`).

*Hint:* `Math.hypot`, `Math.atan2`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    double x_var1 = 5;
    double y_var1 = 10;

    double r = Math.hypot(x_var1, y_var1);
    double theta = Math.atan2(x_var1, y_var1);
    System.out.printf("The complex number in polar form is
                       (%f, %f)\n", r, theta);
  }
}
```

**Increment, decrement and shorthand operators**

112. Given the following initialized variable:
`int i=2;`
State what its value is after the statement:
`i++;`

`3`
The increment operator(`++`) simply means that the value of the variable in question should be incremented by **`1`**.

113. Given the following variable:
`double i=8.46;`
State what its value is after the statement:
`i++;`

`9.46`
The increment operator also works on floating point numbers, incrementing the value of the variable in question by **`1`**.

114. Given the following initialized variable:
`int i=2;`
State what its value is after the statement:
`++i;`

`3`
It does not matter for a variable in a standalone statement like this whether the increment operator appears before or after the variable. The same logic applies to the decrement operator.

115. Given the following variable: `int i=26;`
State what its value is after the statement:
`i--;`

`25`
The decrement operator(`--`) simply means that the value of the variable in question should be decremented by **`1`**.

116. Given the following variable: `int i=23;`
State what the values of `j` and `i` are after the following statement:
`int j=++i;`

This statement is saying `j = incremented value of i`
Thus `i` is incremented first *before* assigning the incremented value of `i` to `j`. Thus the result is:
`j=24`
`i=24`

117. Given the following variable:
`int i=23;`
State what the values of `j` and `i` are after the following statement:
`int j=i++;`

`j=23`
`i=24`
The statement `j=i++;` means the following:
assign `i` to `j` and then afterwards increment `i`
Thus `j` is assigned the current value of `i`, that is `23` and then `i` is

| | | |
|---|---|---|
| | | incremented to 24. |
| 118. | Given the following code snippet:<br>`int i=5; j=27;`<br>`int result = i++ + --j;`<br>1. What is the expected value of `i`?<br>2. What is the expected value of `j`?<br>3. What is the expected value of `result`? | 1. `i = 6`<br>2. `j = 26`<br>3. `result = 5+26 = 31`<br>In words we can say that the line of code means the following:<br>`result=`<br>`(i, (then increment i), PLUS (decremented value of j))` |
| 119. | Given the following code snippet:<br>`int i=5; j=27;`<br>`result = --i - --j;`<br>1. What is the expected value of `i`?<br>2. What is the expected value of `j`?<br>3. What is the expected value of `result`? | 1. `i=4`<br>2. `j=26`<br>3. `result= 4-26 = -22`<br>In words we can say:<br>`result = (decrement i MINUS decrement j)` |
| **Shorthand operators** | | |
| 120. | Given two numerical variables `g` and `y`, what does the following statement mean?<br>`g+=y;` | This means *increment g by y*, in short, `g = g+y;` |
| 121. | Given two numerical variables `g` and `y`, what does the following statement mean?<br>`g*=y;` | This means *multiply g by y*, in short it means exactly the same thing as:<br>`g = g*y;` |
| 122. | Given two numerical variables `g` and `y`, what does the following statement mean?<br>`g/=y;` | This means *divide g by y*, in short, `g = g/y;` |
| 123. | Given two numerical variables `g` and `y`, what does the following statement mean?<br>`g-=y;` | This means *decrement g by y*, in short, `g = g-y;` |
| 124. | Given the following variable: `int i=12`, state the value of `i` after the statement<br>`i+=29;` | `i=41`<br>The given statement is a shorthand way of saying: `i = i+29;` (i.e. *add 29 to* `i`) |
| 125. | Given the following variable: `int i=39`, state the value of `i` after the statement<br>`i/=3;` | `i=13`<br>The given statement is a shorthand way of saying: `i = i/3;` (i.e. *divide* `i` *by 3*) |
| 126. | Given the following variable: `int j=9`, state the value of `j` after the statement<br>`j*=6;` | `j=54`<br>The given statement is a shorthand way of saying: `j = j*6;` (i.e. *multiply* `j` *by 6*) |
| 127. | Given the following variable: `float x=5.2f`, state the value of `x` after the following statement: `x-=2;` | `3.2`<br>The given statement is a shorthand way of saying: `x = x-2;` (i.e. *decrement* `x` *by 2*) |
| 128. | What is the difference between the following two statements?<br>`i+=1;`<br>`i++;` | There is no difference; both mean the same thing, which is *increment i by 1*. |

| | |
|---|---|
| 129. | Describe the concept of integer overflow. |
| | The concept of integer overflow can be described as follows.<br>Using the type `int` as our example, imagine that all its values are represented on a wheel, starting from the lowest integer to the highest integer.<br>Now, say we have `int x=Integer.MAX_VALUE` (largest integer). if we add the value of 1 to `x`, i.e. `x=x+1`, then interestingly what happens is simply that `x` is moved to the next point on the wheel, which is `Integer.MIN_VALUE`! |

If I added 2 to the value of `x`, then `x` would be moved to the position `Integer.MIN_VALUE - 1`.

This is how integer overflow occurs in Java.

Integer overflow should be assumed to be a risk for any integer computation.
Note: Even though we used the `int` type as an example, this concept applies to any integer type.

130. Describe the concept of integer underflow.

Integer underflow happens when the result of an integer computation results in a value that is less than (i.e. closer to -∞) the minimum value that the integer type in question can contain. Continuing with the previous analogy of `int` values on a wheel, imagine we have `x= Integer.MIN_VALUE`. If we subtract the value of 1 from `x`, i.e. `x=x-1`, we are moving backwards on the wheel and the result is `x= Integer.MAX_VALUE`, which is clearly incorrect!

131. How do I guard against the integer overflow/underflow issue?

Use the "`Exact`" set of methods *(introduced in Java 8)* of the class `Math` instead of the regular integer operators.

132. What is the result if a computation overflows or underflows when we use an "`Exact`" method?

An exception will be generated by the program. If the exception is not explicitly caught by the program, the program will stop running. We will look at exceptions in a later chapter.

133. I have two `int` variables `a` and `b`, the addition of which has a risk of overflowing and I want to catch this overflow. Write a sample program to show how this would be done.
State what the output of this program is. Use a value of `Integer.MAX_VALUE` for both variables `a` and `b`.
*Hint:* `Math.addExact`

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MAX_VALUE; //just a value for the purposes of this program
    int b = Integer.MAX_VALUE; //as above
    int total;

    total = Math.addExact(a, b);
    System.out.println(total);
  }
}
```

