# PROG 10082
# Object Oriented Programming 1

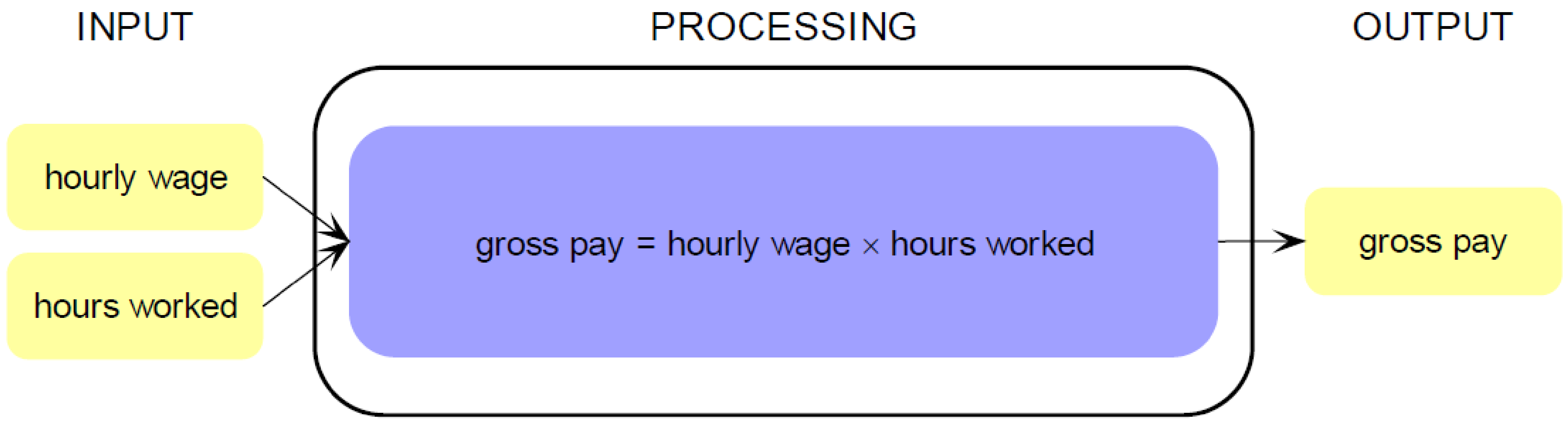
Copyright Sheridan College
May not be published or modified without permission.

## Review

## Example: Worker's Gross Pay

Write a Java application that reads a worker's hourly wage and the number of hours he or she worked and prints his or her pay.

The worker's pay is the number of hours worked times the hourly wage.



The coded Java program is shown below, you need only create one Scanner object from which to read all the data

```
import java.util.Scanner;
public class Pay
{
      public static void main( String [] args )
      {
         // declare data
         double wage; // worker's hourly wage
         double hours; // worker's hours worked
         double pay; // calculated pay
      // build a Scanner object to obtain input
         Scanner in = new Scanner( System.in );
      // prompt for and read worker's data
         System.out.println( "Wage and hours?" );
         wage = in.nextDouble( );
         hours = in.nextDouble( );
      // calculate pay
         pay = hours * wage;
      // print result
         System.out.print( "Pay = $" + pay );
         }
}
```

## Conversion Rules

When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:

1. If one of the operands is double, the other is converted into double.
2. Otherwise, if one of the operands is float, the other is converted into float.
3. Otherwise, if one of the operands is long, the other is converted into long.
4. Otherwise, both operands are converted into int.

## Formatting Console Output

### printf() method

The printf() method is used a little bit differently than print() and println(). The printf() method will take more than one argument or piece of data, unlike print() and println().

The first argument you give printf() is a **format string**: a string value that contains one or more **format specifiers**.

The second and subsequent arguments are numeric values that will be output using the format specifiers in the output string. For example:

```
System.out.printf("Print this
number: %d”, 25);
```

## Formatting Console Output

In this example, a special format specifier **%d** is used in the first argument.

