Chapter 2

Simple Java Programming

Computational recipes are expressed by writing programs in a programming language. In this chapter, we will explore how some very simple programs can be expressed in Java. In the context of these programs, we will get our first glimpse of how Java programs are structured and how such programs are executed. In subsequent chapters, we will study the structure and execution of Java programs in much more detail.

Programming languages are usually taught in layers. The first programs are usually written in a very restricted subset of the language that uses only a few concepts and features, and new ones are introduced gradually. In Java, many of the concepts and features are so tightly intertwined that it is difficult to carve out a small subset of the language to use for introductory programs. A thorough understanding of even the simplest Java programs requires understanding many concepts and features.

To address this problem, these notes will first present many concepts at an intuitive level and treat certain features as magical incantations whose details need not be understood right away. In later chapters, the missing details will eventually be explained. This approach can be disconcerting, especially to "bottom-up" thinkers who are most comfortable understanding all fundamental concepts in detail before building on top of them. However, this approach is in line with the "black-box abstraction" theme of this course. In programming, as in any engineered system, it is important to be able to use features without understanding the details of how they work.

2.1 A First Program

Our first program simply displays the message `Welcome to Java!` on the computer screen. Here is how such a program can be written in Java:

```
public class Welcome {
  public static void main (String [] args) {
    System.out.println("Welcome to Java!");
  }
}
```

The core of the program is the phrase

```
System.out.println("Welcome to Java!");
```

which causes the message to be displayed in a window on the computer. This is an example of a *statement*, a phrase whose execution causes an action to happen. For now, you should treat `System.out.println(`*exp*`);` as a magical incantation that prints out the value of *exp*, where *exp* is a phrase (known as an *expression*) that denotes a value. In the next chapter, we will study the taxonomy of statements and expressions and explain exactly how to understand `System.out.println("Welcome to Java!");` in terms of this taxonomy.

The pair of double quote marks in `"Welcome to Java!"` are used to signify the beginning and end of a *string* – a value consisting of a sequence of characters – i.e., letters, numbers, punctuation, spaces, etc. The string `"Welcome to Java!"` consists of sixteen characters starting at `W` and ending at `!`. The double quote marks are not themselves characters in the string, and they will not appear in the displayed message.

For now, you should treat the rest of the above program as a boilerplate that is necessary for writing Java programs. That is, to write a simple Java program named *P*, you should write

```
public class P {
  public static void main (String [] args) {
    ...
  }
}
```

where ... is one or more statements specifying the computation you want to perform. Later we will explain what this boilerplate means. The subphrase

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

is called a *main method declaration* and the portion filling in the ... is called the *body* of the main method. Sometimes we will just write a main method declaration with the understanding that it needs to be embedded within a larger program in order to be executed.

As an example of a program with multiple statements in the body of the main method, consider the following:

```
public static void main (String [] args) {
  System.out.println("Welcome");
  System.out.println("to");
  System.out.println("Java!");
}
```

The multiple statements are executed one-by-one in order from top to bottom. It turns out that `System.out.println` moves to the next line after printing the given string, so that executing the new program displays the following the three-line message on the computer console:

```
Welcome
to
Java!
```

There is a variant of `System.out.println` named `System.out.print` that displays the given message but does *not* go to the next line. For example, executing the following three statements

```
System.out.print("Welcome ");
System.out.print("to ");
System.out.println("Java!");
```

displays the message

```
Welcome to Java!
```

The space characters at the end of `"Welcome "` and `"to "` are important in this example; without them, the resulting message would be `WelcometoJava!`, which is not what was intended.

To actually execute one of the above programs on a computer, jyou need to take several steps. First, you have to type the program into a *text editor* and save it as a text file. Then you need to invoke a *Java compiler* on this file to translate the Java program into a simpler language known as *Java bytecodes*. Finally, you need to execute the Java bytecodes via an application that implements what is called a *Java Virtual Machine (JVM)*. The details of each of these steps depends on your local computing environment and are beyond the scope of these notes.

If all goes well, the result of the above steps is that the message `Welcome to Java!` will be displayed in some window on your computer screen. But what do we mean by "if all goes well"? In practice, it is very easy to write programs that contain errors, and these can cause one of the above steps to fail. For instance, Java is very picky about the structure of a program, and if we neglect to include the keyword `static` in the above boilerplate, or we misspell `void` as `vod`, or we forget the matching close squiggly brace for an open squiggly brace, or we forget the semi-colon after the print statement, the Java compiler will complain that the program is ill-formed. In this case, we need to edit the text file containing the program to fix the error(s) before we resubmit the program to the compiler.

