

1.7 Application Program Interface (API) and Libraries

Self-Study Guide · About 45 minutes · With answer key

By the end you will be able to

- Explain what a library is and why programmers use classes that someone else already wrote.
- Name the package a library class belongs to and explain when a program needs an import line.
- Read a simplified page of API documentation and find the method you need.
- Sort what a class offers into attributes (data stored in variables) and behaviors (actions defined by methods).
- Explain that a class name such as `String` or `Scanner` is a type, and that a class is used to create objects.

What to remember from earlier

- Every Java program lives inside a class, and the compiler reports errors before the program runs. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.5 The Compiler: Checking Before Running](#))
- Types are either primitive (`int`, `double`, `boolean`) or reference. ([Topic 1.2: Variables and Data Types, 1.2.2 Primitive Types and Reference Types](#))
- `print` and `println` display output, and a string literal is text in double quotes. ([Topic 1.3: Expressions and Output](#))
- To read input, import `java.util.Scanner`; and create a `Scanner` with `new`. ([Topic 1.4: Assignment Statements and Input, 1.4.6 Input and the Scanner Class](#))
- `Integer.MAX_VALUE` and `Integer.MIN_VALUE` hold the largest and smallest `int` values. ([Topic 1.5: Casting and Range of Variables, 1.5.5 The Range of `int` Values](#))

1.7.1 Libraries Are Collections of Classes

Suppose a program needs the length of one side of a square garden whose area is 144 square feet. You need a square root. With only `+`, `-`, `*`, `/`, and `%`, there is no direct way to get one; you would have to invent an algorithm that guesses and improves its guess over and over. That is a lot of work for one small calculation, and if you made a mistake, every program that used your code would inherit it.

Programmers almost never start from nothing. Someone has already written, tested, and fixed code for the jobs that nearly every program needs: printing, reading input, working with text, doing math. You reuse their work instead of rewriting it.

To see how that work is packaged, recall something from Topic 1.1. Every program you have written starts with a line such as `public class Greeting`, and all of its code sits inside the braces that follow. That wrapper is a **class**: a named unit of Java code that groups related data together

with the actions that work on that data. (Read past the word `public` for now; a later topic explains it.) Your programs have each been one small class. The ready-made code that comes with Java is organized the same way, into classes, but there are thousands of them.

A **library** is a collection of classes that were written to be used by other programs. Java comes with a large standard library, and you have been using it all along:

Name you have typed	What it is	Where you used it
System	A library class that gives a program access to the screen	Topic 1.3 output
String	A library class for text	Topic 1.3
Scanner	A library class that reads input	Topic 1.4
Integer	A library class with facts about <code>int</code> values	Topic 1.5
Math	A library class for calculations, such as square roots	This topic

None of these words is part of the Java language itself the way `int` and `double` are. They are names of classes that other programmers wrote and that come with every Java installation.

KEY IDEA

A library is a collection of classes someone else already wrote and tested. Your program uses them by name.

WORKED EXAMPLE Spotting the library classes in a program

Read this program and look for capitalized names that you did not write yourself.

GardenPlan.java

```

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

Go through it one name at a time:

Line	Class name	Who wrote it?	What the program uses it for
1	GardenPlan	You	Holds this program's own code
5	String	Library	The type of <code>plant</code> , which holds text
6	Math	Library	Its square-root code, <code>sqrt</code>
7–10	System	Library	<code>System.out.print</code> and <code>println</code> display results

Line 6 is new. `Math.sqrt(144.0)` hands the value `144.0` to the square-root code in the `Math` class, and that code gives back `12.0`, which is stored in `side`. You will learn the full rules for

calls like this in Topic 1.10; for now, read it as "use the `Math` class's square-root code on 144.0." The program displays:

```
Plant: tomatoes
Side length in feet: 12.0
```

(`String` also appears on line 3, inside the `main` line that every program has.) The program has only six statements, yet it relies on three classes it never defined. All the work of finding a square root, storing text, and talking to the screen lives in the library.

Now vary it. Here is a different program with the same shape:

`ScoreBoard.java`

```
1 public class ScoreBoard
2 {
3     public static void main(String[] args)
4     {
5         String team = "Hawks";
6         int lead = Math.abs(-12);
7         System.out.println(team);
8         System.out.println(lead);
9     }
10 }
```

```
Hawks
12
```

This time the program's own class is `ScoreBoard`, and the `Math` calculation it uses is `abs`, which gives the distance of a number from zero, so `Math.abs(-12)` is 12. The library classes are still `String`, `Math`, and `System`. The lesson does not depend on which calculation you do: one class is yours, and every other capitalized class name came from the library.

WATCH OUT

Changing the spelling of a library class. A library class has exactly one name, and Java is case-sensitive. Students who type quickly often write `math` instead of `Math`:

`LowercaseClass.java`

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

The compiler stops at line 5:

```
LowercaseClass.java:5: error: cannot find symbol
    double side = math.sqrt(144.0);
                   ^
    symbol:   variable math
```

There is no class named `math` in the library, so Java guesses you meant a variable called `math`, and no such variable exists either. The exact wording can vary between Java versions, but `cannot find symbol` is the key phrase. The fix is to spell the class name exactly as the library does: `Math.sqrt(144.0)`.

PAUSE AND TRY

Question 1 Multiple choice

In the program `ScoreBoard.java` above, which name is **not** the name of a library class?

- A. `String`
- B. `Math`
- C. `System`
- D. `ScoreBoard`

Question 2 Short answer

Jordan says, "I'd rather write my own square-root code than use `Math.sqrt`, so I know exactly what it does." Give two reasons a programmer would usually use the library class instead.

1.7.2 Packages and import

Java's standard library has thousands of classes. If they were all in one giant pile, two problems would appear. Finding the class you need would be hard, and two programmers could not give different classes the same short name without a clash. Java solves both problems the way a school organizes its library: books are grouped on labeled shelves.

A **package** is a named group of related classes. The package name is written in lowercase words joined by dots. Two packages matter in this course:

Package	Some classes in it	Needs an import line?
<code>java.lang</code>	<code>String</code> , <code>Math</code> , <code>System</code> , <code>Integer</code>	No
<code>java.util</code>	<code>Scanner</code>	Yes

A class's **full name** is its package name, a dot, and its class name. `Scanner`'s full name is `java.util.Scanner`: the class `Scanner` in the package `java.util`. `String`'s full name is `java.lang.String`.

Typing full names everywhere would be tiring, so Java lets you name a class once at the top of the file and then use its short name. That is what an **import statement** does. In Topic 1.4 you wrote

```
1 import java.util.Scanner;
```

at the top of a program before using `Scanner`. Now you can read that line: "In this file, when I write `Scanner`, I mean the class `Scanner` from the package `java.util`." An import line goes above the class, ends with a semicolon, and does not run anything; it only tells the compiler where to find a name.

Why do `String` and `Math` never need an import line? The `java.lang` package holds the classes nearly every program needs, so Java makes all of its classes **available by default**: every program can use them by their short names without importing them. Every other package, including `java.util`, must be imported.

KEY IDEA

Library classes are grouped into packages. Classes in `java.lang` are available by default; a class from any other package, such as `java.util.Scanner`, needs an import line before you can use its short name.

WORKED EXAMPLE Importing Scanner

This program reads two numbers and adds them. Look at line 1, and notice which class names on later lines it makes available.

TicketCount.java

```
1 import java.util.Scanner;
2
3 public class TicketCount
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner("12 7");
8         int adults = reader.nextInt();
9         int children = reader.nextInt();
10        int total = adults + children;
11        System.out.print("Tickets: ");
12        System.out.println(total);
13    }
14 }
```

Step by step:

1. **Line 1** tells the compiler that `Scanner` means `java.util.Scanner`. Nothing happens yet when the program runs; this is information for the compiler.
2. **Line 7** creates a `Scanner`. This one reads from the text "12 7" instead of the keyboard, so the output is the same every time; reading from the keyboard works the same way. Because of line 1, the short name `Scanner` is allowed here.
3. **Lines 8–9** read the two whole numbers, so `adults` is 12 and `children` is 7.
4. **Line 10** stores `12 + 7`, which is 19, in `total`.
5. **Lines 11–12** use `System`, which is in `java.lang`, with no import needed.

Tickets: 19

To see that the import line is only a shortcut, here is the same program with no import line at all. It uses `Scanner`'s full name instead:

FullName.java

```
1 public class FullName
2 {
3     public static void main(String[] args)
4     {
5         java.util.Scanner reader = new java.util.Scanner("12 7");
6         int adults = reader.nextInt();
7         int children = reader.nextInt();
8         int total = adults + children;
9         System.out.print("Tickets: ");
10        System.out.println(total);
11    }
12 }
```

Tickets: 19

The output is identical. Writing `java.util.Scanner` every time tells the compiler exactly where the class lives, so no import is needed. Programmers rarely write code this way, because it is long and hard to read, but it shows what the import line really does: it lets you shorten the full name to `Scanner`.

WATCH OUT

Forgetting the import line. Delete the import line from `TicketCount.java` and the program no longer compiles:

MissingImport.java

```
1 public class MissingImport
2 {
3     public static void main(String[] args)
4     {
5         Scanner reader = new Scanner("12 7");
6         int adults = reader.nextInt();
7         int children = reader.nextInt();
8         int total = adults + children;
9         System.out.print("Tickets: ");
10        System.out.println(total);
11    }
12 }
```

