Contents

Mathematical Computations in Java **2**
- Types 2
  - Definitions 2
  - Primitives 4
  - Wrapper classes 4
  - Classes for extended range or precision 5
- Basic operations 6
  - Arithmetic operators defined by the Java language 6
  - Sign-related operations 6
  - Rounding operations 7
  - Modulo operation (remainder) 9
- Exponents, roots, and logarithms 10
- Sequences 11
  - Definition 11
  - Length 12
  - Recurrence relations 13
  - Sum 13
  - Product 14
    - General case 14
    - Factorial 15
- Sets 15
  - Definition 15
  - Cardinality 16
  - The empty set 16
  - Membership 16
  - Equality 17
  - Subsets & supersets 17
  - Set-builder notation 18
  - Union 20
  - Intersection 20
  - Difference (relative complement) 21
  - Symmetric difference 21
- Probability 22
  - Independent events 22
  - Disjoint events 22
  - Naïve probability 23
  - Uniform discrete probability distributions 23
  - Non-uniform discrete probability distributions 24
  - Continuous probability distributions 24
- Combinations & permutations 25
- Trigonometry 27
  - Right triangles 27
  - Polar-Cartesian relationship 28

Mathematical Computations in Java

While advanced mathematics is generally not required for general-purpose programming, an understanding of arithmetic computations, exponents & roots, logarithms, and basic algebra is an essential part of a solid foundation for programming. When supplemented by a few key concepts from number theory, set theory, probability, and trigonometry, such an understanding can dramatically expand the set of career paths (and career longevity) available to a programmer.

Summarized below are the Java data types (those defined in the language, and those provided by the standard library) used in mathematical computations, along with mathematical concepts and computations you may encounter in the assignments and projects of this bootcamp. Most of these are defined in mathematical terms, and accompanied by short code snippets.[1]

Fair warning: Though I try to avoid it, I'm overly verbose in some moments, and insufficiently rigorous in others. On the other hand, I'm almost always open to requests for clarification and suggestions for improvement.

Types

Definitions

- Integer

  An *integer* is a number without a fractional part—for example, 7, -3, and 0 are all integers, while 1.5 is not. The set of all integers (often referred to as $\mathbb{Z}$) consists of the natural numbers (1, 2, 3, …), referred to collectively as $\mathbb{N}$, their additive inverses (-1, -2, -3, …), and the number 0.

  The Java language defines several primitive integer data types; additionally, the Java standard library defines wrapper object types corresponding to nearly all of the integer primitive types. In use, these types differ from each other primarily in the range of values that can be represented by each.

- Rational number

  A *rational number* is a number that can be expressed as a fraction p/q, where p and q are both integers, with $q \neq 0$. The set of rational numbers, denoted $\mathbb{Q}$, is a superset of $\mathbb{Z}$ (the set of integers).

[1] The Java classes and interfaces referred to in the code snippets and accompanying text are members of the `java.lang`, `java.math`, `java.util`, and `java.util.stream` packages of the Java standard library. Since all classes and interfaces contained directly in the `java.lang` package are automatically imported by the Java compiler, these classes can be referenced directly without an `import` statement. Classes and interfaces in other packages must be imported, or referenced via fully-qualified names; however, for the sake of clarity and brevity, this requirement is ignored in the code snippets.

Neither the Java language nor the Java standard library has direct support for rational numbers, but there are many 3rd-party libraries (including the widely used Apache Commons Math library) that do.

- Real number

  A *real number* is a value of a continuous, rather than discrete, quantity. We can think of real numbers as those numbers that can represent exactly *any* finite value along a number line, from 0 at the center, extending (in opposite directions) towards $-\infty$ and ∞. The set of all real numbers is usually denoted as $\mathbb{R}$, and is a superset of the set of rational numbers, $\mathbb{Q}$.

  The Java language and standard libraries represent real numbers (exactly or approximately) via 2 different approaches:

  - *Fixed-precision floating-point*, consisting of 3 components:

    * A sign (+1 or -1).

    * An *exponent* (positive or negative).

    * A *mantissa*, a fractional value in the interval $[1, 2)$. (Under certain conditions, this is interpreted as a fractional value in $[0, 1)$.)

    The value represented by these components is $sign \cdot mantissa \cdot 2^{exponent}$.

  - *Arbitrary-precision floating-point*, with 2 components:

    * An arbitrary-length integer (positive or negative); this is sometimes called the *unscaled* value.

    * A *scale* exponent (positive or negative).

    The value represented is $unscaled \cdot 10^{-scale}$. This can be understood simply as an integer shifted to the right or left by some number of decimal digit places.

  It's important to observe that, even with the arbitrary-precision representation, the precision of encoded values has limits: there are an infinite number of real values that cannot be represented exactly with either of the above approaches. It's also interesting to note that while most formal definitions of real numbers do not include $-\infty$ and ∞, the floating-point representations used by Java and many other programming languages *are* capable of representing $-\infty$ and ∞, as well as *NaN* ("not a number"—e.g. the value resulting from 0/0).

- Complex numbers

  A *complex number* takes the form $z = a + bi$, where a and b are real numbers, and $i = \sqrt{-1}$. This gives us a definition of the set of complex numbers, $\mathbb{C}$, as

$$\mathbb{C} = \left\{ a + bi \mid a, b \in \mathbb{R} \text{ and } i = \sqrt{-1} \right\}.$$

(The notation used here is *set-builder notation*, described in more detail below.)

Neither the Java language nor the Java standard library supports complex numbers directly; however, Apache Commons Math library does, as do some other 3rd-party libraries.

Primitives

The Java language defines 5 integer primitive types (including `char`, which is treated as an integer type for numeric computations, and as a single Unicode character in string-related operations), and 2 floating-point primitive types. Values of these 7 types, along with those of the primitive `boolean` type, are not objects; they have no behavior (methods), only state (data). However, these primitive types are the basic building blocks—not only of their corresponding wrapper types, but also (indirectly) of all Java classes.

