

1.11 Math Class

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Explain why the Math class can be used in any program without an import line or an object.
- Use Math.abs, Math.pow, and Math.sqrt in expressions and predict the value and type each call produces.
- Combine several Math calls in one expression and evaluate it from the inside out.
- Describe exactly which values Math.random() can return, including which endpoint is possible.
- Turn Math.random() into a random double or a random int in a stated range, and check each endpoint.

What to remember from earlier

- An operation with at least one double gives a double. ([Topic 1.3: Expressions and Output, 1.3.5 Arithmetic Expressions](#))
- Casting with (int) drops the decimal part, and (int) (x + 0.5) rounds a nonnegative double. ([Topic 1.5: Casting and Range of Variables](#))
- A non-void method's return value must be stored or used in an expression. ([Topic 1.9: Method Signatures, 1.9.7 Return Values](#))
- Methods with the same name but different parameter types are overloaded. ([Topic 1.9: Method Signatures, 1.9.8 Overloaded Methods](#))
- A class method has static in its header and is called with the class name, as in Math.sqrt(25). ([Topic 1.10: Calling Class Methods, 1.10.2 Calling with the Class Name and a Dot](#))

1.11.1 The Math Class Is Always Available

Suppose a program knows the area of a square room, 144 square feet, and needs the length of one wall. You need the square root of 144. With only the operators `+`, `-`, `*`, `/`, and `%`, there is no clean way to get it. You could try guesses one after another, but that is a whole algorithm just to answer one small math question.

Java already has that algorithm written for you. In Topic 1.7 you learned that a **library** is a collection of ready-made classes, grouped into **packages**, and that API documentation tells you how to use them. One of those classes is `Math`. It holds methods for common calculations: absolute value, powers, square roots, and random numbers.

No import line needed

In Topic 1.4, using `Scanner` meant writing an `import` line at the top of the file so Java knew where to find that class. `Math` is different. It lives in a package named `java.lang`, and every class in `java.lang` is **available by default**: Java makes it visible to every program automatically. That is why you never write an import line for `Math` (or for `System`, which is also in `java.lang`). The classes

in `java.lang` are the ones nearly every program needs, so Java saves you from importing them every time.

Only class methods

The second fact about `Math` is that it contains **only class methods**. In Topic 1.10 you saw that a class method has `static` in its header and belongs to the class itself, so you call it with the class name, a dot, and the method name. There is nothing to set up first: you write `Math.sqrt(144.0)` and get an answer. This fits what `Math` does. Finding a square root needs only the number you hand it, so there is no reason to call it on anything other than the class.

Here is the complete set of `Math` methods you need for this course, read from their headers. (Read past `static` here as Topic 1.10 taught: it marks a class method.)

Method header	What it returns
<code>static int abs(int x)</code>	The absolute value of an <code>int</code>
<code>static double abs(double x)</code>	The absolute value of a <code>double</code>
<code>static double pow(double base, double exponent)</code>	base raised to the power exponent
<code>static double sqrt(double x)</code>	The nonnegative square root of <code>x</code>
<code>static double random()</code>	A <code>double</code> from 0.0 (included) up to 1.0 (not included)

Each one gets its own subsection below.

KEY IDEA

`Math` is in the `java.lang` package, so every program can use it without an import line. It contains only class methods, so you always call them through the class name: `Math.methodName(arguments)`.

WORKED EXAMPLE Finding a wall length

Look for three things in this program: there is no import line, the call starts with `Math.`, and the returned value is stored in a variable before it is used.

`SquareTiles.java`

```

1 public class SquareTiles
2 {
3     public static void main(String[] args)
4     {
5         double roomArea = 144.0;
6         double side = Math.sqrt(roomArea);
7         System.out.print("Side length: ");
8         System.out.println(side);
9         System.out.print("Perimeter: ");
10        System.out.println(4 * side);
11    }
12 }
```

Step by step:

1. Line 5 stores 144.0 in `roomArea`.

- Line 6 evaluates the right side first. `Math.sqrt(roomArea)` passes a copy of `144.0` to the method (call by value, Topic 1.9). The method returns `12.0`, and that value is stored in `side`.
- Lines 7 and 8 print the label and then `12.0`.
- Line 10 evaluates `4 * side`. One operand is a `double`, so the result is the `double 48.0`, which is printed.

```
Side length: 12.0
Perimeter: 48.0
```

Notice what line 10 shows: once `Math.sqrt` has returned, its result is an ordinary `double` that you can use in any expression. You could also skip the variable and write `4 * Math.sqrt(roomArea)` directly. Java would call the method first, then multiply what it returned by 4, and print the same `48.0`. A method call can sit anywhere a value of its return type can sit.

WATCH OUT

Leaving off `Math`. Students sometimes write the method name alone, as if `sqrt` were a built-in word:

`MissingClassName.java`

```
1 public class MissingClassName
2 {
3     public static void main(String[] args)
4     {
5         double roomArea = 144.0;
6         double side = sqrt(roomArea);
7         System.out.println(side);
8     }
9 }
```

```
MissingClassName.java:6: error: cannot find symbol
    symbol:   method sqrt(double)
    location: class MissingClassName
```

Topic 1.10 explained when the class name is optional: only when the call is inside the class that defines the method. This call is inside `MissingClassName`, which has no method named `sqrt`, so Java searches that class, finds nothing, and reports `cannot find symbol`. Being "available by default" means the `Math` class is visible without an import; it does not mean its methods can be called without the class name. The fix is `Math.sqrt(roomArea)`.

PAUSE AND TRY

Question 1 Multiple choice

Which statement about the `Math` class is true?

- A program must include an import line before it can call `Math.sqrt`.
- `Math` is in `java.lang`, so its methods can be called through the class name with no import line.

- C. Because `Math` is available by default, its methods can be called without writing `Math.` in front.
- D. Some `Math` methods are class methods and others must be called a different way.

Question 2 Debugging

This program does not compile. Find the error, write the corrected line, and explain what the compiler reports.

LowercaseMath.java

```
1 public class LowercaseMath
2 {
3     public static void main(String[] args)
4     {
5         double gardenArea = 49.0;
6         double side = math.sqrt(gardenArea);
7         System.out.println(side);
8     }
9 }
```

1.11.2 Absolute Value

A guessing game picks the number 50. One player guesses 42 and another guesses 61. Who was closer? Subtracting gives $42 - 50 = -8$ and $61 - 50 = 11$. The minus sign only tells you which side of the target the guess was on; for "how close," you want the distance, which is never negative: 8 and 11.

In plain words, you want the number without its sign. That is the **absolute value**: a number's distance from zero. The absolute value of -8 is 8, the absolute value of 11 is 11, and the absolute value of 0 is 0. `Math.abs` computes it.