Sometimes the program will compile without error, but will encounter an error when executed by the JVM. It's hard to show an example of this in the context of the current program, but we will see many such examples later.

Finally, the program might compile and execute fine, but still contain a *logical error*. That is, the program does something, but not what we intended. For instance, if we misspell `Welcome` as `Welcom` in the above program, a message is still printed on the screen, but not the one we intended.

2.2 A Sentence Generator

A program that displays a fixed message is not very interesting. Here we consider some programs that randomly generate sentences whose words are taken from a given vocabulary.

Consider simple English sentences like "a dog runs" or "the cat eats". These are formed by having an article ("a" or "the") followed by a singular noun ("dog" or "cat") and a matching intransitive verb ("runs" or "eats").

We can generate sentences of this form in Java by randomly choosing an article, a noun, and a verb and displaying them in order. To randomly choose a word from a given vocabulary, we shall assume the existence of an expression, `StringChooser.chooseLine(`*`filename`*`)`. This expression stands for one of the lines in the text file named by the string *filename*. The line is chosen randomly and with equal probability from all possibilities. For example, suppose that the file named `verbs.txt` consists of the following three lines:

```
runs
eats
sleeps
```

Then `StringChooser.chooseLine("verbs.txt")` stands for one of `runs`, `eats`, or `sleeps`, each of which has a one in three chance of being chosen.

Here is a Java program that uses `StringChooser.chooseLine` to generate simple sentences:

```
public class Sentence {
  public static void main (String [] args) {
    System.out.print(StringChooser.chooseLine("articles.txt"));
    System.out.print(" ");  // Separate words
    System.out.print(StringChooser.chooseLine("nouns.txt"));
    System.out.print(" ");  // Separate words
    System.out.println(StringChooser.chooseLine("verbs.txt"));
  }
}
```

It is assumed that the files named `articles.txt`, `nouns.txt`, and `verbs.txt` contain articles, singular nouns, and singular verbs, respectively, with one word on each line. The program separates consecutive words with a space. The phrases `// Separates words` are *comments* that are ignored by the computer during Java execution but help the human reader understand the program. Comments begin with the character sequence `//` and run until the end of the line.

For example, suppose that:

- `articles.txt` contains the two articles `a` and `the`;
- `nouns.txt` contains the four nouns `dog`, `cat`, `bug`, and `mouse`;
- `verbs.txt` contains the three verbs `runs`, `eats`, and `sleeps`.

Then there are $2 \times 4 \times 3 = 24$ different sentences that can be generated by the program, each of which is equally likely:

```
a dog runs      a cat runs      a bug runs      a mouse runs
a dog eats      a cat eats      a bug eats      a mouse eats
a dog sleeps    a cat sleeps    a bug sleeps    a mouse sleeps

the dog runs    the cat runs    the bug runs    the mouse runs
the dog eats    the cat eats    the bug eats    the mouse eats
the dog sleeps  the cat sleeps  the bug sleeps  the mouse sleeps
```

It is easy to modify the sentence generator to generate more sentences. One way to do this without changing the program is to extend the vocabulary lists in each of the text files. But the program can also be modified to generate more complex sentences with adjectives, adverbs, transitive verbs, etc. – something that you are encouraged to do on your own.

Note that `StringChooser.chooseLine` is not limited to work on files where there is only one word per line. It returns a randomly chosen line from any text file. For example, it could be used in an "eight ball" program that answers yes/no questions via responses like:

```
Without a doubt.
It seems unlikely.
My sources say no.
It is certain.
Concentrate and ask again.
```

It could also be used to generate poems or stories from text files with phrases or sentences on each line. Even though we don't know yet how `StringChooser.chooseLine` can be built from scratch, it is a clearly useful abstraction that we can readily employ for many purposes without knowing exactly how it works.

2.3 Numerical Calculations

So far, our example programs have only manipulated strings. But Java can manipulate many different kinds of values. Here we will see how to calculate with numbers in Java.