- Integer

Type	Size (bits)	Range (inclusive)
`byte`	8	$-128 \ldots 127$
`char`	16	$0 \ldots 65{,}535$
`short`	16	$-32{,}768 \ldots 32{,}767$
`int`	32	$-2^{31} \ldots (2^{31} - 1)$
`long`	64	$-2^{63} \ldots (2^{63} - 1)$

- Floating-point

Type	Size (sign/exponent/mantissa bits)	Range	Sig. digits
`float`	1/8/23	$(-2^{128}, 2^{128})$	~7
`double`	1/11/52	$(-2^{1024}, 2^{1024})$	~16

Wrapper classes

The `java.lang` package of the Java standard library includes a wrapper class for each of the primitive types defined in the Java language. Since these are object types, rather than primitive types, they can (and do) define methods. The wrapper classes include methods for parsing and constructing string representations of numeric values, testing for special values, and performing additional

operations not provided by the Java language itself. These classes also include constants for the maximum and minimum values representable by the integer types, and for the largest and smallest magnitudes representable by the floating-point types.

Primitive type	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

(Note that while the `char` primitive has a corresponding `Character` wrapper type, the latter is not a subclass of `java.lang.Number`, and is thus not considered a numeric wrapper type.)

The Java compiler will, in many (but not all) cases, automatically generate code to wrap a primitive in an instance of its corresponding wrapper type when the latter is expected, or to unwrap a primitive from an instance of its wrapper type when the former is expected. These operations are called *auto-boxing* and *auto-unboxing*, respectively.

```
int a = 15;
Integer b = a + 10; // int value auto-boxed, assigned to Integer.
Integer c = a + b;  // Integer auto-unboxed for addition; result
                    // auto-boxed for assignment.
int d = b * c;      // Integers auto-unboxed for multiplication;
                    // result assigned to int.
```

Classes for extended range or precision

The following classes are found in the `java.math` package of the Java standard library, and support extended range and precision integer and decimal values, and operations on those values. Unlike the primitive and wrapper types, the size of instances of these types is not fixed by definition, nor at compile time, but depends on the values assigned to them, up to the maximum sizes shown here.

Type	Max. size (bits)	Range (inclusive)
`BigInteger`	2^{32}	$-2^{(2^{31}-1)} \ldots (2^{(2^{31}-1)} - 1)$
`BigDecimal`	$32 + 2^{32}$	$-2^{(2^{31}-1)} \cdot 10^{(2^{31})} \ldots (2^{(2^{31}-1)} - 1) \cdot 10^{(2^{31})}$

Both `BigInteger` and `BigDecimal` have a maximum of 646,456,993 significant digits. The smallest absolute value representable by `BigDecimal` is $10^{(1-2^{31})}$.

As you might infer from the above information, a `BigDecimal` instance is composed of a `BigInteger` instance (the unscaled value), along with an `int` specifying how many decimal digit places the `BigInteger` value should be shifted to the right or the left (the scale).

Basic operations

Arithmetic operators defined by the Java language

The Java language defines several arithmetic operators—as well as bitwise, logical, reference, and string operators. These are listed—along with their evaluation precedence and other details—in Java Operators.

Sign-related operations

- Signum

  The *signum* function (or sign function) of a number is simply a value corresponding to its sign, where 1 denotes positive, -1 denotes negative, and 0 denotes 0.

  $$\operatorname{sgn}(x) = \begin{cases} 1, \text{ if } x > 0 \\ 0, \text{ if } x = 0 \\ -1, \text{ if } x < 0 \end{cases}$$

  The Java standard library provides the `Math.signum` method for obtaining the sign of a floating-point value:

  ```
  double x = 1.5;
  System.out.println(Math.signum(x)); // 1.0
  double y = -2.5;
  System.out.println(Math.signum(y)); // -1.0
  ```

  The `Math.signum` method can also be used with an integer value, which will automatically be *widened* to a floating-point representation. Otherwise, a ternary operation can be used—enclosing another ternary operation or an *unsigned shift right*:

  ```
  int a = 2;
  int b = -3;
  System.out.println(Math.signum(a)); // 1.0
  System.out.println(Math.signum(b)); // -1.0
  System.out.println((a == 0) ? 0 : ((a > 0) ? : 1 : -1)); // 1
  ```

```
System.out.println((b == 0) ? 0 : ((b > 0) ? : 1 : -1)); // -1
System.out.println((a == 0) ? 0 : 1 - 2 * (a >>> 31)); // 1
System.out.println((b == 0) ? 0 : 1 - 2 * (b >>> 31)); // -1
```

- Absolute value

  The *absolute value* of a number is the distance from the origin (zero) to that number, without regard to direction (i.e. the distance is always non-negative).

  $$|x| = \begin{cases} x, \text{ if } x \geq 0 \\ -x, \text{ if } x < 0 \end{cases}$$

  We can view the absolute value and the signum function as complementary operations:

  $$x = \operatorname{sgn}(x) \cdot |x|$$

  In Java, the absolute value of an integer or floating-point value can be computed with the `Math.abs` method. Alternatively, the overhead of a method call can be avoided (at the cost of reduced code clarity) by using a *ternary operation*:

```
double x = 1.5;
System.out.println(Math.abs(x)); // Prints 1.5.
double y = -2.5;
System.out.println(Math.abs(y)); // 2.5
System.out.println((y >= 0) ? y : -y); // 2.5
```

Rounding operations

- Floor

  The *floor* of a number is the largest integer that is less than or equal to that number. In other words, the floor of a number is the result obtained by rounding that number down towards $-\infty$.

  $$\lfloor x \rfloor = \max \{m \in \mathbb{Z} \mid m \leq x\}.$$

  (The notation used here is *set-builder notation*, described in more detail below.)

  The Java standard library method `Math.floor` is used for floor rounding of floating-point values to integer values:

```
double x = 1.5;
System.out.println(Math.floor(x)); // Prints 1.
double y = -2.5;
System.out.println(Math.floor(y)); // Prints -3.
```

Java also provides the `Math.floorDiv` method, which performs integer division with automatic floor rounding.