Two versions with the same name

The `Math` documentation lists two `abs` headers:

```
1 static int abs(int x)
2 static double abs(double x)
```

These are **overloaded** methods (Topic 1.9): the same name with different parameter types. Java picks the version whose parameter type matches the argument you pass. An `int` argument calls the `int` version, which returns an `int`. A `double` argument calls the `double` version, which returns a `double`. So the type you get back matches the type you put in. Why have two? If there were only a `double` version, `Math.abs(-8)` would hand back `8.0`, and you could not store it in an `int` variable without a cast.

KEY IDEA

`Math.abs(x)` returns the distance of `x` from zero, which is never negative. An `int` argument gives an `int` result; a `double` argument gives a `double` result.

WORKED EXAMPLE How far off was the guess?

Watch the expression inside the parentheses: Java evaluates it first, and only its value is passed to `Math.abs`.

```
1 int target = 50;
2 int guess = 42;
3 int distance = Math.abs(guess - target);
4 System.out.println(distance);
5 guess = 61;
6 distance = Math.abs(guess - target);
7 System.out.println(distance);
```

Line	Expression evaluated	Value passed to <code>abs</code>	<code>distance</code>	Printed
3	<code>guess - target = 42 - 50</code>	-8	8	
4			8	8
6	<code>guess - target = 61 - 50</code>	11	11	
7			11	11

On line 3, the argument is an `int` expression, so the `int` version of `abs` runs and returns the `int` 8. On line 6, the argument is already positive, so `abs` returns it unchanged. The guess of 42 was closer. Notice that the order of the subtraction does not matter: `Math.abs(target - guess)` gives the same distances.

Now change one thing: the type of the argument.

```
1 System.out.println(Math.abs(-4));
2 System.out.println(Math.abs(-4.0));
3 System.out.println(Math.abs(-3.5));
4 System.out.println(Math.abs(6));
```

```
4
4.0
3.5
6
```

`-4` is an `int` literal, so the `int` version returns 4. `-4.0` is a `double` literal, so the `double` version returns 4.0; the value is the same distance, but the type is `double`, and Java prints a `double` with a decimal point. The last line shows that a positive argument comes back unchanged.

WATCH OUT

Expecting `Math.abs` to change the variable. A student wants to remove the sign from `change`:

```
1 int change = -15;
2 Math.abs(change);
3 System.out.println(change);
```

It compiles and prints `-15`. Two ideas from Topic 1.9 explain why. First, call by value: `abs` receives a copy of `-15`, so it has no way to change `change` itself. Second, `abs` returns 15, but

the program never stores or uses that return value, so it is thrown away. Store the result back into the variable:

```
1 int change = -15;
2 change = Math.abs(change);
3 System.out.println(change);
```

This prints 15. The same rule holds for every `Math` method in this chapter: each one returns a new value and leaves its arguments alone.

PAUSE AND TRY

Question 3 Tracing

What does this code print?

```
1 int low = -8;
2 int high = 5;
3 System.out.println(Math.abs(low));
4 System.out.println(Math.abs(high - low));
5 System.out.println(Math.abs(low + high));
6 System.out.println(Math.abs(low / 2.0));
```

Question 4 Debugging

This program should print the gap between `start` and `end`, but it does not compile. Explain why and give a corrected line.

GapWrong.java

```
1 public class GapWrong
2 {
3     public static void main(String[] args)
4     {
5         double start = 5.5;
6         int end = 2;
7         int gap = Math.abs(end - start);
8         System.out.println(gap);
9     }
10 }
```

1.11.3 Raising to a Power

A square with sides of 7 has area 7^2 . A savings account might grow by a factor each year for 10 years, which means raising that factor to the 10th power. For a small whole-number exponent you can multiply by hand, side * side, but $2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$ is long and easy to miscount, and you cannot write a repeated multiplication at all when the exponent is stored in a variable whose value you do not know in advance.

In plain words, you want "this number multiplied by itself that many times." The number being multiplied is the **base**, and the count is the **exponent**. `Math.pow` computes it:

```
1 static double pow(double base, double exponent)
```

Take the header apart:

Part	What it tells you
<code>double</code> (first)	The return type. <code>pow</code> always returns a <code>double</code> , even for whole-number answers.
<code>pow</code>	The method name.
<code>double base</code>	The first argument is the base.
<code>double exponent</code>	The second argument is the exponent.

Both parameters are `double`, but you may pass `int` values: Java widens an `int` argument to a `double` automatically (Topic 1.5), so `Math.pow(2, 10)` works exactly like `Math.pow(2.0, 10.0)`.

KEY IDEA

`Math.pow(base, exponent)` returns `base` raised to `exponent`, always as a `double`. The first argument is the base and the second is the exponent.

WORKED EXAMPLE Powers of two and a square area

Look at the type of each printed value.

```
1 System.out.println(Math.pow(2, 10));
2 System.out.println(Math.pow(2.5, 2));
```

```
1024.0
6.25
```

- Line 1: the `int` arguments 2 and 10 are widened to 2.0 and 10.0. The method computes 2 multiplied by itself 10 times, 1024, and returns it as the `double` 1024.0. That is why a .0 appears, even though 1024 is a whole number.
- Line 2: a square tile with side 2.5 has area $2.5 \times 2.5 = 6.25$, returned as 6.25.

Now change one thing: swap the two arguments.

```
1 System.out.println(Math.pow(2, 3));
2 System.out.println(Math.pow(3, 2));
```

```
8.0
9.0
```

`Math.pow(2, 3)` is $2 \times 2 \times 2 = 8$, while `Math.pow(3, 2)` is $3 \times 3 = 9$. As Topic 1.9 taught, arguments are matched to parameters by position, so the first is always the base. Swapping them asks a different question. (A warning for anyone who has used a calculator: Java has no power operator. The `^` symbol means something else in Java and is not part of this course, so `2 ^ 3` does not compute 8. Use `Math.pow`.)

WATCH OUT

Storing a pow result in an int. The area of a 7-by-7 square is a whole number, so a student writes:

PowToInt.java

```
1 public class PowToInt
2 {
3     public static void main(String[] args)
4     {
5         int side = 7;
6         int area = Math.pow(side, 2);
7         System.out.println(area);
8     }
9 }
```

PowToInt.java:6: error: incompatible types: possible lossy conversion from double to int

The answer happens to be whole, but the compiler goes by the header, not the value: `pow` returns a `double`, and Java will not drop a `double`'s decimal digits into an `int` without being told to. Either keep the result as a `double`, or cast it on purpose:

```
1 int side = 7;
2 double area = Math.pow(side, 2);
3 System.out.println(area);
4 int wholeArea = (int) Math.pow(side, 2);
5 System.out.println(wholeArea);
```

49.0
49

The cast `(int)` applies to the value `Math.pow` returned, `49.0`, and truncates it to `49` (Topic 1.5).