**Note(s):**

1. Running this code for the choice of numbers herein will result in an overflow, which causes the throwing of the exception `java.lang.ArithmeticException`. Exceptions are discussed in a later chapter.
2. Note that if you did not use the `Math.addExact` method keyword, the result would overflow anyway, but not inform you (see the next exercise).
3. The same exception occurs for integer underflow conditions.

134. I have two `int` variables `a` and `b` each with the value `Integer.MAX_VALUE`. Add these using the addition operator and print out the result. Contrast the results of this computation with the result of the preceding exercise.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MAX_VALUE; //just a value for the purposes of this program
    int b = Integer.MAX_VALUE; //as above
    int total;

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

**Output:** `-2`
**Notes:** The result of the computation is clearly wrong due to the fact that it has overflowed! Also, note that there was no exception generated!
**Conclusion:** If we want to detect under/overflow conditions, then the "`Exact`" methods should be used. However, their exceptions should be caught in order to avoid program stoppage due the exception being thrown. We look at how to handle exceptions in a later chapter.

135. We have the `int` variable, `a = Integer.MAX_VALUE`.
Write two programs to show the result of incrementing the value `a` by 1. The first program should use the standard increment operator, the second should use the `Math.incrementExact` method.

```
public class PracticeYourJava {
  public static void main(String[] args){

    int a = Integer.MAX_VALUE;

    int total =  ++a;
    System.out.println(total);
  }
}
```

```
public class PracticeYourJava {
  public static void main(String[] args){

    int a = Integer.MAX_VALUE;
    int total = Math.incrementExact(a);
    System.out.println(total);
  }
}
```

**Result:** The computation results in an overflow. However, in the first case, the overflow is silent, thus putting an erroneous result `Integer.MIN_VALUE` into the variable `total`. In the second case using `Math.incrementExact` the following exception is generated: `java.lang.ArithmeticException.`

136. We have two `int` values, `a = Integer.MAX_VALUE` and `b = 2`.
Write two programs to show the result of multiplying `a` by `b`. The first program should use the standard multiplication operator, the second should use the `Math.multiplyExact` method.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MAX_VALUE;
    int b = 2;

    int total =  a * b;
    System.out.println(total);
  }
}
```

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MAX_VALUE;
    int b = 2;

    int total = Math.multiplyExact(a, b);
    System.out.println(total);
  }
}
```

**Result:** As expected, the computation results in an overflow. However, in the first case, the overflow is silent, thus putting an erroneous result into the variable `total`: `-2`. In the second case using `Math.multiplyExact` the following exception is generated: `java.lang.ArithmeticException.`

137. We have two `int` values, `a = Integer.MIN_VALUE` and `b = 1`.
Write two programs to show the result of subtracting `b` from `a`. The first program should use the standard subtraction operator, the second should use the `Math.subtractExact` method.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MIN_VALUE;
    int b = 2;

    int total =  a - b;
    System.out.println(total);
  }
}
```

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MIN_VALUE;
    int b = 2;

    int total = Math.subtractExact(a, b);
    System.out.println(total);
    }
}
```

**Result:** As expected, the computation results in an underflow. However, in the first case, the underflow is silent, thus putting a clearly erroneous result into the variable `total`: `2147483646`. In the second case using `Math.subtractExact` the following exception is generated: `java.lang.ArithmeticException.`

138. We have the following `int` variable, `a = Integer.MIN_VALUE`.
Write two programs to show the result of decrementing the value `a` by 1. The first program should use the standard decrement operator, the second should use the `Math.decrementExact` method.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MIN_VALUE;

    int total =  --a;
    System.out.println(total);
  }
}
```

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MIN_VALUE;

    int total = Math.decrementExact(a);
    System.out.println(total);
  }
}
```

**Result:** As expected, the computation results in an underflow. However, in the first case, the underflow is silent, thus putting a clearly erroneous result into the variable `total`: `2147483647`. In the second case using

| | |
|---|---|
| | `Math.decrementExact` the following exception is generated: `java.lang.ArithmeticException`. |
| 139. | We have the following `int` variable, `a = Integer.MIN_VALUE`. Write two programs to show the result of attempting to negate this value. The first program should use the negation operator (-) and the second should use the method `Math.negateExact`. State the result of the program and explain why it is so. |

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MIN_VALUE;

    int total = -a;
    System.out.println(total);
  }
}
```

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = Integer.MIN_VALUE;

    int total = Math.negateExact(a);
    System.out.println(total);
  }
}
```

**Result:** An exception, `java.lang.ArithmeticException` is generated.
**Reason:** When worked out manually, the result of this negation `= Integer.MAX_VALUE + 1`. This clearly is a number that an `int` cannot contain (this is an overflow scenario) and therefore the method `negateExact` throws an exception.

| | |
|---|---|
| 140. | We have the following `long` variable, `a = 3047483688`. Write two separate programs to cast it to an `int`, the first program using a cast and the second using the method `Math.toIntExact`. Observe and comment on the output of each program. |

```
public class PracticeYourJava {
  public static void main(String[] args) {

    long a = 3047483688L;

    int result = (int)a;
    System.out.println(result);
  }
}
```

```
public class PracticeYourJava {
  public static void main(String[] args) {

    long a = 3047483688L;

    int total = Math.toIntExact(a);
    System.out.println(total);
  }
}
```