The second argument is the integer value 25. The printf() method will substitute the %d specifier with the value 25.

The "%d" format specifier is used to print integer numbers. Integers are formatted with no decimal places and include the thousands separator when appropriate.

When you give printf() the value you'd like to print, you have to make sure that you always give it the format String first.

This format string can contain anything you'd like to print, but any values that you want formatted should be represented by format specifiers. Other format specifiers are shown in Table 4.11 in chapter 4.6.

After you specify the String you want to output, you then specify the value(s) you want formatted. Separate each argument, including the string, with commas. In the above example, the %d in the string will be replaced by the value 25.

If you want to print a decimal value, you can use the %f specifier, which will print a decimal value with 6 digits after the decimal. For example:

```
System.out.printf(”This is a floating
point value: %f”, 2.345);
```

You can also format more than one value by including more than one format specifier. You must make sure you include an argument for each specifier:

```
System.out.printf("The result of %f times %f
is %f.”,      .25, 1.1,   (.25 * 1.1));
```

The **values and format specifiers** match by the order in which they appear.

In the example above, the first %f will be the formatted value .25, the second %f will be the formatted value 1.1, and the last %f will be the formatted result of (.25 * 1.1).

You can also specify how many digits after the decimal you'd like to display. Between the % and the "f" of the floating point format specifier, you can include information about the number of spaces a value should take up, and the number of digits that should appear after the decimal. For example:

```
System.out.printf(”This is a nicer decimal
value: %4.2f.”, (.25 * 1.1));
```

This will output:

```
This is a nicer decimal value: 0.28.
```

The format specifier "%4.2f" indicates that a floating point number is displayed in a total of 4 character spaces, including 2 digits after the decimal.

What happens if you try it this way:

```
System.out.printf(”This is a nicer decimal value:
%10.2f.”, (.25 * 1.1));
```

You'll get the following result:

```
This is a nicer decimal value:     0.28.
```

If the first number in the format specifier is more space than you need, extra padding will be added to the left of the value to make up the number of character spaces specified. In our example, we specified 10 spaces, but we only required 4, so 6 extra spaces were added. This can come in handy when you want to line up columns of decimal values:

```
System.out.println("Some Values:"); System.out.printf("%9.2f\n", 123.45);
// why is there an "f" after the number in the next statement? // not sure?
take it out and recompile! System.out.printf("%9.2f\n", 1234f);
System.out.printf("%9.2f\n", .12);
System.out.printf("%9.2f\n", 1.2);
```

Try this code! You'll get the following output:

```
Some Values:
   123.45
  1234.00
     0.12
     1.20
```

If you want to make your formatted values left-justified instead of right-justified, add a "-" (minus sign or dash) in front of the total space value:

```
System.out.println("Some Values”);
System.out.printf("Value  1:
System.out.printf("Value  2:
System.out.printf("Value  3:
System.out.printf("Value  4:
```

| | | |
|---|---|---|
| 1: | %-9.2f\n", | 123.45); |
| 2: | %-9.2f\n", | 1234f); |
| 3: | %-9.2f\n", | .12); |
| 4: | %-9.2f\n", | 1.2); |

Output:

Some Values:

```
Value 1: 123.45
Value 2: 1234.00
Value 3: 0.12 Value
4: 1.20
```

**1.** What is the output from each of the following print statements?

1. System.out. printf( '%3d", 5);
2. System.out. printf( '%3d", 12345);
3. System.out. printf( %5.2f", 7.24); %5.2f",
4. System.out. printf( 7.277); %5.2f",
5. System.out. printf( 123.456); %5.2f",
6. System.out. printf( 123.4); '%-5.2f",
7. System.out. printf( 7.277); '%-5.2f",
8. System.out. printf( 123.456);