```
MissingImport.java:5: error: cannot find symbol
    Scanner reader = new Scanner("12 7");
    ^
symbol:   class Scanner
```

Students often forget because `String`, `Math`, and `System` never need an import, so it feels as though library classes are always available. Only `java.lang` classes are. The `symbol: class Scanner` line is the clue: Java does not know any class by that short name. (The compiler reports the same problem a second time for the second `Scanner` on that line; wording can vary between Java versions.) The fix is to put `import java.util.Scanner;` back at the top of the file.

PAUSE AND TRY**Question 3** Multiple choice

In the full name `java.util.Scanner`, which part names the package?

- A. `java`
- B. `java.util`
- C. `util.Scanner`
- D. `Scanner`

Question 4 Short answer

A program uses `String`, `Math`, and `Scanner`. How many import lines does it need, and why?

1.7.3 Reading API Documentation

You now know that `Math` holds square-root code. But how would you know that in the first place? And once you know, how would you find out its exact name, what to give it, and what it gives back? You cannot open the library's code and read thousands of lines; even if you could, that would defeat the point of reusing it.

Think of a microwave. You do not need to understand its wiring to use it. You need to know what the buttons are and what each one does. The buttons are the microwave's interface: the part meant for users.

A library class works the same way. An **application programming interface (API)** is the set of classes, and the parts of those classes, that a library offers for other programs to use. (The College Board calls this topic "Application Program Interface"; both names mean the same thing.) The **API specification**, usually called the **documentation**, is the written description of the API: which classes exist, which package each one is in, what each one is for, and how to use each part. Java's documentation is free online, with one page per class. Reading it is how programmers learn what a class can do without reading its code.

Here is a simplified version of the `Math` documentation page, written in our own words. The real page has more lines and many more methods, arranged in tables.

```
Package java.lang
Class Math
```

```
java.lang.Object
    java.lang.Math
```

`Math` holds methods for common number work, such as absolute values, powers, and square roots.

Method Summary

<code>static int</code>	<code>abs(int a)</code>	the absolute value of <code>a</code>
<code>static double</code>	<code>pow(double a, double b)</code>	<code>a</code> raised to the power <code>b</code>
<code>static double</code>	<code>sqrt(double a)</code>	the nonnegative square root of <code>a</code>

Take it apart from the top:

- **Package** says where the class lives. `java.lang` means no import is needed (1.7.2).
- **The two indented lines** show where `Math` sits in a family tree of classes. Read past them for now; Topic 1.12 explains them.
- **The description** says what the class is for, in plain words.
- **Method Summary** lists the class's **methods**: named pieces of code that each do one job. Each row gives the method's name, what it needs in the parentheses, and what kind of value it gives back, shown just before the name. `sqrt` needs one `double` and gives back a `double`. (Read past the word `static` for now; Topic 1.10 explains it. Topic 1.9 teaches these method headers in full.)

KEY IDEA

The API documentation tells you what a library class is for and how to use it, so you can use the class without seeing its code.

WORKED EXAMPLE Finding the method you need

A weather program needs the size of the change between a morning temperature of 58 and an afternoon temperature of 71, as a positive number. Subtracting gives `58 - 71`, which is `-13`, but a "size of change" should not be negative.

Scan the Method Summary. `abs` returns "the absolute value of a": a number's distance from zero, which is never negative. That is the job. Its row says it takes one `int`, and both temperatures are `ints`.

```
1 int morning = 58;
2 int afternoon = 71;
3 int swing = Math.abs(morning - afternoon);
4 System.out.println(swing);
```

Step	Value
<code>morning - afternoon</code>	<code>-13</code>
<code>Math.abs(-13)</code> gives back	<code>13</code>
<code>swing</code> holds	<code>13</code>

13

You found and used `abs` from three facts in the documentation: its name, what it does, and what it needs. You never saw the code inside it.

Now vary it. The documentation matters most when a method needs **more than one** value, because the order of the values is part of what the documentation tells you. The `pow` row says the result is "a raised to the power b", so the first value is the base and the second is the exponent.

```
1 double first = Math.pow(2.0, 10.0);
2 double second = Math.pow(10.0, 2.0);
3 System.out.println(first);
4 System.out.println(second);
```

```
1024.0
100.0
```

`Math.pow(2.0, 10.0)` is 2 multiplied by itself 10 times, 1024.0. Swapping the values gives 10 squared, 100.0. Same method, same two numbers, different answers; only the documentation tells you which order means what.

WATCH OUT

Guessing a method's name instead of reading the documentation. A method called `squareRoot` sounds reasonable, so a student writes:

`GuessedName.java`

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

```
GuessedName.java:5: error: cannot find symbol
    double side = Math.squareRoot(144.0);
                        ^
symbol:   method squareRoot(double)
```

The `Math` class has no method with that name, so the compiler cannot find it. The `symbol:` line names the method it looked for. (Wording can vary between Java versions.) The fix is to check the documentation and use the real name: `Math.sqrt(144.0)`.

PAUSE AND TRY

Question 5 Multiple choice

Using only the simplified `Math` documentation above, which call gives back the distance of -40 from zero?

- A. `Math.sqrt(-40)`
- B. `Math.abs(-40)`
- C. `Math.pow(-40, 2)`
- D. `Math.distance(-40)`

Question 6 Tracing

Use the `pow` row of the documentation to predict exactly what this code displays.