PAUSE AND TRY**Question 5** Multiple choice

What is printed by `System.out.println(Math.pow(3, 2) + Math.pow(2, 3));`?

- A. 17
- B. 17.0
- C. 12.0
- D. 18.0

Question 6 Short answer

A classmate expects `System.out.println(Math.pow(5, 2));` to print 25 and is surprised by 25.0. Explain the output, using the method header.

1.11.4 Square Roots

Going the other way is just as common. If a square garden has an area of 20.25 square meters, how long is each side? You need a number that, multiplied by itself, gives 20.25. That number is the **square root**. Every positive number has two numbers that square to it (4.5×4.5 and -4.5×-4.5 both give 20.25), but a length cannot be negative, so `Math.sqrt` always gives the **nonnegative** one: zero or positive, never negative.

```
1 static double sqrt(double x)
```

The header says: one `double` parameter (an `int` argument is widened), and a `double` return value. Treat a nonnegative argument as a precondition, the idea from Topic 1.8: no real number multiplied by itself gives a negative result, so a negative argument has no square root to return. As Topic 1.8 explained, a method does not promise to check its precondition for you. If you pass a negative number, you do not get a usable answer (what Java returns in that case is outside this course), so making sure the argument is nonnegative is your job.

KEY IDEA

`Math.sqrt(x)` returns the nonnegative square root of `x` as a `double`. The argument should not be negative.

WORKED EXAMPLE Side of a garden, then length of a ladder

First, the garden:

```
1 double gardenArea = 20.25;
2 double side = Math.sqrt(gardenArea);
3 System.out.println(side);
```

`Math.sqrt(20.25)` returns 4.5, because $4.5 \times 4.5 = 20.25$, and the program prints 4.5.

Now a harder question. A ladder leans against a wall. Its foot is 6 feet from the wall, and its top touches the wall 8 feet up. The ladder, the wall, and the ground form a right triangle, and the length of the ladder is the square root of ($6^2 + 8^2$). That needs `pow` and `sqrt` in one expression:

```
1 double base = 6.0;
2 double height = 8.0;
3 double ladder = Math.sqrt(Math.pow(base, 2) + Math.pow(height, 2));
4 System.out.println(ladder);
```

When one call is the argument of another, Java evaluates from the **inside out**: an argument must have a value before the method that receives it can run.

Step	What Java evaluates	Result
1	<code>Math.pow(base, 2) = 6.0²</code>	36.0
2	<code>Math.pow(height, 2) = 8.0²</code>	64.0
3	<code>36.0 + 64.0</code>	100.0
4	<code>Math.sqrt(100.0)</code>	10.0

The value 10.0 is stored in `ladder` and printed: the ladder is 10 feet long.

Now change one thing: use a number that is not a perfect square, such as 44.

```
1 double root = Math.sqrt(44);
2 System.out.println(root);
3 System.out.println((int) root);
4 System.out.println((int) (root + 0.5));
```

```
6.6332495807108
```

```
6
```

```
7
```

The square root of 44 is not a whole number, so `sqrt` returns a long decimal. The second line casts it to `int`, which truncates to 6. The third line uses the rounding pattern from Topic 1.5, `(int) (x + 0.5)`: $6.633 + 0.5 = 7.133$, which truncates to 7, the nearest whole number. Results from `Math` methods combine with casting exactly like any other `double`.

WATCH OUT

Taking square roots one piece at a time. A student computes the ladder length as `sqrt(62) + sqrt(82)`:

```
1 double base = 6.0;
2 double height = 8.0;
3 double ladder = Math.sqrt(Math.pow(base, 2)) + Math.sqrt(Math.pow(height, 2));
4 System.out.println(ladder);
```

It compiles and prints 14.0, not 10.0. The square root of a sum is not the sum of the square roots: $\sqrt{36 + 64} = \sqrt{100} = 10$, but $\sqrt{36} + \sqrt{64} = 6 + 8 = 14$. This is a logic error (Topic 1.1); the compiler cannot catch it because every call is legal. Put the whole sum inside a single `Math.sqrt( ... )`, as in the worked example, and test with a case you can check by hand, such as 6, 8, and 10.

PAUSE AND TRY

Question 7 Tracing

What does this code print? Show the inside-out steps for line 1.

```
1 double a = Math.sqrt(Math.pow(5, 2) - Math.pow(3, 2));
2 System.out.println(a);
3 double b = Math.sqrt(Math.abs(9 - 25));
4 System.out.println(b);
5 System.out.println(Math.sqrt(9) + Math.sqrt(16));
```

Question 8 Multiple choice

What does `Math.sqrt(36)` return?

- A. 6
- B. 6.0
- C. -6.0

D. Both 6.0 and -6.0

1.11.5 Random Numbers

Games roll dice, quizzes shuffle questions, and simulations model events that cannot be predicted, such as the weather. Everything you have written so far produces the same output every time it runs. To get unpredictable behavior, a program needs a source of **random numbers**: values that cannot be predicted ahead of time.

Math provides one method for this:

```
1 static double random()
```

The header has an empty parameter list, so the call takes no arguments: `Math.random()`. It returns a `double`. Each call returns a new value that you cannot predict, chosen from this **range** (the set of values that are possible):

- the smallest possible value is 0.0, and 0.0 itself **can** be returned;
- every return value is **less than** 1.0, and 1.0 itself **cannot** be returned.

Each end of a range is called an **endpoint**. An endpoint is **inclusive** when the value is part of the range (it can actually occur) and **exclusive** when it is not. So `Math.random()` returns a `double` from 0.0 inclusive up to 1.0 exclusive. Possible results include 0.0, 0.25, 0.5, and 0.9999, but never 1.0, never a negative number, and never anything bigger.

KEY IDEA

`Math.random()` takes no arguments and returns an unpredictable `double` that is at least 0.0 and less than 1.0. The 0.0 endpoint is inclusive; the 1.0 endpoint is exclusive.

Tracing code that uses random numbers

You cannot trace `Math.random()` by predicting its value. Instead, ask "what if?" Pick a value the call *could* return, such as 0.5, and substitute it into the code. In this chapter, a value used this way is called a **stand-in**. (It is our name for the technique, not an official term.) Checking the smallest stand-in, 0.0, and a value just under 1.0 tells you the whole range of possible results.

WORKED EXAMPLE One call, one value

Each call to `Math.random()` produces its own value. Storing the result in a variable lets you use one random value more than once.

```
1 double r = Math.random();
2 System.out.println(r - r);
```

This always prints 0.0. Line 1 calls `Math.random()` once and stores whatever it returned in `r`. Line 2 does not call the method again; it reads `r` twice. Suppose the call returned the stand-in 0.42: then line 2 computes 0.42 - 0.42, which is 0.0. Whatever the stand-in, a value minus itself is 0.0, so the output is predictable even though `r` is not.