- Ceiling

  The *ceiling* of a number is the smallest integer that is greater than or equal to that number. In other words, the ceiling of a number is the result obtained by rounding that number up towards ∞.

  $$\lceil x \rceil = \min \{m \in \mathbb{Z} \mid m \geq x\}.$$

  The `Math.ceil` method in the Java standard library can be used to perform ceiling rounding:

```
double x = 1.5;
System.out.println(Math.ceil(x)); // Prints 2.
double y = -2.5;
System.out.println(Math.ceil(y)); // Prints -2.
```

- Truncation

  While not as common as floor or ceiling in mathematics, truncation—or rounding towards zero—is useful in many computational problems. The simplest way to define this operation is in terms of the floor and ceiling operations:

  $$\operatorname{trunc}(x) = \begin{cases} \lfloor x \rfloor, \text{ if } x \geq 0 \\ \lceil x \rceil, \text{ if } x < 0 \end{cases}$$

  Some programming languages have a function specifically for truncation. In Java, however, truncation is performed implicitly when *casting* a floating-point value to an integer-type value.

```
double x = 1.5;
System.out.println((int) x); // 1
double y = -2.5;
System.out.println((int) y); // -2
```

Also, when an integer-type dividend is divided by an integer-type divisor, using the Java `/` operator, the result is automatically truncated.

```
int a = 7;
int b = 3;
```

```java
System.out.println(a / b);  // 2
System.out.println(a / -b); // -2
```

- Rounding to nearest integer

  Rather than rounding towards $-\infty$ (floor), ∞ (ceiling) or 0 (truncate), we often want to round to the integer closest to the original value. This is usually what is meant when we refer to *rounding* without specifying a rounding strategy.

  But how should we round a value equidistant from its two nearest integers—for example, should ½ be rounded to 0, or to 1? In the method shown below, the Java library uses the *round half up* convention, in which a tie results in rounding towards ∞. This is expressed mathematically as

$$\text{round}\,(x) = \left\lfloor x + \frac{1}{2} \right\rfloor = \left\lceil \frac{\lfloor 2x \rfloor}{2} \right\rceil$$

  The Java standard library provides the `Math.round` method to support nearest-integer rounding using the round half up convention:

```java
double x = 1.25;
double y = 2.5;
double z = -3.5;
System.out.println(Math.round(x)); // 1
System.out.println(Math.round(y)); // 3
System.out.println(Math.round(z)); // -3
```

Modulo operation (remainder)

The *modulo operation* is the computation of the remainder obtained after division of one number by another. Typically, this is defined as

$$a \bmod n = a - n \left\lfloor \frac{a}{n} \right\rfloor, \text{ where } n \in \mathbb{N} \quad (1)$$

Alternatively, we may define it as

$$a \bmod n = a - n \cdot \text{trunc}\left(\frac{a}{n}\right), \text{ where } n \in \mathbb{N} \quad (2)$$

In number theory, the divisor (or *modulus*) is generally assumed to be a positive integer, as shown above. In computation, it is often useful to broaden this assumption, allowing the divisor to be positive or negative, and integer or real-valued. (In any event, the result of the modulo operation using a divisor of 0 is generally undefined.)

Java has two mechanisms for performing the modulo operation: the `Math.floorMod` method, which implements (1), and the `%` operator (included in [Java Operators]({{ "/assets/pdf/Operators%20(tabloid).pdf" | relative_url }})), which implements (2). This distinction leads to the two approaches giving different results when the signs of the dividend and divisor are different. (Both `%` and `Math.floorMod` allow negative values for both the divisor and the dividend—however, `Math.floorMod` only allows integer-type dividend and divisor, while `%` supports floating-point values as well.)

```
int a = 7;
int b = 3;
System.out.println(a % b);                  // 1
System.out.println(Math.floorMod(a, b)); // 1
int c = -3;
System.out.println(a % c);                  // 1
System.out.println(Math.floorMod(a, c)); // -2
```

Exponents, roots, and logarithms

Assume we have 3 numbers, b, p, and c, with

$$b^p = c \quad (3)$$

We might read this as "b raised to the pth power equals c."

If we recognize that $(a^m)^n = a^{mn}$, and if $p \neq 0$, we might solve for b in this fashion:

$$\begin{aligned} (b^p)^{1/p} &= c^{1/p} \\ b &= c^{1/p} \end{aligned}$$

By definition, and by the conventions used with the √ notation, this gives us

$$b = \sqrt[p]{c} \quad (4)$$

Thus, one solution to (3) is found in (4). We can read the latter as "b equals the nth root of c." We also see that computing the nth root of a number is the same as raising that number to the $(1/n)^{\text{th}}$ power; in general, this is how we compute roots in Java.

If $b > 0$ and $c > 0$, we can express (3) in terms of a solution for n:

$$
\begin{aligned}
\log_b(b^n) &= \log_b c \\
n \log_b b &= \log_b c \\
n &= \log_b c \quad (5)
\end{aligned}
$$

Finally, an identity of logarithms tells us that we can express the base-b logarithm of (5) using a different base—for example, e, the base of *natural logarithms*:

$$
\log_b c = \frac{\ln c}{\ln b} \quad (6)
$$

The `Math` class of the Java standard library defines the `log` (natural logarithm), `log10` (base-10, or *common*, logarithm), `pow` (power), and `exp` (e^x) methods to perform the above operations.

```
double x = 10;
double y = 2;
double z = Math.pow(x, y);
System.out.println(z);                              // 100.0
System.out.println(Math.pow(z, 1 / y));             // 10.0
System.out.println(Math.log10(z));                  // 2.0
System.out.println(Math.log(z));                    // 4.605170185988092
System.out.println(Math.log(z) / Math.log(x)); // 2.0
System.out.println(Math.exp(Math.log(z)));   // 100.00000000000004
```

There are also a number of methods in the `Math` class that implement special-case power, root, and logarithmic computations. These include `sqrt` ($x^{1/2}$ or $\sqrt{x}$), `cbrt` ($x^{1/3}$ or $\sqrt[3]{x}$), `expm1` ($e^x - 1$), `log1p` ($\ln(1 + x)$), and `scalb` ($x \cdot 2^y$).

Sequences

Definition

A *sequence* is an ordered, enumerable collection of values. While it may not be obviously the case, such a collection need not be finite in size—neither from a theoretical standpoint, nor a practical standpoint. Conversely, a sequence may also be empty—that is, its length can be 0.