```
1 double area = Math.pow(3.0, 2.0);
2 double volume = Math.pow(2.0, 3.0);
3 System.out.println(area);
4 System.out.println(volume);
```

1.7.4 Attributes and Behaviors

A documentation page can list dozens of entries. To make sense of them, it helps to know that everything a class offers falls into two kinds. Think of describing a phone. Some things are facts about it: its battery level, its phone number. Other things are actions: it can make a call, take a photo. Facts and actions are different kinds of information, and classes are described the same way.

- An **attribute** is a piece of data that belongs with a class. Attributes are stored in variables.
- A **behavior** is something the class's objects can do, or something that can be done with them. Behaviors are defined by methods.

(An **object** is one particular thing made from a class, such as one particular piece of text; 1.7.5 explains this.) A handy test is to ask whether the entry is a fact or an action: a noun such as "the largest value" is an attribute, and a verb phrase such as "give back the square root" is a behavior.

You have already used an attribute. In Topic 1.5, the constant `Integer.MAX_VALUE` held the largest possible `int`; a constant is an attribute whose value never changes. The `Integer` class's documentation lists it in a section called **Field Summary**, which is where a documentation page lists the attributes that programs are allowed to use. (A **field** is Java's name for a variable that belongs to a class.) The behaviors, as you saw in 1.7.3, are in the **Method Summary**. Here is a simplified excerpt; the real page lists more entries.

```
Package java.lang
Class Integer

Field Summary
  static int  MAX_VALUE      the largest value an int can hold
  static int  MIN_VALUE      the smallest value an int can hold

Method Summary
  static int  parseInt(String s)  the int written in the text s
```

Not every attribute is listed. A `String` object must store its characters somewhere, but that variable is kept hidden inside the class, so the documentation describes it in words ("a `String` holds a sequence of characters") rather than listing it. You use the attribute through the class's behaviors instead.

KEY IDEA

Attributes are a class's data, stored in variables. Behaviors are what the class's objects can do, or what can be done with them, and are defined by methods.

WORKED EXAMPLE Using an attribute

An attribute is a stored value, so you read it by name with no parentheses: the class name, a dot, the attribute name.

```

1 int biggest = Integer.MAX_VALUE;
2 int smallest = Integer.MIN_VALUE;
3 System.out.println(biggest);
4 System.out.println(smallest);

```

Line	What happens	Value stored
1	Read the MAX_VALUE attribute of Integer	2147483647
2	Read the MIN_VALUE attribute of Integer	-2147483648

```

2147483647
-2147483648

```

Nothing is calculated here. The values were stored in the `Integer` class's variables, and the program reads them.

Now vary it: use a behavior instead of an attribute. Here is a simplified excerpt of the `String` documentation page. It has the same layout as the `Math` page, including the two indented lines near the top, which Topic 1.12 explains.

```

Package java.lang
Class String

```

```

java.lang.Object
    java.lang.String

```

A `String` object holds a sequence of characters (text).
Every string literal, such as `"abc"`, is a `String` object.

Method Summary

```
int length()    the number of characters in this string
```

The description mentions the attribute every string has, its characters, without listing a variable for it. The Method Summary lists the behavior `length()`. Every `String` object has this behavior, and each one answers with its own count.

```

1 String home = "Denver";
2 String trip = "Kansas City";
3 System.out.println(home.length());
4 System.out.println(trip.length());

```

```

6
11

```

`home.length()` asks the text `"Denver"` for its length, which is 6. `trip.length()` asks `"Kansas City"`, which has 11 characters because the space counts. (The rules for calls like `home.length()` are taught fully in Topic 1.14.) Two differences from the attribute example are worth noticing. A behavior is an action, so it is written with parentheses, even when nothing goes inside them. And the result depends on which object performs it, because each string holds different characters.

WATCH OUT

Treating an attribute like a behavior. `MAX_VALUE` is data, not an action, but a student who has just used `length()` adds parentheses:

AttributeParens.java

```
1 public class AttributeParens
2 {
3     public static void main(String[] args)
4     {
5         int biggest = Integer.MAX_VALUE();
6         System.out.println(biggest);
7     }
8 }
```