Now change one thing: write `Math.random() - Math.random()` instead. That expression makes **two** calls, and each call returns its own unpredictable value, so the difference is almost never `0.0` and changes from run to run. When a program needs the same random value in two places (for example, the number rolled on a die), store it in a variable once.

WATCH OUT

Passing the range as an argument. Students who want a number up to 6 often write `Math.random(6)`:

RandomArgument.java

```
1 public class RandomArgument
2 {
3     public static void main(String[] args)
4     {
5         double roll = Math.random(6);
6         System.out.println(roll);
7     }
8 }
```

```
RandomArgument.java:5: error: method random in class Math cannot be applied to given
types;
    required: no arguments
    found:    int
```

The header `random()` has an empty parameter list, and Topic 1.9 taught that arguments must match the parameter list in number and type. There is no version of `random` that takes a number, so the compiler rejects the call. You choose the range with arithmetic *after* the call, as the next two subsections show.

PAUSE AND TRY

Question 9 Multiple choice

Which of these values could `Math.random()` return?

- A. `1.0`
- B. `0.0`
- C. `-0.3`
- D. `1`

1.11.6 Random Decimal Values in a Range

A weather simulation needs a random temperature from 60.0 up to 75.0 degrees. `Math.random()` only gives values from 0.0 up to 1.0. The values you get are useful only after you stretch and slide that small range to the one you need, and that is done with ordinary arithmetic from Topic 1.3.

In plain words, there are two moves:

1. **Stretch** with multiplication. Multiplying every possible value by 15 turns the range "0.0 up to 1.0" into "0.0 up to 15.0." The width of the range becomes 15.
2. **Slide** with addition. Adding 60 to every possible value moves "0.0 up to 15.0" to "60.0 up to 75.0." The width stays 15; only the position changes.

The endpoints keep their kind. `0.0` is a possible result of `Math.random()`, so `0.0 * 15 + 60 = 60.0` is a possible result: the low endpoint stays inclusive. `1.0` is never returned, so `75.0` is never reached: the high endpoint stays exclusive.

KEY IDEA

`Math.random() * width + low` gives a random `double` from `low` (inclusive) up to `low + width` (exclusive). Multiply to set the width, then add to set the starting point.

WORKED EXAMPLE A random temperature from 60.0 up to 75.0

The expression is `Math.random() * 15 + 60`. To see what it can produce, use stand-ins: the program below replaces the call with a variable `r` and tries three possible values.

```
1 double r = 0.0;
2 System.out.println(r * 15 + 60);
3 r = 0.5;
4 System.out.println(r * 15 + 60);
5 r = 0.75;
6 System.out.println(r * 15 + 60);
```

```
60.0
67.5
71.25
```

Stand-in <code>r</code>	<code>r * 15</code> (done first)	+ 60
0.0	0.0	60.0
0.5	7.5	67.5
0.75	11.25	71.25
just under 1.0	just under 15.0	just under 75.0

Multiplication happens before addition (precedence, Topic 1.3), so each row stretches first, then slides. The smallest possible result is `60.0`, and the results creep up toward `75.0` without reaching it. So `Math.random() * 15 + 60` gives a `double` from 60.0 inclusive up to 75.0 exclusive.

To write the expression for any range, find the **width** first: high minus low. For 60.0 up to 75.0, the width is $75 - 60 = 15$.

Now change one thing: a range that starts below zero. A game needs a random wind speed from -1.0 up to 1.0 . The width is $1 - (-1) = 2$, and the low end is -1 , so "adding the low end" means adding -1 , which is subtracting 1: `Math.random() * 2 - 1`.

```

1 double r = 0.0;
2 System.out.println(r * 2 - 1);
3 r = 0.25;
4 System.out.println(r * 2 - 1);
5 r = 0.75;
6 System.out.println(r * 2 - 1);

```

```

-1.0
-0.5
0.5

```

With stand-ins 0.0, 0.25, and 0.75, the results are -1.0 , -0.5 , and 0.5 . The low endpoint -1.0 is inclusive, and the results approach 1.0 without reaching it.

WATCH OUT

Writing the slide before the stretch. A student writes `Math.random() + 60 * 15`, intending "random, then 60 to 75." Precedence makes Java compute `60 * 15` first:

```

1 double r = 0.5;
2 System.out.println(r + 60 * 15);
3 System.out.println(r * 15 + 60);

```

With the stand-in `0.5`, the first line gives `900.5`, because Java computed `0.5 + 900`. That expression ranges from `900.0` up to `901.0`, nowhere near the temperatures intended. The second line is the correct `r * 15 + 60`, giving `67.5`. Multiply the random value by the width, then add the low end. When in doubt, check with the stand-in `0.0`: the result should be the low end of your range.

PAUSE AND TRY

Question 10 Multiple choice

What values can `Math.random() * 4 + 2` produce?

- A. From 0.0 inclusive up to 4.0 exclusive
- B. From 2.0 inclusive up to 6.0 exclusive
- C. From 2.0 inclusive up to 6.0 inclusive
- D. From 2.0 inclusive up to 4.0 exclusive

Question 11 Multiple choice

A program needs a random `double` from 50.0 inclusive up to 100.0 exclusive. Which expression produces it?

- A. `Math.random() * 100 + 50`
- B. `Math.random() * 50 + 50`
- C. `Math.random() + 50 * 50`
- D. `Math.random() * 50`

1.11.7 Random Integers in a Range

A die shows a whole number from 1 to 6, not 3.72. Many random choices need an `int`: a die roll, a card number, a random question number. You already have the two tools: `Math.random()` with arithmetic to stretch and slide, and the `(int)` cast from Topic 1.5, which truncates a `double` by chopping off everything after the decimal point.

Stretch, cast, then slide

Start with six possible results, numbered 0 through 5:

1. `Math.random() * 6` gives a `double` from 0.0 up to 6.0 (exclusive).
2. `(int) (Math.random() * 6)` truncates it. Anything from 0.0 up to 1.0 becomes 0, anything from 1.0 up to 2.0 becomes 1, and so on up to anything from 5.0 up to 6.0, which becomes 5. Since 6.0 is never reached, 6 is impossible. The results are the six `int` values 0, 1, 2, 3, 4, 5.
3. Adding 1 slides them to 1, 2, 3, 4, 5, 6.

Notice what the cast does to the endpoints. In the `double` range, 6.0 was an exclusive endpoint. After truncating, the `int` results run from 0 to 5 with **both** endpoints inclusive: 0 and 5 really can occur. When you describe an `int` range, say which whole numbers are possible.