The values in a sequence are generally homogeneous (all of the same type), and are not necessarily unique—that is, a given value may appear more than once in a single sequence. Each value in a sequence can be referenced by its *index*, or position. Typically (but not always), index values begin at either 0 (*zero-based*) or 1 (*one-based*). So, depending on the context, the finite sequence A of length n might be defined as

$$A = (a_0, a_1, \ldots, a_{n-1}),$$

or

$$A = (a_1, a_2, \ldots, a_n).$$

In Java, finite sequences are most often implemented as *arrays* or *lists*. A Java array is of finite and fixed length, while a list (i.e. some object of a class that implements the *`List`* interface) is of finite, but not necessarily fixed, length. Arrays are structures supported directly by the Java language itself, while the *`List`* interface (as well as several implementations, including `ArrayList` and `LinkedList`) is defined in the Java standard library; thus, the syntax for using an array is a bit more direct than that for a list. Both, however, use zero-based indices.

```
int[] dataArray = {10, 5, 3, 2, 5}; // Declare & initialize.
System.out.println(dataArray[2]); // 3
dataArray[3] = 20; // {10, 5, 3, 20, 5}.
System.out.println(dataArray[3]); // 20

List<Integer> dataList = new LinkedList<>(); // Create empty List.
Collections.addAll(dataList, 10, 5, 3, 2, 5); // Populate the list.
dataList.set(4, 20); // {10, 5, 3, 2, 20}.
System.out.println(dataList.get(4)); // 20
```

Another key difference between arrays and lists is that an array can be declared to have elements of any type, whether primitive or object, while a list can only contain instances of object types. (Fortunately, as implied in Wrapper classes, we can add a primitive value to a list that is declared to contain the corresponding wrapper type; the compiler simply auto-boxes the primitive value into an object of the wrapper type.)

While the Java language doesn't have explicit, built-in support for it, we can implement an infinite sequence in Java—for example, by writing our own implementations of the *`Iterable`* and *`Iterator`* interfaces. An example of such an implementation is beyond the scope of this document, but we'll learn how to implement these interfaces in the bootcamp.

Length

The length of a sequence is simply the number of terms in that sequence. Of course, this is not necessarily the same as the number of *distinct* terms in the sequence, since a term may appear multiple times in the same sequence.

Implementations of the Java standard library interface *`Collection`* (which is extended by *`List`*) include the `size` method, which returns the number of items

in that collection. Since that method is declared with a return type of `int`, and since `int` is used for element indices in *`List`*, this implies that all instances of *`Collection`* are limited to a maximum of $\left(2^{31}-1\right)$, or `Integer.MAX_VALUE` elements.

If a sequence is implemented in Java code as an array, then the `length` property (not a method) of the array contains the number of elements in the sequence. As with the Collection.size(), the `length` property is an `int`; thus, arrays (and sequences implemented with arrays) are limited to a maximum of $\left(2^{31}-1\right)$, or `Integer.MAX_VALUE` elements.

Recurrence relations

Some sequences are defined (at least in part) according to a *recurrence relation* between the terms of the sequence. In this type of definition, the value of the nth term of a sequence is defined as a function of the preceding terms. Typically, this type of definition is used for infinite sequences.

Given the sequence A,

$$A = (a_0, a_1, \ldots),$$

a recurrence relation may be defined (in very general terms) as

$$a_n = f(a_0, a_1, \ldots, a_{n-1}).$$

In most cases, the function f isn't a function of *all* the preceding terms, but of a small number of terms immediately preceding a_n. Also, note that the recurrence relation usually doesn't define A completely; the definition generally includes one or more *initial values*, as well. For example, the well-known *Fibonacci* sequence may be defined as

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ , where } n > 1 \end{aligned}$$

Sum

We use the *summation* symbol, $\sum$, to denote the sum of the terms of a sequence, e.g.

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \ldots + a_n$$

In Java, if a given finite sequence is implemented with an array, or with a *List* instance—or with any instance of a class that implements the *Iterable* interface—then we can sum the terms of the sequence in several ways; one is via an *enhanced for* loop:

```
int[] fib10 = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34}; // 1st 10
                                                 // Fibonacci terms.
int sum = 0;
for (int f : fib10) {
  sum += f;
}
System.out.println(sum);                         // 88
```

Another approach, available in Java 8 and later versions, is to leverage the stream framework, e.g.

```
int[] fib10 = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
int sum = IntStream.of(fib10).reduce(0, (a, b) -> a + b);
System.out.println(sum);                         // 88
```

Product

General case

The product of terms in a sequence is denoted by $\prod$:

$$\prod_{i=1}^{n} a_i = a_1 \cdot a_2 \cdot \ldots \cdot a_n$$

Just as we can compute the sum of an array or *Iterable* instance with an enhanced `for` loop in Java, we can use the same mechanism to compute the product. (Depending on the type of values in the sequence, and the length of the sequence, we should watch out for possible overflow.)

```
int[] lookSay6 = {1, 11, 21, 1211, 111221, 312211}; // 1st 6 terms
                                                    // of look & say.
long product = 1;
for (int a : lookSay6) {
  product *= a;
}
System.out.println(product); // 9713843871995571
```

Again, one alternative approach (out of several) uses the streams framework:

```
int[] lookSay6 = {1, 11, 21, 1211, 111221, 312211};
long product = IntStream.of(lookSay6)
    .asLongStream().reduce(1, (a, b) -> a * b);
System.out.println(product); // 9713843871995571
```

Factorial

The *factorial* of a non-negative integer n, denoted by $n!$, is a special case of a sequence product, where the sequence is the positive integers less than or equal to n:

$$n! = \prod_{i=1}^{n} i$$

Following the *empty product* convention, the product of zero terms is defined to be equal to 1; therefore, $0! = 1$.

Of course, we can use iteration to compute a factorial in Java, just as we can for the general case of a sequence product; however, rather than an enhanced `for`, we would typically use a basic or traditional `for`. (There are also other techniques for computing $n!$, such as *Stirling's approximation*, but a discussion of these is beyond the scope of this document.) For example, we might compute and display 20! with the following code.

```
long product = 1;
for (int i = 1; i <= 20; i++) {
  product *= i;
}
System.out.println(product); // 2432902008176640000
```

The above is compact enough that we wouldn't gain much economy of expression by using the streams framework to compute a factorial. Nonetheless, we can use streams if we want to. (Note that the example below uses `mapToObj` to convert an `IntStream` to a `Stream<BigInteger>`; using the `BigInteger` type lets us compute 21! or higher—much higher, in fact—without overflow.)

```
BigInteger product = IntStream.rangeClosed(1, 21) // (1, ..., 21).
     .mapToObj(BigInteger::valueOf)
     .reduce(BigInteger.ONE, (a, b) -> a.multiply(b));
System.out.println(product); // 51090942171709440000
```

Sets

Definition

In contrast to a sequence, a *set* is an *unordered* collection of objects—i.e. an element of a set cannot generally be referenced by index or position, and when iterating over the contents of a set, the order of iteration may not be known in advance. Another important difference is that while a given object may appear multiple times in some sequence, this is not the case, in any meaningful way, for a set: an object is simply either an element of the set, or it is not. Adding an object to the same set multiple times has no effect after the first addition.