**1.** What is the output from each of the following print statements? ****NOTE:**

**using the ■ to denote a space**

1. System.out.printf("%3d", 5);

■■5

2. System.out.printf("%3d", 12345);

12345

3. System.out.printf("%5.2f", 7.24);

■7.24

4. System.out.printf("%5.2f", 7.277);

■7.28

5. System.out.printf("%5.2f", 123.456);

123.46

6. System.out.printf("%5.2f", 123.4);

123.40

7. System.out.printf("%-5.2f", 7.277);

7.28^

8. System.out.printf("%-5.2f", 123.456); 123.46

**2.** What are the errors in each of the following print statements?

```
System.out.printf("%d", 23) System.out.printf("%f", 5);
System.out.printf("%4d", 123.4);
System.out.printf("value 1: %d value 2: d", 5, 10);
```

**2.** What are the errors in each of the following print statements?

1. System.out.printf("%d", 23)

Missing a sem-colon!

2. System.out.printf("%f", 5);

%f is for floating-point numbers, but 5 is an int literal.

3. System.out.printf("%4d", 123.4);

%4d is an int format specifier, but 123.4 is a floating-point literal.

4. System.out.printf("value 1: %d value 2: d", 5, 10);

The second "d" is missing the % sign - it's treated as just the letter "d" and the compiler will tell you that the value 10 is missing a format specifier.

## Other Formatting Characters

The format specifier itself follows a specific pattern or format:

**%[flags][width][.precision]typechar**

Each item in [square brackets] is optional.

The named values in the general format above are described as follows:

**flags** - one or more flags that alter the behaviour of the formatted value. See Flags, below.

**width** - the number of total character spaces this formatted value will take. For example, a width of 6 will give you 6 "slots" in which to place the formatted value.

If the number of characters to be formatted is less than the width, the formatted value will be padded with spaces.

If the number of characters to be formatted is more than the width, the formatted value will appear as normal and will NOT be truncated.

**.precision** - a dot/decimal, followed by a numeric value indicates the number of digits to place after the decimal point in the formatted value.

This number, plus the decimal point, is included as part of the number of characters specified in the width parameter.

For example, %6.2f allows for a total of 6 spaces for formatted characters, one of those 6 is for the decimal point, and two of those are for two digits after the decimal. That leaves 3 spots for the digits to the left of the decimal.

**typechar** - a character that specifies which data type is being formatted. See Summary of Common Format Type Specifiers, below.

## Common Type Specifiers

| **Summary of Common Format Type Specifiers** | |
|---|---|
| **Type Specifier** | **Used For** |
| %s | String |
| %d | integers |
| %f | floating-points |
| %c | characters |

In addition, you can use the following specifiers for other values:

**%n** - a platform-specific line separator (will use the system's default line separator character).

**%e, %E** - formats the value in exponential notation. If you use the %e, the E in your value is lower-case. If you use %E, the E will be upper-case. There will always be 6 digits after the decimal point in your formatted value, and the exponent will always take minimum 2 digits and will include the sign + or -.

Example (%n line separators included for clarity):
**System.out.printf("%e%n", 12345.782983);**
**System.out.printf("%E%n", 12345.782983);** ...will yield the output:
**1.234578e+04**
**1.234578E+04**

## Flags

There are a number of special flags you can use after the % sign to change how the formatted value will appear. The list below summarizes a few of them.

**-** (dash, minus sign)
Result is left-aligned (by default, all results are normally right-aligned) Example:

```
System.out.printf("%10s%n", "Java");
System.out.printf("%-10s%n", "Java");
```

...will yield the output:

```
      Java
Java
```

**0**
Allows you to pad a numeric value with leading and trailing zeros.
Example:

```
System.out.printf("%05d%n", 25);
System.out.printf("%05.2f%n",1.2);
```

...will yield the output:

```
00025
01.20
```

Includes the thousands separator in a value that has more than three digits to the left of the decimal point. The comma character is included in the number of character spaces you've indicated using the width parameter. Example:

```
System.out.printf("%,15d%n",10000000);
System.out.printf("%,15.2f%n", 34987.289);
```

...will yield the output:

```
    10,000,000
    34,987.29
```

What is the output from each of the following print statements?