The general pattern for a random `int` from `low` to `high`, both inclusive, is:

```
1 (int) (Math.random() * (high - low + 1)) + low
```

Why `high - low + 1`? It is the **count** of whole numbers in the range, and the multiplier must equal the count, because truncating `Math.random() * count` produces exactly `count` different values, 0 through `count - 1`. From 1 to 6 the count is `6 - 1 + 1 = 6`. Without the `+ 1`, you would get one value too few and the high end could never occur.

KEY IDEA

`(int) (Math.random() * count) + low` gives a random `int` from `low` to `low + count - 1`, both inclusive. For a range from `low` to `high` inclusive, the count is `high - low + 1`.

WORKED EXAMPLE Rolling a die

The expression is `(int) (Math.random() * 6) + 1`. The program uses four stand-ins for the random value and prints `r * 6` and the final roll for each.

```
1 double r = 0.0;
2 System.out.println(r * 6);
3 System.out.println((int) (r * 6) + 1);
4 r = 0.25;
5 System.out.println(r * 6);
6 System.out.println((int) (r * 6) + 1);
7 r = 0.5;
8 System.out.println(r * 6);
9 System.out.println((int) (r * 6) + 1);
10 r = 0.875;
11 System.out.println(r * 6);
12 System.out.println((int) (r * 6) + 1);
```

```
0.0
1
1.5
2
3.0
4
5.25
6
```

Stand-in r	r * 6	(int) truncates to	+ 1
0.0	0.0	0	1
0.25	1.5	1	2
0.5	3.0	3	4
0.875	5.25	5	6

The order of operations matters here. The parentheses around `Math.random() * 6` make the multiplication happen first; the cast then truncates the product; the `+ 1` happens last. The smallest stand-in gives the smallest roll, 1, and values close to 1.0 give 6. Every whole number from 1 through 6 can come up.

Now change the range: a random `int` from 10 to 20, both inclusive. The count is $20 - 10 + 1 = 11$, so the expression is `(int) (Math.random() * 11) + 10`. Check the high end with the stand-in 0.999, a value close to 1.0, and compare the version that forgets the `+ 1`:

```
1 double r = 0.999;
2 System.out.println((int) (r * 11) + 10);
3 System.out.println((int) (r * 10) + 10);
```

```
20
19
```

With `* 11`, $0.999 \times 11 = 10.989$, which truncates to 10, plus 10 is 20: the high end is reachable. With `* 10`, $0.999 \times 10 = 9.99$, which truncates to 9, plus 10 is 19, and no stand-in can ever produce 20. That version gives 10 through 19 only.

Both endpoints were inclusive in that problem. When a problem says "from 10 up to but not including 20," the high endpoint is exclusive, so the possible values are 10 through 19 and the count is 10. Always decide first which whole numbers are allowed, then count them.

WATCH OUT

Casting too early. Leaving out the parentheses around the multiplication looks harmless:

```
1 int roll = (int) Math.random() * 6 + 1;
2 System.out.println(roll);
```

This prints 1 every time it runs. A cast applies only to the value immediately after it, so `(int) Math.random()` is done first. `Math.random()` is always less than 1.0, so truncating it

always gives 0. Then $0 \times 6 + 1 = 1$. The program compiles and runs, but the "die" has only one face: a logic error. Put parentheses around the whole product:

```
(int) (Math.random() * 6) + 1.
```

PAUSE AND TRY

Question 12 Multiple choice

Which expression produces a random `int` from 5 to 12, both inclusive?

- A. `(int) (Math.random() * 12) + 5`
- B. `(int) (Math.random() * 7) + 5`
- C. `(int) (Math.random() * 8) + 5`
- D. `(int) Math.random() * 8 + 5`

Question 13 Debugging

A raffle has tickets numbered 1 through 50. This line is supposed to pick a winning ticket, but it has a logic error. What values can it produce, what is wrong, and what is the corrected line?

```
1 int winner = (int) (Math.random() * 50);
```

Putting it together

A treasure-hunt game hides treasure at a random spot on a grid. Each coordinate is a random `int` from -5 to 5 inclusive. The player stands at $(1, 2)$. The game reports two distances: the number of **blocks** to walk along the grid lines (horizontal distance plus vertical distance), and the straight-line distance "as the crow flies." Here are the lines that do the work:

```
1 int treasureX = (int) (Math.random() * 11) - 5;
2 int treasureY = (int) (Math.random() * 11) - 5;
3 int playerX = 1;
4 int playerY = 2;
5 int blocks = Math.abs(treasureX - playerX) + Math.abs(treasureY - playerY);
6 double distance = Math.sqrt(Math.pow(treasureX - playerX, 2) + Math.pow(treasureY - playerY, 2));
```

Choosing the range (1.11.7). From -5 to 5 inclusive there are $5 - (-5) + 1 = 11$ whole numbers, so the multiplier is 11. Truncating gives 0 through 10, and sliding by -5 gives -5 through 5. Each coordinate makes its own call, so the two coordinates are independent random values (1.11.5).

Tracing with stand-ins. We cannot predict the real values, so suppose the two calls returned 0.85 and 0.3:

```

1 double rx = 0.85; // stand-in for the first Math.random() call
2 double ry = 0.3;  // stand-in for the second Math.random() call
3 int treasureX = (int) (rx * 11) - 5;
4 int treasureY = (int) (ry * 11) - 5;
5 int playerX = 1;
6 int playerY = 2;
7 int blocks = Math.abs(treasureX - playerX) + Math.abs(treasureY - playerY);
8 double distance = Math.sqrt(Math.pow(treasureX - playerX, 2) + Math.pow(treasureY -
  playerY, 2));
9 System.out.println(treasureX);
10 System.out.println(treasureY);
11 System.out.println(blocks);
12 System.out.println(distance);

```

```

4
-2
7
5.0

```

Step	Expression	Value
1	(int) (0.85 * 11) - 5	(int) 9.35 = 9, minus 5 = 4
2	(int) (0.3 * 11) - 5	(int) 3.3 = 3, minus 5 = -2
3	treasureX - playerX, treasureY - playerY	4 - 1 = 3, -2 - 2 = -4
4	Math.abs(3) + Math.abs(-4)	3 + 4 = 7 (an int)
5	Math.pow(3, 2) + Math.pow(-4, 2)	9.0 + 16.0 = 25.0
6	Math.sqrt(25.0)	5.0 (a double)

Why each method (1.11.2 to 1.11.4). The vertical difference is -4 , but nobody walks "negative 4 blocks," so `Math.abs` turns each difference into a distance. `blocks` stays an `int` because both `abs` calls receive `int` arguments. The straight-line distance squares each difference with `Math.pow`, which also makes the sign disappear (-4 squared is 16), adds them, and takes one square root of the whole sum, never one per piece. Because `pow` and `sqrt` return `double` values, `distance` must be a `double`.

Checking the extremes. With stand-in 0.0 for both calls, the treasure is at $(-5, -5)$, the low corner. With values just under 1.0 the coordinates are 5, the high corner. Every whole-number point on the grid between them is possible.