**Result:** The first solution gives the clearly erroneous result of this cast: `-1247483608`. The second solution rightfully throws the exception `java.lang.ArithmeticException`.

| | |
|---|---|
| 141. | If we have integer computations whose input values or results we expect to exceed the value `Long.MAX_VALUE`, what means can we use to safely handle such numbers in Java? |
| | Use the class `BigInteger`. This class represents arbitrarily long integers. Exercises on the class `BigInteger` are presented in Chapter 12, entitled *Number Handling III*. |

**Binary Numbers (Base 2)**

142. Write a program which assigns the binary value `00001110` to an `int` variable named `var01`. Print out the equivalent base 10 number.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int var01 = 0b00001110;
    System.out.println(var01);
  }
}
```

143. Write a program which adds the binary values `1000011` and `1111101`. Print out the resulting base 10 number.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int var01 = 0b1000011;
    int var02 = 0b1111101;
    int var03 = var01 + var02;

    System.out.println(var03);
  }
}
```

**Hexadecimal numbers**

144. Write a program which assigns the

```
public class PracticeYourJava {
  public static void main(String[] args) {
```

| | | |
|---|---|---|
| | hexadecimal value `FFF0` to an `int` variable named `var01`.<br>Print out the equivalent base 10 number. | `int var01 = 0xFFF0;`<br>`System.out.println(var01);`<br>`}`<br>`}` |
| 145. | Write a program which adds the hexadecimal values `1EC05` and `F238C`.<br>Print out the resulting base 10 number. | `public class PracticeYourJava {`<br>`public static void main(String[] args) {`<br>`int var01 = 0x1EC05;`<br>`int var02 = 0xF238C;`<br>`int var03 = var01 + var02;`<br>`System.out.println(var03);`<br>`}`<br>`}`<br>**Note:** *later we'll see how to print out the result in hexadecimal.* |

| **Random Number Generation** | | |
|---|---|---|
| 146. | Write a program which will generate a random `double` value $x$, where the following is the case:<br>$0 \leq x < 1$<br>*Hint:* `Math.random` | `public class PracticeYourJava {`<br>`public static void main(String[] args) {`<br>`double random01 = Math.random();`<br>`System.out.println(random01);`<br>`}`<br>`}` |
| 147. | What is the upper limit of the value generated by the method `Math.Random`? | The upper value is defined as < 1. This means that it doesn't quite get to 1. |
| 148. | The answer to the preceding exercise being the case (where the upper limit does not get to 1) write a program which will generate a random double value $x$ where the following is the case:<br>$0 \leq x < 10$ | `public class PracticeYourJava {`<br>`public static void main(String[] args) {`<br>`double random01 = Math.random() * 10;`<br>`System.out.println(random01);`<br>`}`<br>`}` |
| 149. | Write a program which will generate a random integer value between 1 and 10. | `public class PracticeYourJava {`<br>`public static void main(String[] args) {`<br>`int random01 = (int)Math.ceil(Math.random() * 10);`<br>`System.out.println(random01);`<br>`}`<br>`}`<br>**Note:** Further exerises on random numbers are presented in Chapter 20 entitled *Random Numbers*. |

**Formatting**

In this section we move on to exercises with which to hone our skills on formatting mathematical output.
In particular we look at the formatting options available for the method `printf`. Unless otherwise specified in this section, every output directive implies the usage of `printf`.
It should be noted that the method `println` does have formatting specifiers but we do not address those here.

| | | |
|---|---|---|
| 150. | Given the `int` variable `m=304638`, print the value of this variable out on separate lines in decimal, hexadecimal and octal format.<br>*Hint:* `%d`, `%x`, `%o` | `public class PracticeYourJava {`<br>`public static void main(String[] args) {`<br>`int m = 304638;`<br>`System.out.printf("In decimal :%d\n", m);`<br>`System.out.printf("In hexadecimal :%x\n", m);`<br>`System.out.printf("In octal :%o\n", m);`<br>`}`<br>`}` |

151. Given the `int` variable `m=304638`, print the value of this variable out in hexadecimal, ensuring that the letters in the hexadecimal number are uppercase.