As with a sequence, the elements of a set are generally homogeneous. A set may also be finite (i.e. it contains a finite number of elements) or infinite.

The Java standard library includes the *`Set`* interface (extending the *`Collection`* interface), as well as a number of classes implementing that interface (e.g. `HashSet`, `EnumSet`, `TreeSet`), to provide set functionality. None of these classes implement infinite sets, but infinite sets can be implemented (with some constraints) by writing our own implementation of *`Set`*.

While most of set theory is far beyond the scope of this document, some important terms and notation conventions are defined below.

Cardinality

For a finite site, the *cardinality* (size) of the set is simply the number of elements in the set. (For an infinite set, things get a bit more interesting.)

The cardinality of a Java set can be obtained from the `size()` method.

The empty set

The *empty set*, denoted by $\emptyset$, $\varnothing$, or $\{\}$, is the set containing no elements. Formally, there is a single empty set; in a mathematical context, we usually say "*the* empty set", rather than "*an* empty set".

In Java, the `Collections.EMPTY_SET` constant refers to an immutable empty set, and we can test any instance of a *`Set`* implementation with the `isEmpty` method, to determine whether that set is empty. We can also use the `size` method (declared by the *`Collection`* interface) to get the number of elements in a set (the cardinality of the set); the cardinality of the empty set is 0.

```
Set set1 = Collections.EMPTY_SET;       // Immutable empty set.
System.out.println(set1.size());        // 0
System.out.println(set1.isEmpty());     // true
Set<Integer> set2 = new HashSet<>();    // HashSet implements set.
System.out.println(set2.size());        // 0
System.out.println(set2.isEmpty());     // true
System.out.println(set1.equals(set2)); // true
```

Membership

The central relationship constituting a set is that between the set and its members. An object is either an element of (in) a set or it is not, and in general, a set can be completely characterized by its elements. We can assert that a object is an element of a set with the $\in$ symbol, or assert that an object is not an element of a set with $\notin$:

$$
\begin{aligned}
1 &\in \mathbb{N} \\
-1 &\notin \mathbb{N}
\end{aligned}
$$

The `contains` method is used to test an object for membership in a Java set:

```
Set<Integer> set = Set.of(1, 2, 3);  // Immutable set {1, 2, 3}.
System.out.println(set.contains(2)); // true
System.out.println(set.contains(4)); // false
```

Equality

Because sets are unordered, the only thing that matters, when comparing sets for equality, is that both sets contain the same elements. In other words, 2 sets are equal if and only if every element of the first is also an element of the second, and vice versa. We might write this as

$$S = T \iff (v \in S, \forall v \in T) \land (v \in T, \forall v \in S)$$

Here, $\forall$ means "for all", and $\iff$ means "if and only if". Thus, we have "S equals T if and only if v is an element of S, for all v in T, and v is an element of T, for all v in S.

Java makes comparing sets for equality quite easy: All implementations of *`Set`* in the standard library implement the `equals` method appropriately for set comparison:

```
Set<Integer> set1 = Set.of(1, 2, 3);
Set<Integer> set2 = Set.of(3, 1, 2);
System.out.println(set1.equals(set2)); // true
Set<Integer> set3 = Set.of(1, 2, 3, 4);
System.out.println(set1.equals(set3)); // false
```

Subsets & supersets

One set is a *subset* of another if every element of the first is also an element of the second; this relationship is denoted by $\subseteq$. If there is at least one element of the second that is not an element of the first (i.e. the 2 sets aren't equal), then the first is a *proper subset* (sometimes called a *strict subset*), indicated by the $\subset$ symbol. Mathematically, we could define these relationships as

$$
\begin{aligned}
S \subseteq T &\iff v \in T, \forall v \in S \\
S \subset T &\iff (v \in T, \forall v \in S) \land (S \neq T)
\end{aligned}
$$

Note that by this definition, 2 sets that are equal are also subsets and supersets of each other, but not proper subsets or supersets of each other. Also, note that since the empty set has no elements, it trivially satisfies the conditions for being a subset of all sets, and for being a proper subset of all sets other than itself.

We can easily define *superset* (and proper superset) in terms of the subset relationship: one set is a superset (or proper supserset) of another if and only if the second is a subset (or proper subset) of the first. The symbols used here are those used for the subset relationship, with the direction reversed:

$$
\begin{aligned}
S \supseteq T &\iff T \subseteq S \\
S \supset T &\iff T \subset S
\end{aligned}
$$

The `Collection` interface of the Java standard library declares the `containsAll` method, which—when invoked on a set—tests for subset/superset relationships; combining that with the `equals` method and the logical negation operator `!` lets us test for proper subset/superset relationships:

```
Set<Integer> set1 = Set.of(1, 2, 3);
Set<Integer> set2 = Set.of(3, 1, 2);
System.out.println(set1.containsAll(set2)); // true
System.out.println(set1.containsAll(set2)
    && !set1.equals(set2)); // false (not a proper superset)
Set<Integer> set3 = Set.of(1, 2, 3, 4);
System.out.println(set3.containsAll(set1)); // true
System.out.println(set3.containsAll(set1)
    && !set3.equals(set1)); // true (a proper superset)
```

Set-builder notation

Set-builder notation is a set of conventions commonly used to describe a set. Since this notation is quite convenient when defining many set theory concepts and terms, we'll illustrate some aspects of it before going further.

- Enumeration

  The simplest type of set description in set-builder notation is *enumeration*—that is, listing the members of a set, either element by element (*roster method*), or using ellipses for commonly understood ranges of values. For example,

  $$\{2, 3, 5, 7, 11, 13, 17, 19\}$$

  is the set of prime numbers less than 20;

  $$\{1, 2, 3, \ldots\}$$

is the set of natural numbers.

Note that when using enumeration, it's common to list the elements in natural (usually ascending) order. However, this is not required; in any event, the list order has no effect, since sets are not ordered.