Common mistakes

Every mistake in this chapter falls into one of two groups: the compiler stops you, or the program runs and quietly gives a wrong result. Use this table to go from what you see to its cause.

What you see	Likely cause	Fix
cannot find symbol, symbol: method sqrt(double)	Called <code>sqrt(...)</code> without <code>Math</code> .	Write <code>Math.sqrt(...)</code>

cannot find symbol, symbol: variable math	Wrote math in lowercase	Capital M: Math
possible lossy conversion from double to int	Stored a pow, sqrt, or double- version abs result in an int	Use a double variable, or cast on purpose
method random in class Math cannot be applied to given types	Passed an argument to Math.random()	Call Math.random() with no arguments, then do arithmetic
Compiles, but the variable still has its old value	Called a Math method without storing the result	<code>x = Math.abs(x);</code>
Compiles, but a random "roll" is always the same	Cast before multiplying: <code>(int) Math.random() * n</code>	<code>(int) (Math.random() * n)</code>
Compiles, but the high end of an int range never appears	Multiplied by high - low instead of high - low + 1	Count the values, including both ends
Compiles, but a square-root result is too big	Took sqrt of each piece instead of the whole sum	One <code>Math.sqrt(...)</code> around the whole sum

Vocabulary

Term	Definition	Example
<code>java.lang</code>	The package whose classes, including Math and System, are available to every program by default	No import line for Math
available by default	Usable without an import line	<code>Math.sqrt(9)</code> works in any program
Math class	A class in <code>java.lang</code> that contains only class methods for calculations	<code>Math.abs</code> , <code>Math.pow</code> , <code>Math.sqrt</code> , <code>Math.random</code>
absolute value	A number's distance from zero, never negative	<code>Math.abs(-8)</code> is 8
base / exponent	In a power, the number multiplied / how many times it is multiplied	<code>Math.pow(2, 10)</code> : base 2, exponent 10
square root	A number that, multiplied by itself, gives the original; <code>Math.sqrt</code> returns the nonnegative one	<code>Math.sqrt(20.25)</code> is 4.5
nonnegative	Zero or positive	0.0, 4.5
random number	A value that cannot be predicted ahead of time	The result of <code>Math.random()</code>
range	The set of values an expression can produce	0.0 up to 1.0 for <code>Math.random()</code>
endpoint	Either end of a range	60.0 and 75.0
inclusive / exclusive	An endpoint that can occur / one that cannot	<code>Math.random()</code> : 0.0 inclusive, 1.0 exclusive
stand-in	This chapter's name for a possible return value substituted for <code>Math.random()</code> to trace code	Try <code>r = 0.0</code> to find the low end

width / count	The size of a <code>double</code> range (high – low) / the number of whole numbers in an <code>int</code> range (high – low + 1)	1 to 6: count 6
----------------------	--	-----------------

Chapter summary

- `Math` is in the `java.lang` package, so it is available in every program without an import line. It contains only class methods, called as `Math.methodName(...)`.
- `Math.abs` returns a value's distance from zero. It is overloaded: an `int` argument gives an `int`, a `double` argument gives a `double`.
- `Math.pow(base, exponent)` and `Math.sqrt(x)` always return a `double`. Argument order matters for `pow`, and `sqrt` returns the nonnegative root of a nonnegative argument.
- A `Math` call is an expression: store or use its return value, and when calls are nested, Java evaluates them from the inside out. The methods never change their arguments.
- `Math.random()` returns an unpredictable `double` from 0.0 (inclusive) up to 1.0 (exclusive). Trace it with stand-in values, especially 0.0 and a value just under 1.0.
- Multiply to set the width and add to set the starting point: `Math.random() * width + low` for a `double`, and `(int) (Math.random() * count) + low` for an `int`, where `count` is `high - low + 1` when both ends are included.

Review questions

Question 14 Multiple choice

Each statement is in a program with no import lines. Which statement compiles?

- A. `double a = Math.abs(-3);`
- B. `int b = Math.sqrt(16);`
- C. `int c = math.abs(-3);`
- D. `double d = Math.random(5);`

Question 15 Tracing

What does this code print?

```
1 int side = 3;
2 double hyp = Math.sqrt(Math.pow(side, 2) + Math.pow(4, 2));
3 System.out.println(hyp);
4 System.out.println(Math.abs(side - 10));
5 System.out.println((int) Math.pow(2, 5) / 3);
6 System.out.println(Math.pow(2, 5) / 3);
```

Question 16 Short answer

A program finds the side of a square whose area equals the difference between two other areas:

```
1 double areaA = 9.0;
2 double areaB = 25.0;
3 double side = Math.sqrt(areaA - areaB);
```

Explain what is wrong with the third line, and how wrapping part of it in `Math.abs` fixes the problem. What value does `side` hold after the fix?

Question 17 Multiple choice

What values can `1 - Math.random()` produce?

- A. From 0.0 inclusive up to 1.0 exclusive
- B. Greater than 0.0, up to and including 1.0
- C. From -1.0 inclusive up to 0.0 exclusive
- D. Exactly 0.0 or exactly 1.0

Question 18 Debugging

A simulation needs a random temperature from -10.0 inclusive up to 30.0 exclusive. The programmer wrote:

```
1 double temp = Math.random() * 30 - 10;
```

What range does this line actually produce? Write the corrected line.

Question 19 Multiple choice

Which values can `(int) (Math.random() * 20) + 5` produce?

- A. Every `int` from 5 to 25, inclusive
- B. Every `int` from 5 to 24, inclusive
- C. Every `int` from 0 to 19, inclusive
- D. Every `int` from 6 to 25, inclusive

Question 20 Short answer

A game needs a random direction: either -1 or 1. The programmer uses `(int) (Math.random() * 2) * 2 - 1`. List every value this expression can produce, and explain with stand-ins why no other value is possible.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

☐ I can explain why `Math` needs no import line and why its methods are called with the class name. Reread: 1.11.1 The Math Class Is Always Available

☐ I can predict the value and the type of a `Math.abs` call. Reread: 1.11.2 Absolute Value

☐ I can evaluate `Math.pow` and explain why it returns a `double`. Reread: 1.11.3 Raising to a Power

☐ I can evaluate nested `Math` calls from the inside out. Reread: 1.11.4 Square Roots

☐ I can state which values `Math.random()` can return and which endpoint is exclusive. Reread: 1.11.5 Random Numbers

☐ I can write and check the range of a random `double` expression. Reread: 1.11.6 Random Decimal Values in a Range

☐ I can find the possible values of a random `int` expression and spot an off-by-one error. Reread: 1.11.7 Random Integers in a Range

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. Classes in the `java.lang` package, including `Math`, are available to every program by default, and every `Math` method is a class method called as `Math.methodName(...)`.