*Hint:* `%X`

```
public class PracticeYourJava {
  public static void main(String[] args) {

     int m = 304638;
    System.out.printf("In hexadecimal :%X\n", m);
  }
}
```

152. Write a program which adds the hexadecimal values `1EC05` and `F238C`. Print out the resulting base 16 number.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int var01 = 0x1EC05;
    int var02 = 0xF238C;
    int result = var01 + var02;

    System.out.printf("result = %X\n", result);
  }
}
```

153. Run this program and comment on each line of output, explaining why each is so.

```
public class FormattingExercises {
  public static void main(String[] args){

    float var01 = 20.59f;
    System.out.printf("A) %f\n\n", var01);

    System.out.printf("B) %5.3f\n", var01);
    System.out.printf("C) %1.2f\n", var01);
    System.out.printf("D) %5.1f\n", var01);
    System.out.printf("E) %5.0f\n", var01);
    System.out.printf("F) %5f\n\n", var01);

    System.out.printf("G) %.4f\n", var01);
    System.out.printf("H) %.3f\n", var01);
    System.out.printf("I) %.2f\n", var01);
    System.out.printf("J) %.1f\n", var01);
    System.out.printf("K) %.0f\n", var01);
    System.out.printf("L) %.15f\n", var01);
  }
}
```

**Output**