```
AttributeParens.java:5: error: cannot find symbol
    int biggest = Integer.MAX_VALUE();
                           ^
symbol:   method MAX_VALUE()
```

The parentheses tell Java to look for a **method** named `MAX_VALUE`, and `Integer` has none; the `symbol: method` line shows that. (Wording can vary between Java versions.) The fix is `Integer.MAX_VALUE` with no parentheses. Check which section of the documentation an entry is in: Field Summary entries are attributes, written without parentheses; Method Summary entries are behaviors, written with them.

PAUSE AND TRY**Question 7** Short answer

The `Scanner` documentation describes these, in simplified form. Sort each into **attribute** or **behavior**, and give a reason for one of them.

- the text the scanner is reading from
- read the next whole number
- how far through the text the scanner has read
- read the next word

Question 8 Multiple choice

Which line uses an **attribute** of a class rather than a behavior?

- A. `int smallest = Integer.MIN_VALUE;`
- B. `int count = name.length();`
- C. `double root = Math.sqrt(49.0);`
- D. `int size = Math.abs(-7);`

1.7.5 Classes Are Types That Make Objects

In Topic 1.2 you learned that data types come in two groups: primitive types such as `int`, `double`, and `boolean`, and reference types. Since then you have written `String` and `Scanner` as types of

variables. You may have wondered why `String` is capitalized while `int` is not. Now there is an answer: `String` is the name of a class, and every class is a type.

A class does two jobs for your program.

A class defines a type. When you write a class name in a declaration, such as `String city` or `Scanner reader`, you are saying what kind of value the variable can hold. Each class defines its own **reference type**: `String` variables hold text objects, `Scanner` variables hold scanner objects, and a variable of one type cannot hold the other. (What a reference variable actually stores is explained in Topic 1.12.)

A class is used to create objects. Think of a class as a **blueprint**. A blueprint for a house says how every house built from it will be laid out, but you cannot live in a blueprint; you live in a house built from it. In the same way, the `String` class describes what every piece of text has and can do, while `"Denver"` is one actual piece of text. One particular thing made from a class is an **object**, also called an **instance** of the class. Each object has its own values for the class's attributes: `"Denver"` and `"Kansas City"` are both `String` objects, each holding its own characters. The analogy has a limit: a house can be repainted or have rooms added beyond the plan, but an object can only have the attributes and behaviors its class defines.

So the library gives you more than ready-made calculations. It gives you ready-made blueprints, and your program can create objects from them.

KEY IDEA

Each class defines a reference type. A class is a blueprint, and an object (an instance) is one particular thing created from it.

WORKED EXAMPLE Library classes as types and as blueprints

TripInfo.java

```
1  import java.util.Scanner;
2
3  public class TripInfo
4  {
5      public static void main(String[] args)
6      {
7          int miles = 412;
8          String city = "Denver";
9          Scanner reader = new Scanner("3");
10         int days = reader.nextInt();
11         System.out.print("Driving to ");
12         System.out.println(city);
13         System.out.print("Miles: ");
14         System.out.println(miles);
15         System.out.print("Days: ");
16         System.out.println(days);
17     }
18 }
```

Look at each declaration and ask two questions: what type is the variable, and is an object being created from a library class?

Line	Variable	Type	Object created?
7	miles	int (primitive)	No; 412 is a plain number
8	city	String (a class)	Yes; the literal "Denver" is a String object
9	reader	Scanner (a class)	Yes; new Scanner("3") creates a Scanner object
10	days	int (primitive)	No; nextInt() gives back the number 3

Line 9 uses the `new` keyword, which you first saw with `Scanner` in Topic 1.4; it tells Java to create a new object from the class named after it. (Topic 1.13 explains `new` in detail.) Line 8 needs no `new`, because Java creates a `String` object automatically for every string literal. Both are objects made from library blueprints that your program never had to write.

```
Driving to Denver
Miles: 412
Days: 3
```

Compare this with `Math`. You have used `Math` in every subsection, yet the program above never declares a `Math` variable or creates a `Math` object. Its behaviors, such as `sqrt`, do not belong to any particular object, so you use them through the class name itself. Topic 1.10 explains this kind of method. A class can be a blueprint for objects, a holder of behaviors you use directly, or both.

WATCH OUT

Using the wrong type for an object. Because the variable will hold a city's name, a student declares it as an `int`:

TextInInt.java

```
1 public class TextInInt
2 {
3     public static void main(String[] args)
4     {
5         int city = "Denver";
6         System.out.println(city);
7     }
8 }
```

```
TextInInt.java:5: error: incompatible types: String cannot be converted to int
    int city = "Denver";
              ^
```

"Denver" is a `String` object, and an `int` variable can only hold a whole number. The message names both types: the value's type (`String`) and the variable's type (`int`). The fix is to declare the variable with the class that matches the object: `String city = "Denver";`.

PAUSE AND TRY

Question 9 Multiple choice

Which declaration has a **reference type** made by a library class?

- A. `int total = 19;`
- B. `double side = 12.0;`
- C. `boolean done = false;`
- D. `Scanner reader = new Scanner("5");`

Question 10 Debugging

This program does not compile. Find the error and correct the line.

LowercaseString.java

```

1 public class LowercaseString
2 {
3     public static void main(String[] args)
4     {
5         string pet = "Biscuit";
6         System.out.println(pet);
7     }
8 }
```

Putting it together

This program decides how far a word's length is from a goal length. Before reading the trace, find every library class and decide what the program uses each one for.

WordGoal.java