Integer operations in Java are similar to those in mathematics. For instance, `3 + 4` denotes `7` and `3 - 4` denotes `-1`. In Java, the operator `*` stands for multiplication, so `3 * 4` (denoting `12`) is the Java way of writing what might be expressed in mathematics as 3×4, $3 \cdot 4$, or $(3)(4)$.

Division in Java is written via `/` rather than a horizontal line. E.g., $\frac{10}{2}$ (denoting `2`) is written as `10 / 2` in Java. An unusual aspect of division in Java is that the result of dividing two integers is always an integer – in particular, the integer part of the division (ignoring the remainder). Thus, `6 / 3`, `7 / 3`, and `8 / 3` all denote the integer `2` in Java. The remainder of a division is returned by the `%` operator; `6 % 3` denotes `0`, `7 % 3` denotes `1`, and `7 % 3` denotes `2`.

Java also has *floating point numbers*, which are numbers that conceptually have an integer part before the decimal point and a fractional part after the decimal point. Examples of floating point numbers are 3.14159 and -273.15. While 10 and 10.0 are indistinguishable in mathematics, Java treats as having different *types*; the first is an integer (has Java type `int`) while the second is a floating point number (has Java type `double`[1]). The difference can be seen with numerical operators like `+`, `-`, `*`, and `/`. In Java, if both operands are integers, the result is an integer, but if at least one operand is a floating point number, the result is a floating point number. For example:

[1]The name of the `double` type comes from the phrase "double-precision floating point numbers". Java also supports single-precision floating point numbers with type `float`, but we shall not study these.

Expression	Result	Type of Result
`2.718 + 3.141`	`5.859`	`double`
`2.718 + 3`	`5.718`	`double`
`2 + 3.141`	`5.141`	`double`
`2.0 + 3`	`5.0`	`double`
`2 + 3.0`	`5.0`	`double`
`2 + 3`	`5`	`int`
`9.0 / 4.0`	`2.25`	`double`
`9.0 / 4`	`2.25`	`double`
`9 / 4.0`	`2.25`	`double`
`9 / 4`	`2`	`int`

Another difference between Java numbers and mathematics is that operations involving floating point numbers are only an approximation of the actual mathematical result. For example, in math, $7.0/3.0 = 2\frac{1}{3} = 2.333\ldots$, where the final representation is a 2 followed by a decimal point followed by an infinite sequence of 3s. In Java, however, the result of `7.0/3.0` is `2.3333333333333335`, in which there are only a finite number of digits after the decimal point. The details of how floating point numbers are approximated in Java are important to anyone who cares about the meaning of the result of floating point computation, but such details are beyond the scope of this book.

The interpretation of sequences of numerical operators in Java follows the conventions of those used in mathematics. For example, the expression `1 - 4 + 5 * 6 + 7 / 3` is interpreted as if it were written `((1 - 4) + (5 * 6)) + (7 / 3)`, where explicit parenthesis have been used to indicate the order in which operations are done. The details of how such operator sequences are interpreted are explained in the next chapter.

We can display the result of any numerical calculation in Java using `System.out.println`, which displays numbers as well as strings. For example, here is a program performing some of the sample calculations from above:

```
public class NumericalExamples {
  public static void main (String [] args) {
    System.out.println(2 + 3.0);
    System.out.println(2 + 3);
    System.out.println(9.0 / 4);
    System.out.println(9 / 4);
  }
}
```

The result displayed by executing this program is:

```
5.0
5
2.25
2
```

Note that `System.out.println(2 + 3)`, which displays the result `5`, is very different than `System.out.println("2 + 3")`, which displays `2 + 3`. In the first case, the argument of `System.out.println` is the integer that results from adding `2` and `3`, while in the second

case the argument is a string of five characters: the digit `2`, a space, the symbol `+`, a space, and the digit `3`.