- **A** is wrong: that is true of classes such as `Scanner`, but not of classes in `java.lang`. No import is needed for `Math`.
- **C** is wrong: "available by default" means the class is visible. The class name is still required, because your code is not inside the `Math` class. Leaving it off gives `cannot find symbol`.
- **D** is wrong: `Math` contains only class methods.

Reread 1.11.1 *The Math Class Is Always Available* if you chose differently.

Question 2

Error: line 6 writes `math` with a lowercase `m`.

Corrected line:

```
1 double side = Math.sqrt(gardenArea);
```

Java is case-sensitive (Topic 1.1), so `math` and `Math` are different names. Java does not know anything called `math`, so it reports `cannot find symbol: variable math: it guessed math might be a variable, and there is none`. Nothing runs until the line is fixed; after the fix the program prints `7.0`.

Reread 1.11.1 *The Math Class Is Always Available* if you missed the error.

Question 3

```
8
13
3
4.0
```

Line	Argument evaluated	Version of abs	Returned
3	<code>low = -8</code>	<code>int</code>	8
4	<code>high - low = 5 - (-8) = 13</code>	<code>int</code>	13
5	<code>low + high = -8 + 5 = -3</code>	<code>int</code>	3
6	<code>low / 2.0 = -4.0</code>	<code>double</code>	4.0

On line 4, subtracting a negative number adds, so the argument is `13`, which is already positive. On line 6, dividing by the `double 2.0` makes the argument a `double` (Topic 1.3), so the `double` version runs and the result prints with a decimal point.

Reread 1.11.2 *Absolute Value* if your output differed.

Question 4

The compiler reports, for line 7:

```
error: incompatible types: possible lossy conversion from double to int
```

end - start mixes an `int` and a `double`, so the result is the `double` `-3.5`. A `double` argument calls the `double` version of `abs`, which returns the `double` `3.5`. Java will not store a `double` in an `int` variable on its own, because the digits after the decimal point would be lost.

Corrected line (keep the exact gap):

```
1 double gap = Math.abs(end - start);
```

```
1 double start = 5.5;
2 int end = 2;
3 double gap = Math.abs(end - start);
4 System.out.println(gap);
```

This prints `3.5`. Writing `int gap = (int) Math.abs(end - start);` would also compile, but the cast truncates the gap to `3`, which loses information the problem cares about.

Reread *1.11.2 Absolute Value* if you missed why the result is a `double`.

Question 5

B. `Math.pow(3, 2)` is $3 \times 3 = 9.0$ and `Math.pow(2, 3)` is $2 \times 2 \times 2 = 8.0$. Adding two `double` values gives the `double` `17.0`.

```
1 System.out.println(Math.pow(3, 2) + Math.pow(2, 3));
```

- **A** is wrong: the value is right but the type is not. `pow` returns a `double`, so the sum prints with `.0`.
- **C** is wrong: it treats `pow(3, 2)` as 3×2 and `pow(2, 3)` as 2×3 (multiplying base and exponent instead of repeated multiplication).
- **D** is wrong: it uses 9 twice, as if the order of the arguments did not matter. `pow(2, 3)` is 8, not 9.

Reread *1.11.3 Raising to a Power* if you chose differently.

Question 6

Model answer: The header is `static double pow(double base, double exponent)`. The return type is `double`, so `pow` always returns a `double`, even when the answer is a whole number. The `int` arguments 5 and 2 are widened to `5.0` and `2.0`, the method returns `25.0`, and Java prints a `double` with a decimal point.

A complete answer must:

- point to the `double` return type in the header, and
- say that the type of the result does not depend on whether the answer happens to be whole.

Reread *1.11.3 Raising to a Power* if your explanation differed.

Question 7

```
4.0
4.0
7.0
```

Line 1, inside out: `Math.pow(5, 2)` is 25.0; `Math.pow(3, 2)` is 9.0; 25.0 - 9.0 is 16.0; `Math.sqrt(16.0)` is 4.0.

Line 3: 9 - 25 is -16. Passing -16 straight to `sqrt` would break its precondition, so the code first takes `Math.abs(-16)`, which is the `int` 16. That is widened to 16.0, and `sqrt` returns 4.0.

Line 5: `Math.sqrt(9)` is 3.0 and `Math.sqrt(16)` is 4.0; their sum is 7.0. Notice this is not `Math.sqrt(25)`, which would be 5.0.

Reread 1.11.4 *Square Roots* if your output differed.

Question 8

B. `sqrt` returns the nonnegative square root, and its return type is `double`, so the result is 6.0.

- **A** is wrong: the value is right but the type is not. The `int` argument is widened, and `sqrt` returns a `double`.
- **C** is wrong: -6.0 squared is also 36, but `sqrt` always returns the nonnegative root.
- **D** is wrong: a method returns exactly one value.

Reread 1.11.4 *Square Roots* if you chose differently.

Question 9

B. The range of `Math.random()` is from 0.0 inclusive to 1.0 exclusive, so 0.0 is a possible result.

- **A** is wrong: the 1.0 endpoint is exclusive. Results get as close to 1.0 as you like but never reach it.
- **C** is wrong: no result is below 0.0.
- **D** is wrong: `random` returns a `double`, never the `int` 1, and its results are always less than 1.

Reread 1.11.5 *Random Numbers* if you chose differently.

Question 10

B. Multiplying by 4 stretches the range to 0.0 up to 4.0; adding 2 slides it to 2.0 up to 6.0. Stand-in 0.0 gives 2.0, so 2.0 is inclusive. `Math.random()` never returns 1.0, so 6.0 is never reached: exclusive.

- **A** is wrong: it forgets the + 2, which slides the range.
- **C** is wrong: the high endpoint stays exclusive because `Math.random()` never returns 1.0.
- **D** is wrong: it treats 4 as the high end, but 4 is the width. The high end is 2 + 4 = 6.

Reread 1.11.6 *Random Decimal Values in a Range* if you chose differently.

Question 11

B. The width is $100 - 50 = 50$ and the low end is 50, so the expression is `Math.random() * 50 + 50`. Stand-in 0.0 gives 50.0; values just under 1.0 give just under 100.0.

- **A** is wrong: it multiplies by the high end instead of the width, giving 50.0 up to 150.0.
- **C** is wrong: precedence makes it `Math.random() + 2500`, a range from 2500.0 up to 2501.0.
- **D** is wrong: it stretches correctly but never slides, giving 0.0 up to 50.0.

Reread 1.11.6 *Random Decimal Values in a Range* if you chose differently.

Question 12

C. The count of whole numbers from 5 to 12 is $12 - 5 + 1 = 8$. Truncating `Math.random() * 8` gives 0 through 7, and adding 5 gives 5 through 12.

- **A** is wrong: it multiplies by the high end instead of the count, giving 5 through 16.