```
A) 20.590000

B) 20.590
C) 20.59
D)  20.6
E)    21
F) 20.590000

G) 20.5900
H) 20.590
I) 20.59
J) 20.6
K) 21
L) 20.590000152587890
```

**Comments/Explanation**

This exercise shows more precise formatting options that are available for the `%f` parameter. We go through each one of the cases presented.

*Note:* The output format pertains to how we want the floating point value to be displayed on the screen, it is does not pertain to how it is stored internally.

`A) 20.590000` The value has been output in the default manner in which data is output for the `%f` formatting parameter.

`B) 20.590` In the formatting parameter `%5.3f`, the `5` refers to the total field width of the value that we want to output, including the decimal point. The `3` after the decimal point refers to how many digits we want after the decimal point. Given the number `20.59` (which has 5 characters) that we want displayed this fits exactly. Note though, that if the specified field width does not provide sufficient space for the whole number portion of the number being displayed, Java will ignore that field width constraint and will always print out the full whole number portion. Consider this to be a safety feature which ensures that erroneous values (due to truncation) are not presented to the user. It will also always print out the fractional part of the number to the number of spaces specified.

`C) 20.59` We used the formatting parameter `%1.2f`, implying a field width of 1 (which is clearly too small) for the whole number and 2 characters after the decimal point. As stated in the preceding explanation, if the field width is too small for the whole number portion (`20` requires 2 character widths), Java will ignore the constraint of the field width which we see it has done here. As requested, it printed out the two characters after the decimal point, further buttressing the point that it will display what is required; if the field width is too small it becomes a secondary factor.

| | | |
|---|---|---|
| | `D) 20.6` We used the formatting parameter `%5.1f`, thus ensuring only 1 character after the decimal point. Observe that rounding was applied to the number. Also observe the alignment of the number within the 5 character field.<br>`E) 21` We used the formatting parameter `%5.0f`, thus ensuring no digits after the decimal point. Observe that the number was rounded up.<br>`F) 20.590000` We used the formatting parameter `%5f`, thus ensuring a field width of five characters and implicitly telling `printf` to take care of the decimal portion as it sees fit; we see `printf` simply applying the standard 6 decimal place formatting for `%f` in this case. Again, `printf` ignored the field width which was too small.<br>`G) 20.5900` Formatting parameter `%.4f,` stating "we don't care how you display the digits before the decimal point, just ensure that there are only 4 digits after the decimal point".<br>`H) 20.590` Similar to the preceding, only that we stated we only want 3 digits after the decimal point.<br>`I) 20.59` Similar to the preceding, only that we stated we only want 2 digits after the decimal point.<br>`J) 20.6` Similar to the preceding, only that we stated we only want 1 digit after the decimal point. We see that Java applies rounding to the value.<br>`K) 21` Similar to the preceding, only that we stated we don't want any digits after the decimal point. We see that Java applies rounding to the value.<br>`L) 20.590000152587890` Similar to the preceding, only that we stated we want 15 places after the decimal point. But see, the result wasn't `20.590000000000000`, but rather a corrupted number! This corruption is due to the way in which computers handle floating point numbers; they sometimes have problems representing them properly internally. Obviously this would be a problem for such issues like the representation of currency values. Later in this book we see how to avoid this misrepresentation of floating point values. | |
| 154. | We have the following `float` numbers, 8.91, -8.9, 27.338 and 1768000111.77. Print each of these out, bounded on both sides by the vertical bar. State your observations of the output. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%f\|\n", f1);`<br>`  System.out.printf("\|%f\|\n", f2);`<br>`  System.out.printf("\|%f\|\n", f3);`<br>`  System.out.printf("\|%f\|\n", f4);`<br>`  }`<br>`}`<br>**Observations:**<br>**1.** In all cases, the numbers were written out to 6 decimal places.<br>**2.** The values are left-aligned.<br>**3.** The output numbers were not necessarily exactly what was input (discussed previously). |