- Predicates

  When we use set-builder notation with a *predicate* to describe or define a set, there are 2 parts inside the braces: a variable (sometimes with another set specified, indicating the base values that the variable might take, prior to evaluating the predicate); and a *predicate*, or logical condition, that a value of the variable must satisfy, in order to be included in the set. These parts are usually separated by a vertical bar (read as "such that"). For example, the set of all even numbers can be described as

  $$\{x \mid (x \in \mathbb{Z}) \land (x \bmod 2 = 0)\}$$

  The $\land$ symbol means "and", so we can read the above (in a rather overly verbose fashion) as "the set of all x, such that x is an element of the set of all integers, and the remainder when x is divided by 2 is equal to 0."

  We might describe the same set more simply as

  $$\{x \in \mathbb{Z} \mid x \bmod 2 = 0\}$$

  We would read this as "the set of all integer x, such that the remainder when x is divided by 2 is equal to 0."

  There are many other ways we might describe the same set; one more, included here mostly to introduce a few more useful notation elements, is

  $$\{x \mid (\exists k \in \mathbb{Z})\,[x = 2k]\}$$

  Here, we read $\exists$ as "there exists", and the brackets after the parentheses are read "where". Putting it together, we have "the set of all x, such that there exists some integer k, where $x = 2k$." In other words, it is the set of all numbers that are twice the value of an integer. (We don't need to specify that x is an integer, since 2 times the value of any integer is also an integer.)

Java doesn't support set-builder notation directly; however, that's not the only way to express the underlying concepts, and there are features in the Java standard library for defining a set (or adding elements to a set) by enumeration, and for filtering sets via predicates (or as shown below, the logical inverse of a predicate—a condition under which elements are *removed* from a set). Thus, it's

usually fairly straightforward to work from a definition in set-builder notation to a Java implementation.

For example, we can use set-builder notation to describe the set of odd natural numbers under 20:

$$\{x \in \mathbb{N} \mid (x < 20) \wedge (x \bmod 2 \neq 0)\}$$

Then, we can create this set using iteration and the `removeIf` method in Java code:

```
Set<Integer> set = new HashSet<>();
for (int i = 1; i < 20; i++) { // Add 1 through 19 to the set.
  set.add(i);
}
set.removeIf(x -> x % 2 == 0); // Remove even values.
System.out.println(set); // [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Union

The union of 2 sets, denoted by $\cup$, contains only those objects that are elements of either set (or both sets). We can express this using set-builder notation (with the symbol $\vee$, meaning "or") as

$$S \cup T = \{v \mid (v \in S) \vee (v \in T)\}$$

In Java, we construct a set as the union of 2 other sets by initializing the union set with the elements of the first of other 2, and then adding to it the elements of the second via the `addAll` method; since an object cannot be included multiple times in the same set, duplicated values are ignored.

```
Set<Integer> set1 = Set.of(1, 2, 3);
Set<Integer> set2 = Set.of(1, 3, 4, 5);
Set<Integer> union = new HashSet<>(set1); // Initialize from set1.
union.addAll(set2); // Add set2 elements.
System.out.println(union); // [1, 2, 3, 4, 5]
```

Intersection

The intersection of 2 sets, denoted by $\cap$, contains only those objects that are elements of both sets. We can express this as

$$S \cap T = \{v \mid (v \in S) \wedge (v \in T)\}$$

In Java, we can construct a set as the intersection of 2 other sets by initializing the intersection set with the elements of the first of the other 2, then using the `retainAll` method to keep only those elements that are also elements of the second.

```
Set<Integer> set1 = Set.of(1, 2, 3);
Set<Integer> set2 = Set.of(1, 3, 4, 5);
Set<Integer> intersection = new HashSet<>(set1); // Initialize from set1.
intersection.retainAll(set2); // Retain those also in set2.
System.out.println(intersection); // [1, 3]
```

Difference (relative complement)

The difference between 2 sets is the set containing only those elements of the first that are not in the second. This is also called the *relative complement* of the second set in the first (i.e. the intersection of the first set and the complement of the second set), and is denoted by $\setminus$.

$$S \setminus T = \{v \in S \mid v \notin T\}$$

In Java, we can construct a set as the difference of 2 other sets by initializing the intersection set with the elements of the first of the other 2, then using `removeAll` to remove those elements that are also elements of the second.

```
Set<Integer> set1 = Set.of(1, 2, 3);
Set<Integer> set2 = Set.of(1, 3, 4, 5);
Set<Integer> diff = new HashSet<>(set1); // Initialize from set1.
diff.removeAll(set2); // Remove set2 elements.
System.out.println(diff); // [2]
```

Symmetric difference

The *symmetric difference* between 2 sets is the set containing only those objects that are elements of either—but not both—of the sets. It is denoted by the symbol $\triangle$, and can be expressed in terms of union of the differences of the 2 sets:

$$S \triangle T = (S \setminus T) \cup (T \setminus S)$$

It can also be expressed as the difference between the union and intersection of the 2 sets:

$$S \triangle T = (S \cup T) \setminus (S \cap T)$$

We can use either of the above expressions as a guide to a Java implementation. Here, we implement the second:

```
Set<Integer> set1 = Set.of(1, 2, 3);
Set<Integer> set2 = Set.of(1, 3, 4, 5);
Set<Integer> union = new HashSet<>(set1); // Initialize from set1.
union.addAll(set2); // Add set2 elements.
Set<Integer> intersection = new HashSet<>(set1); // Initialize from set1.
intersection.retainAll(set2); // Retain those also in set2.
Set<Integer> diff = new HashSet<>(union); // Initialize from union.
diff.removeAll(intersection); // Remove intersection elements.
System.out.println(diff); // [2, 4, 5]
```

Probability

Independent events

If events A and B have no effect on each other—that is, if the probability that event B occurs ($P(B)$) is not affected by the occurrence (or lack of same) of A, and vice versa, then the 2 events are *independent*. For example, if we toss a coin twice, the outcome of each toss has no effect on the outcome of the other.

If we have 2 or more independent events, then the probability that all of the events occur (their *joint probability*) is the product of their individual probabilities. That is, if

$$E_1, E_2, \dots E_n$$

are independent events, then

$$P(E_1 \cap E_2 \dots \cap E_n) = \prod_{i=1}^{n} E_i$$

Disjoint events

If events A and B are mutually exclusive—that is, if that fact that either 1 of them has occurred implies that the other cannot occur, then the 2 events are *disjoint*. For example, in a single roll of a 6-sided die, rolling a 1 implies that 2 is not rolled—at least, not in the same roll—and vice versa. Note that this does not imply that the 2 events are the only possible outcomes, but only that each one precludes the other.

If we have 2 disjoint events, then the probability that one *or* the other occur is the sum of their individual probability. This can be extended to more than 2 disjoint events as well. In symbolic terms, where

$$E_1, E_2, \ldots E_n$$

are disjoint events (that is, $P(E_i \cap E_j) = 0, \forall i, j \in \{1, 2, \ldots, n\}$),

$$P(E_1 \cup E_2 \ldots \cup E_n) = \sum_{i=1}^{n} E_i$$

Naïve probability

Related to concept of disjoint events is that of *naïve probability*. If an experiment has a finite number of equally likely disjoint outcomes, and if there are n possible outcomes, then each has a probability of $1/n$. Further, we can compute the probability the the outcome is a member of the set E by dividing the cardinality of E by the cardinality of U, the set of all possible outcomes (aka the *universe* of outcomes).

$$P(E) = \frac{|E|}{|U|}$$

For an example of a naïve probability calculation, see the royal flush example, below.

Uniform discrete probability distributions

For some random experiments, the outcomes may be characterized by numeric values, such that each possible outcome is a member of a specified finite set of values. If these values fall within a specified range, with a constant difference between each value and the value preceding and/or succeeding it, and if all of the values are equally likely, then we have a special (and very useful) case of the situation described in Naïve probability. This is a *uniform discrete probability distribution*.

An example of this type of distribution, and this type of random experiment, is found in the roll of a single fair die. In the case of a six-sided die, the possible outcomes are the values 1 through 6, each with a probability of 1/6.

A Java method that samples a value from such a distribution could be written as follows:

```
double discreteUniform(
    double lowerBound, double interval, int n, Random rng) {
  return lowerBound + rng.nextInt(n) * interval;
}
```

In this method, `lowerBound` is a parameter specifying the outcome with the minimum possible value; `interval` is the spacing between possible outcome values; `n` is the number of possible outcomes; `rng` is an instance of the `java.util.Random` class (or a subclass of that), which can be used to generate general pseudorandom values.

We might invoke the `discreteUniform` method to simulate a roll of a six-sided die using code like the following:

```java
Random rng = new Random();
int roll = discreteUniform(1, 1, 6, rng);
```

On the other hand, in this example, where the possible outcomes are all integral, it's probably simpler just to use the `Random.nextInt()` method directly. In fact, one of the most important points to remember about sampling from uniform discrete probability distributions is that the `Random.nextInt(int bound)` method is a very useful building block. That method samples from the uniform discrete distribution over the values $\{0, 1, \ldots (bound - 1)\}$; the value returned can be transformed arithmetically to sample from virtually any uniform discrete distribution.

Non-uniform discrete probability distributions

If the outcome values of some random experiment are not evenly spaced, or all such outcomes are not equally likely, but the set of values is finite, then we have a *non-uniform discrete probability distribution*. A detailed discussion of such distributions is beyond the scope of this document; however, it should be noted that some such distributions are actually combinations of uniform distributions.

For example, the sum obtained from a roll of 2 six-sided dice is non-uniform, because the possible values $ \left (2, 3, \ldots 12 \right)$ are not all equally likely; however, we can still fall back on the concept of naïve probability. For example, if we examine the values shown on the 2 dice, we see that there are a total of 36 possible rolls, all equally likely. 6 of those rolls give a sum of 7; thus, the probability of rolling a 7 is 6/36, or 1/6. On the other hand, there are only 2 possible rolls that give a sum of 3; the probability of rolling a 3 is therefore 2/36, or 1/18.

Continuous probability distributions

The sets of possible numerical outcomes for some random experiments are not finite; instead, the outcome might take any value within a continuous range. In such cases, it's usually meaningless to talk about the probability of a specific single value as an outcome; instead, we might examine the probability of an outcome falling within a stated interval.

Typically, we express such a *continuous probability distribution* as a *probability density function*, $f(x)$, with these general constraints:

$$f(x) \geq 0, \quad -\infty < x < \infty$$
$$\int_{-\infty}^{\infty} f(x)\, dx = 1$$

The probability of an outcome falling within some interval, $[X_a, X_b]$, can be expressed using a definite integral:

$$p(X_a \leq x \leq X_b) = \int_{X_a}^{X_b} f(x)\, dx$$

As is the case for discrete probability distributions, continuous distributions come in uniform and non-uniform varieties; however, the general mathematical statements above apply in both cases. In any event, an in-depth discussion of continuous probability distributions is beyond the scope of this introduction. However, we should note that just as `Random.nextInt()` is an important building block for sampling from uniform discrete distributions, `Random.nextDouble()` is a similarly useful building block for sampling from uniform or non-uniform continuous distributions; for the special case of the *Gaussian distribution* (aka *normal distribution*), `Random.nextGaussian()` is even more directly applicable.

Combinations & permutations

Many computational problems involve *enumerating* (counting) the number of *combinations* of k objects that can be selected from a set of n objects. For example, we might be counting the number of 5-card hands that can be dealt from a deck of 52 standard playing cards. Note that typically, we're assuming that each hand has no effect on subsequent hands—in other words, we're counting the number of distinct hands that might be obtained when each is drawn from a full (and re-shuffled) deck. Also, we're intentionally ignoring the order of the cards in the hand itself. So the hand {2 , 5 , 10 , J , A } is not considered distinct from {5 , 2 , J , 10 , A }.

The number of distinct combinations (without regard to order) of k objects that can be selected from a set of n objects is denoted as $C\,(n,k)$, C_k^n, ${}_nC_k$, or $\binom{n}{k}$, and its value is

$$C\,(n,k) = \frac{n!}{k!\,(n-k)!} \quad (7)$$

Another way to express (7), useful for some computations, is

$$
\begin{aligned}
C(n,k) &= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k} \\
&= \prod_{i=1}^{k} \frac{n-i+1}{i}
\end{aligned}
$$

Or, equivalently:

$$
C(n,k) = \prod_{i=0}^{k-1} \frac{n-i}{i+1} \quad (8)
$$

For example, we can compute the number of distinct 5-card poker hands with the following Java code, implementing (8):