It is hard to read the results displayed by the above program because it is necessary to mentally match each of the four results with the associate expression. To make the matching more obvious, we can print the expression before each result:

```
public class NumericalExamples {
  public static void main (String [] args) {
    System.out.print("2 + 3.0 = ");
    System.out.println(2 + 3.0);
    System.out.print("2 + 3 = ");
    System.out.println(2 + 3);
    System.out.print("9.0 / 4 = ");
    System.out.println(9.0 / 4);
    System.out.print("9 / 4 = ");
    System.out.println(9 / 4);
  }
}
```

The result displayed by executing the modified program is:

```
2 + 3.0 = 5.0
2 + 3 = 5
9.0 / 4 = 2.25
9 / 4 = 2
```

2.4 String Concatenation

Java's `+` operator is not only used to add numbers. It is also used to concatenate two strings together. For example, the expression `"Java" + "line"` denotes the 8-character string `"Javaline"`. This string could also be expressed in any one of the following ways as well:

```
"Ja" + "valine"
"Jav" + "ali" + "ne"
"J" + "a" + "v" + "a" + "l" + "i" + "n" + "e"
```

Using `+` for string concatenation can simplify programs by reducing the number of calls to `System.out.print` and `System.out.println`. For example, the sentence generating program from above can be rewritten so that the main method has a body with a single statement:

```
public class Sentence {
  public static void main (String [] args) {
    System.out.println(StringChooser.chooseLine("articles.txt")
                       + " " // Separate words
                       + StringChooser.chooseLine("nouns.txt"));
                       + " "  // Separate words
                       + StringChooser.chooseLine("verbs.txt"));
  }
}
```

An operator like + that has different meanings in different contexts (e.g., add two integers vs. add two floating point numbers vs. concatenate two strings) is said to be *overloaded*. In Java, the meaning of e_1 + e_2 is determined by the types of the operands e_1 and e_2:

- If e_1 and e_2 are both integers, then e_1 + e_2 stands for an integer addition.
- If one of e_1 or e_2 is a floating point number and the other is a floating point number or integer, then e_1 + e_2 stands for a floating point number addition. The integer operand (if there is one) is converted to a floating point number before the addition is performed. Thus, Java treats `2.0 + 3` as if it were `2.0 + 3.0`.
- If at least one of e_1 or e_2 is a string, then e_1 + e_2 stands for a string concatenation. The non-string operand (if there is one) is converted to a string before the concatenation is performed. For example, `17` would be converted to `"17"` and `-273.15` would be converted to `"-273.15"`. So `"CS" + 111` is treated as `"CS" + "111"`, which denotes the string `"CS111"`, and `3.141 + "..."` is treated as `"3.141" + "..."`, which denotes the string `"3.141..."`.

Here are a some more examples illustrating the overloading of +:

Expression	is treated as	whose value is
`42 + 3.141`	`42.0 + 3.141`	`45.141`
`"42" + 3.141`	`"42" + "3.141"`	`"423.141"`
`42 + "3.141"`	`"42" + "3.141"`	`"23.141"`
`63 + -273.15`	`63.0 + -273.15`	`-210.15`
`"63" + -273.15`	`"63" + "-273.15"`	`"63-273.15"`
`63 + "-273.15"`	`"63" + "-273.15"`	`"63-273.15"`

String concatenation with implicit coercions can be used to shorten the numerical example program from above:

```
public class NumericalExamples {
  public static void main (String [] args) {
    System.out.print("2 + 3.0 = " + (2 + 3.0));
    System.out.print("2 + 3 = " + (2 + 3));
    System.out.print("9.0 / 4 = " + (9.0 / 4));
    System.out.print("9 / 4 = " + (9 / 4);
  }
}
```

In this case, the parenthesis around numerical expressions like `(2 + 3.0)` are required, not optional. If we instead wrote `"2 + 3.0 = " + 2 + 3.0`, this would be parenthesized by Java as `("2 + 3.0 = " + 2) + 3.0`, and by the coercion rules from above this would be equivalent to `("2 + 3.0 = " + "2") + "3.0"`. So the displayed result would be

```
2 + 3.0 = 23.0
```

which is not at all what was intended!

2.5 Naming Values Via Variables

Suppose that we want to calculuate the sum of the series of integers $1, 2, 3, \ldots, n$, where n is a positive integer. Observe that the first and last integers in this series sum to $n + 1$, that the second and second-to-last integers sum to $n + 1$, that the third and third-to-last integers sum to $n + 1$, etc. Since there are $n/2$ pairs of integers in the series, and each pair sums to $n + 1$, a formula for the sum of the series is $\frac{n}{2} \cdot (n + 1)$. For example, the sum of the integers 1 through 9 is $\frac{9}{2} \cdot (9 + 1) = \frac{9}{2} \cdot 10 = 45$. Note that even though the formula contains a division by 2, it always returns an integer result because one of n and $n + 1$ is even, and therefore evenly divisible by 2.