| 155. | Repeat the preceding exercise, this time printing the numbers out in a field with a minimum width of 7 characters. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=27.338, f4=1768000111.77;`<br>`  System.out.printf("\|%7f\|\n", f1);`<br>`  System.out.printf("\|%7f\|\n", f2);`<br>`  System.out.printf("\|%7f\|\n", f3);`<br>`  System.out.printf("\|%7f\|\n", f4);`<br>`  }`<br>`}` |
| 156. | Repeat the preceding exercise, this time printing the numbers out in a character field mininum 8 characters mininum, to a precision of 2 decimal places.<br>State your observations of the output. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%8.2f\|\n", f1);`<br>`  System.out.printf("\|%8.2f\|\n", f2);`<br>`  System.out.printf("\|%8.2f\|\n", f3);`<br>`  System.out.printf("\|%8.2f\|\n", f4);`<br>`  }`<br>`}` |

| | | |
|---|---|---|
| | | **Observations:** the numbers are right aligned (contrast this with the results of the preceding exercise). The largest number, which is stored in the variable `f4` is larger than the specified minimum field width.<br>**Note(s):** Even if you give `printf` a field width specifier, if the number that it is supposed to print out exceeds the width specified, Java will ignore the specified field width and print the number out. This in a sense is a safely feature which prevents the user of the output from being presented with wrong data.<br>**Conclusion:** For tidy output, ensure that the chosen field width is at least the width of the maximum value that is being displayed. |
| 157. | Repeat the above exercise using a field width of 14 characters and 2 decimal places. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%14.2f\|\n", f1);`<br>`  System.out.printf("\|%14.2f\|\n", f2);`<br>`  System.out.printf("\|%14.2f\|\n", f3);`<br>`  System.out.printf("\|%14.2f\|\n", f4);`<br>`  }`<br>`}` |
| 158. | What is the difference between the `%f` and `%e` floating point format specifiers? | |
| | The `%e` specifier prints the data out in exponential format, as opposed to the `%f` specifier which prints the full number out. The exponent itself is written out as "e<sign><value>", for example $10^{-7}$ is written as `e-07`. The number $10^{-29}$ is written out as `e-29`. | |
| 159. | Print out the value `Float.MIN_VALUE` using the `%e` specifier. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  System.out.printf("%e\n", Float.MIN_VALUE);`<br>`  }`<br>`}` |
| 160. | Repeat the above exercise using a field width of 9 characters and 2 decimal places. Use the field specifier `%e` instead of `%f`. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%9.2e\|\n", f1);`<br>`  System.out.printf("\|%9.2e\|\n", f2);`<br>`  System.out.printf("\|%9.2e\|\n", f3);`<br>`  System.out.printf("\|%9.2e\|\n", f4);`<br>`  }`<br>`}` |
| 161. | Repeat the preceding exercise using the `%E` format specifier and a minimum field width of 9, with 2 decimal places. Observe and comment on the differences between this output and the output in the preceding exercise. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%9.2E\|\n", f1);`<br>`  System.out.printf("\|%9.2E\|\n", f2);`<br>`  System.out.printf("\|%9.2E\|\n", f3);`<br>`  System.out.printf("\|%9.2E\|\n", f4);`<br>`  }`<br>`}`<br>**Observation:** The difference between this output and the output with the `%e` format is simply that the "E" that represents the exponent is written as an uppercase letter with the `%E` format specifier. |
| 162. | Describe the functioning of the `%g` format. | |
| | When a floating point number is printed out using the `%g` format specifier and unconstrained by a precision specifier, the floating point number is printed out as follows: if the precision of the number is less than 6 significant digits, then it prints the full number out. If the precision of the number is greater than 6 digits, then it | |

| | | |
|---|---|---|