```
int hands = 1;
for (int i = 0; i < 5; i++) {
  hands *= 52 - i;
  hands /= i + 1;
}
System.out.println(hands); // 2598960
```

(See Factorial for more information on the $n!$ notation convention, and on computing the factorial in Java.)

If all combinations are equally likely—as is the case by definition after a fair shuffle of a deck of cards (for example)—then the probability of a given outcome is simply the total number of combinations that include that outcome, divided by the total number of combinations (see Naïve probability, above). For example, there are 4 "royal flush" poker hands, and a total of 2,598,960 possible 5-card hands; therefore the probability of being dealt a royal flush from a deck after a fair shuffle is 4 / 2,598,960   0.00000153907.

If the order of the k objects is significant, then the number of *permutations* (distinct orderings) of the k selected objects is $k!$. Therefore, the number of permutations of k objects selected from a set of n objects (denoted as $P(n,k)$, P_k^n, or ${}_nP_k$) is

$$
\begin{aligned}
P(n,k) &= \frac{n!}{k!\,(n-k)!} \cdot k! \\
&= \frac{n!}{(n-k)!}
\end{aligned}
$$

Trigonometry

Right triangles

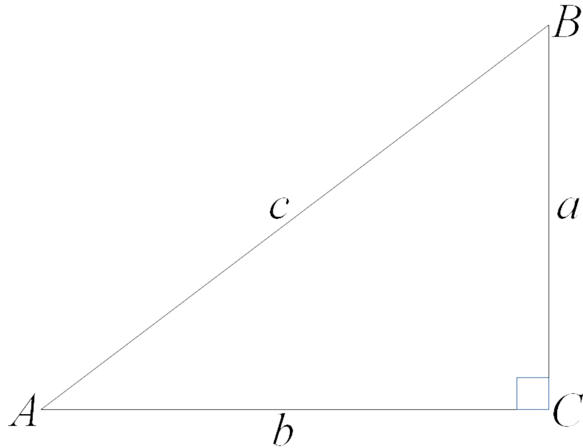

Figure 1: Right triangle used to define basic trigonometric relationships.

Trigonometric functions are initially defined in terms of the angles and sides in a right triangle, for acute angles only—i.e. in the interval $[0, \pi/2)$. By convention, each vertex is identified by an upper-case letter (most commonly, A, B, and C, for the two acute angles and the right angle, respectively), with the side opposite identified by the same letter, but lower-case. Note that an upper-case letter denotes not only a vertex, but the measure of the angle at that vertex; similarly, a lower-case letter refers not only to the side itself, but to the length of that side. Finally, the right angle is identified by a small square at vertex C.

All of the definitions in the following table are in reference to the triangle in figure 1, above.

Measure	Definition	Java method
Sine	$\sin A = \frac{a}{c}$	`double Math.sin(double angle)`
Cosine	$\cos A = \frac{b}{c}$	`double Math.cos(double angle)`
Tangent	$\tan A = \frac{a}{b}$	`double Math.tan(double angle)`
Arcsine	$\arcsin \frac{a}{c} = A$	`double Math.asin(double ratio)`
Arccosine	$\arccos \frac{b}{c} = A$	`double Math.acos(double ratio)`
Arctangent	$\arctan \frac{a}{b} = A$	`double Math.atan(double ratio)`
Pythagorean theorem	$c = \sqrt{a^2 + b^2}$	`double Math.hypot(double a, double b)`

Polar-Cartesian relationship

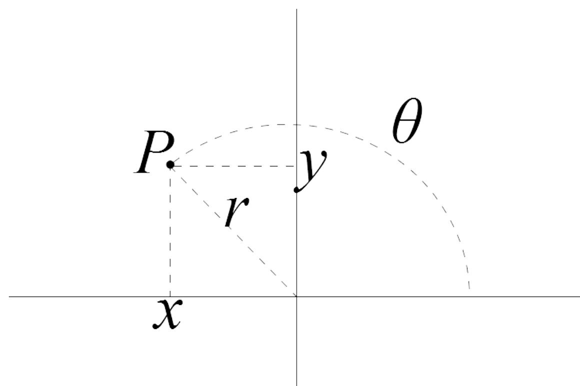

Figure 2: Position of point P shown in polar and Cartesian coordinates.

The position of a point P on a plane is usually expressed either with respect to the X and Y axes—that is, in Cartesian coordinates—or with respect to a *pole* (coincident with the origin of the Cartesian coordinate system) and a *polar axis* (coincident with the positive X axis of the Cartesian coordinate system). The coordinates used in the latter case are called *polar coordinates*, and consist of a distance, r, measured from the pole, and an angle, θ, measured counter-clockwise from the polar axis. When θ is in the interval $[0, \pi/2)$, the x, y, and r values form a right triangle, where θ is the angle opposite y. Recognizing this, we can extend the trigonometric relationships beyond acute angles in an intuitive fashion, by expressing them in terms of polar and Cartesian coordinates, allowing x and y to take negative values, and allowing θ to take values outside the interval $[0, \pi/2)$.

All of the definitions in the following table are in reference to the polar and Cartesian coordinate system used in the example shown in figure 2, above.

Measure	Definition	Java method
Sine	$\sin\theta = \frac{y}{r}$	`double Math.sin(double angle)`
Cosine	$\cos\theta = \frac{x}{r}$	`double Math.cos(double angle)`
Tangent	$\tan\theta = \frac{y}{x}$	`double Math.tan(double angle)`
Arcsine	$\arcsin\frac{y}{r} = \theta$	`double Math.asin(double ratio)`
Arccosine	$\arccos\frac{x}{r} = \theta$	`double Math.acos(double ratio)`
Arctangent	$\arctan\frac{y}{x} = \theta$	`double Math.atan2(double y, double x)`
Pythagorean theorem	$r = \sqrt{x^2 + y^2}$	`double Math.hypot(double x, double y)`

Note that the tangent function has a period of π, instead of 2π (the period of

the sine and cosine functions). Thus, in order to properly distinguish between θ values across the full $[0, 2\pi)$ interval, the `Math.atan2` method takes both `y` and `x` as parameters, rather than just the ratio of the two. This also gives us a clean way to evaluate the arctangent in cases where the ratio is not finite (e.g. for $\theta = \pi/2$ and $\theta = 3\pi/2$).