```

1 import java.util.Scanner;
2
3 public class WordGoal
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner("9");
8         int goal = reader.nextInt();
9         String word = "planet";
10        int letters = word.length();
11        int off = Math.abs(goal - letters);
12        System.out.print("Letters: ");
13        System.out.println(letters);
14        System.out.print("Letters away from goal: ");
15        System.out.println(off);
16    }
17 }
```

Line by line:

Line	What happens	Value
1	Makes the short name <code>Scanner</code> mean <code>java.util.Scanner</code>	
7	Creates a <code>Scanner</code> object that reads from the text <code>"9"</code>	
8	The <code>nextInt</code> behavior reads the number	<code>goal = 9</code>

9	The literal creates a <code>String</code> object	<code>word = "planet"</code>
10	The <code>length</code> behavior of that object counts its characters	<code>letters = 6</code>
11	<code>Math.abs(9 - 6)</code> , which is <code>Math.abs(3)</code>	<code>off = 3</code>
12–15	System displays the results	

Letters: 6
Letters away from goal: 3

The library classes, through this chapter's ideas:

Class	Package	Import?	Used as a type / object, or only through the class?
Scanner	java.util	Yes	Type of reader; a <code>Scanner</code> object is created
String	java.lang	No	Type of word; a <code>String</code> object is created
Math	java.lang	No	Only its <code>abs</code> behavior, through the class name
System	java.lang	No	Only for output

Every behavior the program uses, `nextInt`, `length`, `abs`, `print`, and `println`, is listed in its class's documentation. The program's author wrote nine lines of instructions and borrowed the hard parts from four library classes.

Common mistakes

Four of this chapter's mistakes produce the same message, `cannot find symbol`. The `symbol:` line under it tells you which mistake you made.

What the compiler says	Likely cause	Fix
cannot find symbol, symbol: class Scanner	Missing import line, or the wrong package imported	<code>import java.util.Scanner;</code>
cannot find symbol, symbol: variable math	Class name typed in lowercase	Spell it <code>Math</code>
cannot find symbol, symbol: method squareRoot(double)	Guessed a method name	Look up the real name: <code>sqrt</code>
cannot find symbol, symbol: method MAX_VALUE()	Parentheses after an attribute	<code>Integer.MAX_VALUE</code>
incompatible types: String cannot be converted to int	Wrong type for the object	Declare it as <code>String</code>

Vocabulary

Term	Definition	Example
------	------------	---------

class	A named unit of Java code that groups related data with the actions that work on it	<code>String</code> , <code>Math</code> , <code>GardenPlan</code>
library	A collection of classes written to be used by other programs	Java's standard library
package	A named group of related classes	<code>java.util</code>
full name	A class's package name, a dot, and its class name	<code>java.util.Scanner</code>
import statement	A line at the top of a file that lets the file use a class's short name	<code>import java.util.Scanner;</code>
available by default	Usable without an import line; true of every class in <code>java.lang</code>	<code>String</code> , <code>Math</code> , <code>System</code>
application programming interface (API)	The classes, and parts of classes, that a library offers for other programs to use	The <code>Math</code> class and its methods
API specification / documentation	The written description of an API: its classes, packages, and how to use each part	The <code>Math</code> documentation page
method	A named piece of code that does one job; behaviors are defined by methods	<code>sqrt</code> , <code>length</code>
attribute	Data that belongs with a class, stored in variables	<code>Integer.MAX_VALUE</code>
field	Java's name for a variable that belongs to a class; documentation lists them in the Field Summary	<code>MAX_VALUE</code>
behavior	What a class's objects can do, or what can be done with them, defined by methods	<code>length()</code> , <code>Math.abs</code>
reference type	A type defined by a class	<code>String</code> , <code>Scanner</code>
blueprint	A plan for making things; a class is the blueprint for its objects	The <code>String</code> class
object / instance	One particular thing created from a class	"Denver", a <code>Scanner</code> made with <code>new</code>

Chapter summary

- A library is a collection of classes that someone else wrote and tested. `String`, `Math`, `Scanner`, `Integer`, and `System` all come from Java's standard library.
- Library classes are grouped into packages. Classes in `java.lang` are available by default; classes from other packages, such as `java.util.Scanner`, need an import line to be used by their short names.

- An API's documentation describes each class: its package, its purpose, and how to use each part. Reading it, rather than guessing, is how you find a method's exact name, what it needs, and what it gives back.
- A class offers attributes, which are data stored in variables, and behaviors, which are actions defined by methods. Documentation lists usable attributes under Field Summary and behaviors under Method Summary.
- Every class defines a reference type, and a class is a blueprint used to create objects, also called instances.

Review questions

Question 11 Short answer

Here is a simplified excerpt from the `Integer` documentation:

```
Package java.lang
Class Integer

Field Summary
    static int    MAX_VALUE        the largest value an int can hold
    static int    MIN_VALUE        the smallest value an int can hold

Method Summary
    static int    parseInt(String s)  the int written in the text s
```

(a) Which entries are attributes, and which are behaviors? (b) Does a program need an import line to use `Integer`? Explain using the excerpt.

Question 12 Debugging

This program is meant to display `15`, but it does not compile. Find the error and correct it.

`WrongPackage.java`

```
1  import java.lang.Scanner;
2
3  public class WrongPackage
4  {
5      public static void main(String[] args)
6      {
7          Scanner reader = new Scanner("40 25");
8          int first = reader.nextInt();
9          int second = reader.nextInt();
10         System.out.println(first - second);
11     }
12 }
```

Question 13 Tracing

The `String` documentation says `length()` gives back the number of characters in a string, and the `Math` documentation says `abs` gives back a number's distance from zero. Trace this code and give its exact output.