| | prints the number out in exponential format.<br>When a precision specifier is present, a number whose whole part is less than the width of the precision specifier will be printed out in full (with rounding), otherwise the number is printed out in exponential format. | |
| 163. | Repeat exercise 161 using the `%g` format specifier without any field width or precision specifiers. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%g\|\n", f1);`<br>`  System.out.printf("\|%g\|\n", f2);`<br>`  System.out.printf("\|%g\|\n", f3);`<br>`  System.out.printf("\|%g\|\n", f4);`<br>`  }`<br>`}` |
| 164. | Repeat the preceding exercise, this time however with a minimum field width of 9 and a precision of 2 places. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%9.2g\|\n", f1);`<br>`  System.out.printf("\|%9.2g\|\n", f2);`<br>`  System.out.printf("\|%9.2g\|\n", f3);`<br>`  System.out.printf("\|%9.2g\|\n", f4);`<br>`  }`<br>`}` |
| 165. | Redo the preceding exercise ensuring that the sign even of the positive numbers is made to appear in the output. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%+9.2g\|\n", f1);`<br>`  System.out.printf("\|%+9.2g\|\n", f2);`<br>`  System.out.printf("\|%+9.2g\|\n", f3);`<br>`  System.out.printf("\|%+9.2g\|\n", f4);`<br>`  }`<br>`}` |
| 166. | Redo the preceding exercise, ensuring that the output is left-justified. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%-9.2f\|\n", f1);`<br>`  System.out.printf("\|%-9.2f\|\n", f2);`<br>`  System.out.printf("\|%-9.2f\|\n", f3);`<br>`  System.out.printf("\|%-9.2f\|\n", f4);`<br>`  }`<br>`}` |
| 167. | Redo exercise the preceding exercise, ensuring that negative values are represented by brackets rather than by the negative sign. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%(9.2f\|\n", f1);`<br>`  System.out.printf("\|%(9.2f\|\n", f2);`<br>`  System.out.printf("\|%(9.2f\|\n", f3);`<br>`  System.out.printf("\|%(9.2f\|\n", f4);`<br>`  }`<br>`}`<br>**Observation:** Note that the numbers are right justified. |
| 168. | Redo the preceding exercise, ensuring that the values are left justified. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`  float f1=8.91, f2=-8.9, f3=327.338, f4=1768000111.77;`<br>`  System.out.printf("\|%(-9.2f\|\n", f1);`<br>`  System.out.printf("\|%(-9.2f\|\n", f2);`<br>`  System.out.printf("\|%(-9.2f\|\n", f3);` |

| | | |
|---|---|---|
| | | `    System.out.printf("|%(-9.2f|\n", f4);`<br>`  }`<br>`}` |
| 169. | Print the floating point values 1000, 120,000, -115,000 and 145,000.7743, ensuring that a grouping specifier appears in the output (for example, if the value is 1000, then print out 1,000). **Note:** the grouping specifier printed out is locale specific; for example, in some countries in Europe, it would be a ".", instead of a ",". | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`    float f1=1000f, f2=120000, f3=-115000f, f4=145000.7743f;`<br>`    System.out.printf("% ,f\n", f1);`<br>`    System.out.printf("% ,f\n", f2);`<br>`    System.out.printf("% ,f\n", f3);`<br>`    System.out.printf("% ,f\n", f4);`<br>`  }`<br>`}` |
| 170. | Using a formatting specifier, print out the value 87.5 as 87.5% | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`    float f1=87.5f;`<br>`    System.out.printf("%f%%\n", f1);`<br>`  }`<br>`}` |
| 171. | Modify the preceding exercise to print the number out to only 2 decimal places. | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`    float f1=87.5f;`<br>`   System.out.printf("%.2f%%\n", f1);`<br>`  }`<br>`}` |

| **logical & bitwise operators** | |
|---|---|
| 172. | Write a program to output the logical AND of the values 5 and 2. |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`    int a = 5, b = 2;`<br>`    int result = a & b;`<br>`    System.out.println(result);`<br>`  }`<br>`}` |
| 173. | Write a program to output the logical OR of the values 5 and 3. |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`    int a = 5, b = 3;`<br>`    int result = a \| b;`<br>`    System.out.println(result);`<br>`  }`<br>`}` |
| 174. | Write a program to output the logical XOR of the values 5 and 3. |
| | `public class PracticeYourJava {`<br>`  public static void main(String[] args) {`<br>`    int a = 5, b = 3;`<br>`    int result = a ^ b;`<br>`    System.out.println(result);`<br>`  }`<br>`}` |
| 175. | Write a program to output the one's complement of 26. |
| | `public class PracticeYourJava {` |