In the case of $n = 9$, we can calculate and display the series sum via the statement:

```
System.out.println((9 * (9 + 1)) / 2);
```

Because of the meaning of integer division, we must be sure *not* to write `(9 / 2) * (9 + 1)`, since this would calculate $4 \cdot 10 = 40$, which is not what was intended. Encoding numerical formulae into Java requires more than mere transcription!

In this simple formula, it is not difficult to plug in a particular number for n, but it would be tedious to do so in more complex formulae that mention many mathematical variables and/or reference variables many times. This situation can be improved by using Java variables, which allow programmers to name values and refer to them later by name. Here is a series summation program that uses a Java variable `n`:

```
public class SeriesSum {
  public static void main (String [] args) {
    int n = 9;
    System.out.println((n * (n + 1)) / 2);
  }
}
```

The statement `int n = 9;` is a *local variable declaration statement* that introduces a *local variable* named `n`. This can be envisioned as a box named `n` that holds a value. The `int` part of the declaration says that the box must hold an integer value, and the `= 9` part of the declaration indicates that the initial value stored in the box is the integer `9`. Later mentions to the name `n` within the `main` method stand for the value in the variable box named `n` – in this case, the integer 9.

The general form of a local variable declaration statement is

type name = exp

where *name* is the name of the variable box, *type* indicates the type of value that may be stored in the variable box, and *exp* is an expression whose value will be stored into the variable box.

The fact that the initial value of a variable may be the result of evaluating an expression is important for avoiding repeating calculations in a program. For example, suppose we want to calculate $\frac{a^2+b^2}{a^2-b^2}$, where a is 10.27 and b is 8.56. One way to express this in Java is to write:

```
public static void main (String [] args) {
  double a = 10.27;
  double b = 8.56;
  System.out.println((a*a + b*b)/(a*a - b*b));
}
```

But this program calculates `a*a` twice and `b*b` twice. These repeated calculations could be avoided by introducing additional variables:

```
public static void main (String [] args) {
  double a = 10.27;
  double b = 8.56;
  double a_squared = a*a;
  double b_squared = b*b;
  System.out.println((a_squared + b_squared)/(a_squared - b_squared));
}
```

In the modified program, the result of calculating `a*a` is stored in a variable named `a_squared`, and the result of calculating `b*b` is stored in a variable named `b_squared`. Each of these calculated values is used twice in the remaining computation.

It is even possible to name the result of *every* expression in a Java program. For instance, yet another version of the above program is:

```
public static void main (String [] args) {
  double a = 10.27;
  double b = 8.56;
  double a_squared = a*a;
  double b_squared = b*b;
  double sum = a_squared + b_squared
  double diff = a_squared - b_squared
  double quot = sum/diff
  System.out.println(quot);
}
```

This is going a bit overboard with variables! Variables are most useful for (1) naming values that are used several times in a program (such as `a`, `b`, `a_squared`, and `b_squared`) or (2) naming values that are constants in a particular program, but which might change when the program is edited (such as `a` and `b`)[2]. Expressions whose values are calculated at most

[2]Another essential use for variables is to serve as a placeholder for values that can change over time. We will stress this time-varying nature of variables in later chapters. In the meantime, we will use variables to denote a box whose value does *not* change after it is initialized.

once and are unlikely to be edited during program development do not need to be named via variables. Indeed, the expression itself already serves as a kind of "structured name" for the value it denotes. Even when the value of an expression is used only once, though, it is sometimes helpful to introduce variables because the variable helps to decompose a complex expression into manageable parts and/or the name serves as a comment that helps the programmer to understand the meaning of the expression.

With the notion of variables in hand, we can generalize our series summation program. An *arithmetic series* is a series of n integers $i_1, i_2, \ldots i_n$ where each element is greater than the previous element by the same value d. For instance, here are some examples of arithemetic series:

Series	n	i_1	i_n	d
$7, 8, 9, 10, 11, 12$	7	6	12	1
$2, 4, 6, 8$	4	2	8	2
$6, 11, 16, 21, 26, 31, 36, 41$	8	6	41	5

As before, the sum of such a series is the number of pairs of elements multiplied by the sum of the first and last elements: $\frac{n}{2} \cdot (i_1 + i_n)$.