```
1 String word = "library";
2 int letters = word.length();
3 int gap = Math.abs(letters - 10);
4 System.out.print(letters);
5 System.out.print(" ");
6 System.out.println(gap);
```

Question 14 Multiple choice

A program needs Java's `LocalDate` class, which the documentation says is in the package `java.time`. Which line should go at the top of the file?

- A. `import java.time;`
- B. `import java.time.LocalDate;`
- C. `import LocalDate.java.time;`
- D. No line is needed, because every library class is available by default.

Question 15 Multiple choice

Which statement is true?

- A. A library is a single class that holds all of Java's methods.
- B. The documentation of a class shows the code inside each of its methods, so you can check how it works.
- C. `String` is a class, and `"Ada"` and `"Grace"` are two objects, or instances, created from it.
- D. Attributes are defined by methods, and behaviors are stored in variables.

Reflection

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

- ☐ I can explain what a library is and pick out the library classes in a program. Reread: 1.7.1 Libraries Are Collections of Classes
- ☐ I can name the package of `String`, `Math`, and `Scanner` and say which ones need an import line. Reread: 1.7.2 Packages and import
- ☐ I can use a documentation page to find a method and learn what it needs and gives back. Reread: 1.7.3 Reading API Documentation
- ☐ I can sort what a class offers into attributes and behaviors. Reread: 1.7.4 Attributes and Behaviors
- ☐ I can explain that a class is a type and a blueprint for objects. Reread: 1.7.5 Classes Are Types That Make Objects

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

D. `ScoreBoard` is the class the programmer wrote; its code is the program itself. A library class is one that someone else wrote and that the program uses by name.

- **A** is wrong: `String` comes with Java. The program uses it as the type of `team` but never defines it.
- **B** is wrong: `Math` comes with Java. The program borrows its `abs` calculation.
- **C** is wrong: `System` comes with Java. The program uses it to display output.

Reread *1.7.1 Libraries Are Collections of Classes* if you chose differently.

Question 2

A strong answer includes two of these ideas:

- The library code is already written, so using it saves time; computing a square root by hand would take a whole algorithm.
- The library code has already been tested and used in many programs, so it is less likely to contain a mistake than brand-new code.
- The programmer can focus on the problem their own program solves instead of on a calculation that every program needs.

Jordan's worry about "knowing what it does" is answered by the class's documentation, which 1.7.3 explains. Reread *1.7.1 Libraries Are Collections of Classes* if your reasons differed.

Question 3

B. A full name is the package name, a dot, and the class name. Everything before the last dot, `java.util`, is the package; the last part, `Scanner`, is the class.

- **A** is wrong: `java` is only the first word of the package name. The package is the whole dotted name `java.util`.
- **C** is wrong: it splits the name in the wrong place and includes the class name.
- **D** is wrong: `Scanner` is the class, not the package.

Reread *1.7.2 Packages and import* if you chose differently.

Question 4

One: `import java.util.Scanner;` `String` and `Math` are in the `java.lang` package, whose classes are available by default, so they need no import. `Scanner` is in `java.util`, so its short name works only after it is imported.

Reread *1.7.2 Packages and import* if your answer differed.

Question 5

B. The documentation describes `abs` as returning the absolute value, which is a number's distance from zero, so `Math.abs(-40)` gives back `40`.

- **A** is wrong: `sqrt` finds a square root, which is a different calculation from a distance to zero.

- **C** is wrong: `pow` raises `-40` to the power 2, which multiplies it by itself instead of finding its distance from zero.
- **D** is wrong: the documentation lists no method named `distance`. Guessing a name gives `cannot find symbol`.

Reread *1.7.3 Reading API Documentation* if you chose differently.

Question 6

9.0
8.0

The documentation says `pow(a, b)` is "a raised to the power b", so the first value is the base:

Call	Meaning	Result
<code>Math.pow(3.0, 2.0)</code>	3×3	9.0
<code>Math.pow(2.0, 3.0)</code>	$2 \times 2 \times 2$	8.0

`pow` gives back a `double`, so each value prints with `.0`. If you wrote `8.0` then `9.0`, you swapped the base and the exponent. Reread *1.7.3 Reading API Documentation* if your trace differed.

Question 7

- **Attributes:** the text it is reading from, and how far through that text it has read. These are facts the `Scanner` stores (in variables), not actions.
- **Behaviors:** read the next whole number (`nextInt`) and read the next word (`next`). These are actions, defined by methods.

A good reason: "read the next whole number" is a verb phrase, something the object does, so it is a behavior; "how far it has read" is a fact the object remembers, so it is an attribute. Notice that reading a number changes how far the scanner has read: behaviors can use and change attributes.

Reread *1.7.4 Attributes and Behaviors* if you sorted them differently.

Question 8

A. `MIN_VALUE` is a stored value in the `Integer` class, listed in its Field Summary, and it is used by name with no parentheses.

- **B** is wrong: `length()` is a behavior; the parentheses show a method that acts on the string.
- **C** is wrong: `sqrt` is a behavior of `Math`, a calculation done by a method.
- **D** is wrong: `abs` is also a method of `Math`, so this is a behavior.

Reread *1.7.4 Attributes and Behaviors* if you chose differently.

Question 9

D. `Scanner` is a class from the `java.util` package, and every class defines a reference type. The variable `reader` is declared with that type, and `new Scanner("5")` creates a `Scanner` object.

- **A** is wrong: `int` is a primitive type, not a class.
- **B** is wrong: `double` is a primitive type.
- **C** is wrong: `boolean` is a primitive type.

A quick clue: Java's primitive type names are lowercase, while library class names such as `String` and `Scanner` start with a capital letter. Reread *1.7.5 Classes Are Types That Make Objects* if you chose differently.

Question 10

Line 5 uses `string`, but the class is named `String` with a capital S. Java is case-sensitive, so the compiler looks for a class named `string`, finds none, and reports `cannot find symbol with symbol: class string`. Corrected:

LowercaseStringFixed.java