```
  public static void main(String[] args) {

    int a = 26;
    int result = ~a;
    System.out.println(result);
  }
}
```

176. Write a program which will *left-shift* by four places the value 38. Print the result to the console.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = 38, b = 4;
    int result = a << b;
    System.out.println(result);
  }
}
```

177. What is the difference between

```
int a=38;
result = a*2*2*2*2;
```

*and*

```
int a=38;
result = a << 4;
```

And why is the result what it is?

There is no difference in the result.
The reason for the equivalence is because computers represent data in base 2, therefore left shifting a number is the equivalent of multiplying it by 2; and thus left shifting by 4 means multiplying by $2^4$.

Think about it this way: normally we operate in base 10. Left shifting in base 10, would mean multiplying by 10. Left shifting 4 times in base 10 would mean multiplying the number by $10^4$.

178. Write a program which will *right-shift* by two places the value 38.
Explain the result.

```
public class PracticeYourJava {
  public static void main(String[] args) {

    int a = 38, b = 2;
    int result = a >> b;
    System.out.println(result);
  }
}
```

**Explanation**
Right shifting is equivalent to doing an integer division by 2 and returning the quotient of the result. Therefore right-shifting 2 times means, divide by 2 (integer division, ignoring the remainder), then divide the result (the quotient of the earlier computation) by 2 which yields the following computation and result:

```
38 >> 1 = 19
19 >> 1 =  9
```

E1 Given the following two complex numbers, 10 + 5i and 2 + 9i, write a program to divide the first number by the 2nd (*Hint:* convert both to polar form, do the computation and then convert back to Cartesian format).

E2 Write your own equivalent of the `Math.copySign` method.
*Hint:* `Math.signum`