Suppose that we are not given n, but are given the first and last elements of the series and the difference d between consecutive elements. Then we can calculate n as $\frac{i_n - i_1}{d} + 1$. For instance, for series with first element 6, last element 41 and difference 5, $n = \frac{41-6}{5} + 1 = 8$. The $+1$ in the formula might seem wrong at first, but it is necessary to count both ends of the series. Omitting the $+1$ would lead to a *fence post error*. This term comes from the fact that $n + 1$ (and not n) vertical fence posts are required to construct a fence whose length is n horizontal beams. Such off-by-one errors are extremely common in numerical programs You should always think carefully about your formulae to ensure that they calculate what they expect. An important technique is to plug in particular values for the variables and check if the resulting value is correct.

Here is a Java program for calculating the sum of the more general form of an arithmetic series:

```
public class ArithmeticSeriesSum {
  public static void main (String [] args) {
    int first = 6;
    int last = 41;
    int d = 5;
    int n = ((last - first) / d) + 1;
    int sum = (n * (first + last))/2;
    System.out.println(sum);
  }
}
```

The variables `first`, `last`, and `d` are effectively the "inputs" to the program, and in practice will be changed before every execution of the program. Here these inputs have been instantiated to `6`, `41` and `5`, respectively. The variable `n` names the result of the formula for calculating the number of elements in the series, and the variable `sum` names the result of calculating the summation formula. Although neither `n` nor `sum` is used more than once, naming them via variables clarifies the structure of the program.

There are still several ways in which the general summation program can be improved.

- There should be a more convenient interface for specifying the inputs to the program. It is tedious and error-prone to edit and recompile the program every time you want a new series summation. Suppose that the expression `IO.readInt(`*`prompt`*`)` reports an integer typed in by the user of the program after the user has been prompted with the string *prompt*. Then the program can be improved by writing:

```
public class ArithmeticSeriesSum {
  public static void main (String [] args) {
    int first = IO.readInt("Enter the first element of the series:");
    int last = IO.readInt("Enter the last element of the series:");
    int d = IO.readInt("Enter the difference between elements:");
    int n = ((last - first) / d) + 1;
    int sum = (n * (first + last))/2;
    System.out.println(sum);
  }
}
```

  An even better interface to the program might be to develop a window with labeled fill-in-the-blank slots for the program inputs, an area where the resulting summation is displayed, and a calculation button that calculates the results for the currently specified inputs.

- A nice user interface makes it easier for a *person* to use the summation program, but does not make it easier for other *programs* to use the summation program. It is not difficult to imagine a mathematical application that needs to calculate many series summations. It should be possible to encapsulate the knowledge of such calculations in a black-box computational entity that can be used arbitrarily often. In Java, the right entity to use for this purpose is called a *method*. We shall see how to define methods other than the `main` method in a few chapters.

- The program does not handle certain error or boundary cases. For instance, suppose that `last` cannot be reached from `first` by adding a multiple of `d`. As a concrete example, suppose that `first` is 5, `last` is 9, and `d` is 3. Then the series `5`, `8`, `11`, ... does not contain `9`!. It is important to decide how this situation should be handled. One approach is to indicate that there is an error, perhaps halting the program as well. Another approach is to use a different value in place of `last` – perhaps the largest element of the series less than or equal to `last` (in this case, `8`) or the smallest element of the series greater than or equal to `last` (in this case, `11`).

  As structured, the program does not allow for an empty series – i.e., a series that contains zero elements. Such a series is mathematically sensible and has a sum of 0, but there is no obvious way to specify it in with the current interface. What would `first`, `last`, and `d` be for an empty series?

  Interestingly, the program *does* already handle the case where `last` is smaller than `first`. Consider the series $4, 1, -2, -5, -8$ where `first` is `4`, `last` is `-8` and `d` is `-3`. Then `n` is calculated to be `((-8 - 4)/-3) + 1` = `(-12/-3) + 1` = `4 + 1` = `5`, and the summation is indeed `(5 * (4 + -8))/2` = `(5 * -4)/2` = `-20/2` = `-10`. This is an example of *serendipity*, where the program correctly handles a case that we did

not consider explicilty. But serendipity is rare in programs, so you should carefully consider all boundary cases in practice.