```
1 public class LowercaseStringFixed
2 {
3     public static void main(String[] args)
4     {
5         String pet = "Biscuit";
6         System.out.println(pet);
7     }
8 }
```

Biscuit

A type that is a class must be spelled exactly as the library spells the class, just as with `Math` in 1.7.1. Reread *1.7.5 Classes Are Types That Make Objects* if you missed the error.

Question 11

(a) `MAX_VALUE` and `MIN_VALUE` are **attributes**: they are in the Field Summary, so they are values stored in variables of the class. `parseInt` is a **behavior**: it is in the Method Summary, so it is an action defined by a method. (Its description shows the action: it gives back the number written in a piece of text.)

(b) **No**. The first line shows that `Integer` is in the package `java.lang`, and every class in `java.lang` is available by default. That is why you could use `Integer.MAX_VALUE` in Topic 1.5 with no import.

Reread *1.7.4 Attributes and Behaviors* for part (a) and *1.7.2 Packages and import* for part (b) if your answer differed.

Question 12

Line 1 imports `Scanner` from the wrong package. `Scanner` lives in `java.util`, not `java.lang`, so the compiler reports `cannot find symbol for class Scanner in package java.lang`. Because the import failed, the uses of `Scanner` on line 7 fail too. The corrected line 1 is `import java.util.Scanner;`:

WrongPackageFixed.java

```

1  import java.util.Scanner;
2
3  public class WrongPackageFixed
4  {
5      public static void main(String[] args)
6      {
7          Scanner reader = new Scanner("40 25");
8          int first = reader.nextInt();
9          int second = reader.nextInt();
10         System.out.println(first - second);
11     }
12 }

```

15

The student may have guessed `java.lang` because `String` and `Math` live there. A class's documentation page shows its package at the top, so check it instead of guessing. Reread [1.7.2 Packages and import](#) if you missed the error.

Question 13

Line	What happens	Value
1	word refers to "library"	
2	"library" has 7 characters	letters = 7
3	<code>Math.abs(7 - 10)</code> is <code>Math.abs(-3)</code>	gap = 3
4–6	Print 7, a space, then 3	

7 3

`Math.abs` turned the negative difference `-3` into its distance from zero, `3`. Reread [1.7.3 Reading API Documentation](#) and [1.7.4 Attributes and Behaviors](#) if your trace differed.

Question 14

B. An import line names the class's full name: the package, a dot, then the class name. For `LocalDate` in `java.time`, that is `import java.time.LocalDate;`.

- **A** is wrong: it names only the package, not the class, so the compiler cannot treat it as importing `LocalDate`.
- **C** is wrong: it puts the parts in reverse order. The package comes first and the class name last.
- **D** is wrong: only classes in `java.lang` are available by default. `java.time` is a different package.

Reread [1.7.2 Packages and import](#) if you chose differently.

Question 15

C. A class is a blueprint and defines a reference type; each string literal is one particular `String` object made from that blueprint, with its own characters.

- **A** is wrong: a library is a collection of many classes, grouped into packages.
- **B** is wrong: documentation tells you how to use a class, such as names, what each method needs, and what it gives back, without showing its code. That is the point of an API.

- **D** is wrong: it is backwards. Attributes are data stored in variables; behaviors are defined by methods.

Reread *1.7.5 Classes Are Types That Make Objects* for C, *1.7.1 Libraries Are Collections of Classes* for A, *1.7.3 Reading API Documentation* for B, and *1.7.4 Attributes and Behaviors* for D.