- **B** is wrong: it forgets the `+ 1` in the count, giving 5 through 11, so 12 can never occur.
- **D** is wrong: the cast applies to `Math.random()` alone, which always truncates to 0, so the result is always 5.

Reread *1.11.7 Random Integers in a Range* if you chose differently.

Question 13

The count is correct (50 tickets), but nothing slides the range. Truncating `Math.random() * 50` gives 0 through 49. So ticket 0, which does not exist, can win, and ticket 50 never can.

Corrected line:

```
1 int winner = (int) (Math.random() * 50) + 1;
```

Check with stand-ins: `0.0` gives `0 + 1 = 1`, and `0.999` gives `(int) 49.95 = 49`, plus `1 = 50`. The code compiled and ran before the fix, which is why only checking the endpoints reveals the error.

Reread *1.11.7 Random Integers in a Range* if you missed the error.

Question 14

A. No import is needed, because `Math` is in `java.lang`. `Math.abs(-3)` calls the `int` version and returns the `int` 3. Storing an `int` in a `double` variable is allowed, because Java widens it automatically (Topic 1.5), so `a` holds `3.0`.

```
1 double a = Math.abs(-3);
2 System.out.println(a);
```

- **B** is wrong: `sqrt` always returns a `double`, even for 16, so storing it in an `int` gives possible lossy conversion from `double` to `int`.
- **C** is wrong: Java is case-sensitive, so `math` is an unknown name (cannot find symbol).
- **D** is wrong: `random()` takes no arguments, so the call does not match its header.

```
1 int b = Math.sqrt(16);
```

Choice B does not compile:

error: incompatible types: possible lossy conversion from `double` to `int`.

Reread *1.11.2 Absolute Value*, *1.11.4 Square Roots*, and *1.11.5 Random Numbers* if you chose differently.

Question 15

```
5.0
7
10
10.666666666666666
```

- Line 2, inside out: `Math.pow(3, 2)` is `9.0`, `Math.pow(4, 2)` is `16.0`, their sum is `25.0`, and `Math.sqrt(25.0)` is `5.0`.
- Line 4: side `- 10` is `-7`, an `int`, so the `int` version of `abs` returns `7` (no decimal point).
- Line 5: the cast applies to `Math.pow(2, 5)` only, turning `32.0` into the `int` 32. Then `32 / 3` divides two `int` values, which keeps only the whole part (Topic 1.3): `10`.
- Line 6: without the cast, `32.0 / 3` has a `double` operand, so the division keeps the fraction.

Lines 5 and 6 differ only by the cast, which changes the type and therefore the kind of division.

Reread *1.11.3 Raising to a Power* and *1.11.4 Square Roots* if your output differed.

Question 16

Model answer: `areaA - areaB` is $9.0 - 25.0 = -16.0$, so the argument passed to `Math.sqrt` is negative. That breaks `sqrt`'s precondition that its argument is not negative, and `sqrt` does not check for that, so the program does not get a useful side length. The difference between two areas is meant as a size, so the sign does not matter. Writing `Math.sqrt(Math.abs(areaA - areaB))` evaluates the inner call first, turning -16.0 into 16.0 , and then `sqrt` returns 4.0 .

```
1 double areaA = 9.0;
2 double areaB = 25.0;
3 double side = Math.sqrt(Math.abs(areaA - areaB));
4 System.out.println(side);
```

A complete answer must:

- say the argument is -16.0 , which breaks the nonnegative precondition,
- explain that `Math.abs` runs first (inside out) and makes the argument 16.0 , and
- give the result 4.0 .

Reread *1.11.4 Square Roots* and *1.11.2 Absolute Value* if your explanation differed.

Question 17

B. Try the stand-ins at the edges. If `Math.random()` returns 0.0 (which it can), the result is $1 - 0.0 = 1.0$, so 1.0 is possible: inclusive. As `Math.random()` gets closer to 1.0 , the result gets closer to 0.0 , but `Math.random()` never reaches 1.0 , so the result never reaches 0.0 : exclusive. Subtracting from 1 flips which endpoint is inclusive.

- **A** is wrong: it assumes the endpoints keep their kinds, but the subtraction reverses the range.
- **C** is wrong: that would be `Math.random() - 1`. Here the random value is subtracted from 1 , not the other way around.
- **D** is wrong: `Math.random()` returns many values between its endpoints, not just two.

Reread *1.11.5 Random Numbers* and *1.11.6 Random Decimal Values in a Range* if you chose differently.

Question 18

The line multiplies by the high end, 30 , instead of the width. It stretches to 0.0 up to 30.0 and then slides down by 10 , giving **-10.0 inclusive up to 20.0 exclusive**. Temperatures from 20.0 to 30.0 can never occur.

The width is $30 - (-10) = 40$, so:

```
1 double temp = Math.random() * 40 - 10;
```

Check with stand-ins 0.0 , 0.5 , and 0.999 in place of the call:

```
1 double r = 0.0;
2 System.out.println(r * 40 - 10);
3 r = 0.5;
4 System.out.println(r * 40 - 10);
5 r = 0.999;
6 System.out.println(r * 40 - 10);
```

The low end is -10.0 , the middle is 10.0 , and values just under 1.0 approach 30.0 without reaching it.

Reread *1.11.6 Random Decimal Values in a Range* if you missed the error.

Question 19

B. Truncating `Math.random() * 20` gives the 20 values 0 through 19. Adding 5 gives 5 through 24. Check: stand-in `0.0` gives 5, and `0.999` gives `(int) 19.98 = 19`, plus 5 = 24.

- **A** is wrong: reaching 25 would need the truncated product to be 20, which needs `Math.random()` to return 1.0. It never does.
- **C** is wrong: it forgets the `+ 5`.
- **D** is wrong: it treats the smallest truncated value as 1 instead of 0.

Reread *1.11.7 Random Integers in a Range* if you chose differently.

Question 20

Model answer: It produces only `-1` and `1`. Truncating `Math.random() * 2` gives only 0 or 1, because the product ranges from 0.0 up to 2.0 (exclusive). Then multiply by 2 and subtract 1: $0 \times 2 - 1 = -1$, and $1 \times 2 - 1 = 1$. For example, stand-in 0.3 gives `(int) 0.6 = 0`, so `-1`; stand-in 0.7 gives `(int) 1.4 = 1`, so `1`.

```
1 double r = 0.3;
2 System.out.println((int) (r * 2) * 2 - 1);
3 r = 0.7;
4 System.out.println((int) (r * 2) * 2 - 1);
```

A complete answer must:

- name the two possible truncated values, 0 and 1, and why 2 is impossible,
- show the arithmetic that turns 0 into `-1` and 1 into `1`, and
- note that the cast applies to the whole product because of the parentheses.

Reread *1.11.7 Random Integers in a Range* if your reasoning differed.