1. System.out.printf("%15e%n", 12.78);
2. System.out.printf("%-15e%n", 12.78);
3. System.out.printf("%15s%n", "Programming");
4. System.out.printf("%-15s%n", "Programming");
5. System.out.printf("%-7.2f%n", 12.78);
6. System.out.printf("%-5d%n", 12);
7. System.out.printf("%03d%n", 0);
8. System.out.printf("%07.1f%n", 12.78);
9. System.out.printf("%,010.1f%n", 12345.67);
10. System.out.printf("%10d%n", 7777);

What is the output from each of the following print statements?

****NOTE: using the ■ to denote a space**

1. System.out.printf("%15e%n", 12.78);

   ■■■1.278000e+01
2. System.out.printf("%-15e%n", 12.78);

   1.278000e+01^M
3. System.out.printf("%15s%n", "Programming");

   ■■■■Programming
4. System.out.printf("%-15s%n", "Programming");

   Programming■■
5. System.out.printf("%-7.2f%n", 12.78);

   12.78H
6. System.out.printf("%-5d%n", 12);

   12^M
7. System.out.printf("%03d%n", 0);

   000
8. System.out.printf("%07.1f%n", 12.78);

   00012.8
9. System.out.printf("%,010.1f%n", 12345.67);

   0012,345.7
10. System.out.printf("%,10d%n", 7777);

    ■■■■■7,777

## Using String.format()

The printf() method is a great way to format console output, but there is an alternative method you can use, which will become especially useful in term 2 when you do GUI programs. You can do this by using the String class's format() method.

String.format() is a method that almost like the printf() method. The String.format() method accepts a string to output and then the list of arguments to format. It then returns the entire string formatted. For example:

```
String output =
      String.format("%3.1fx%3.2f is%4.2f”,.1,1.25, (.1*1.25));
       System.out.println(output);
```

In this example the format() method is given the string "%3.1f x %3.2f is %4.2f" and the argument list .1, 1.25, and (.1*1.25). The first two numeric

arguments will replace the first and second %3.1f and %3.2f respectively, and the result of (.1*1.25) will replace the %4.2f. The format() method will thus return the string "0.1 x 1.25 is 0.13".

This string will be stored in the variable output. Then this string is used in the println() method. The output will appear as:

```
0.1 x 1.25 is 0.13
```

Complete the code to display the values below on the console.

Display one value per line. Each value should be formatted to be right-justified, lined up at the decimal point, and showing only one digit after the decimal point. Use the String.format() function to store your formatted string in a String variable called "output":

**35 28.29 143.12 .8899**

```
public class FormatQuestion {

            public static void main(String[]  args) {
      // finish this line using String.format():


        String output =
      System.out.println(output);
```

Your output should appear as:

```
public class FormatQuestion {

  public static void main(String[] args) {

    // finish this line using String.format():
        String output =

 String.format("%6.2f%n%6.2f%n%6.2f%n%6.2f",

               35.0, 28.29, 143.13, .8899);

       System.out.println(output);

    }

  }
```

## Casting in an Augmented Expression

In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 = (T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct.

**int** sum = **0**;

sum += **4.5**; // sum becomes 4 after this statement

**sum += 4.5** is equivalent to **sum = (int)(sum + 4.5)**.

# Type Casting

Implicit casting

`double d = 3;` (type widening)

Explicit casting

`int i = (int)3.0;` (type narrowing)

`int i = (int)3.9;` (Fraction part is truncated)

What is wrong? int x = 5 / 2.0;

```
range increases
──────────────────────────────►
byte, short, int, long, float, double
```

The Math class is a class that already exists in Java, and it is a part of the java.lang package. You can use the Math class to perform mathematical operations such as absolute value, exponentiation, and square roots.

The Math class consists of **methods** that allow to perform these operations. A method is a chunk of code that performs a task. Some methods require data in order to perform that task. Some methods also give a result after performing a tasks: this is called "returning a value".

For example, a Math method called pow() will calculate a number to the power of another number. The pow() method requires some information before it can give you any results. It needs to know the base number and the exponent. You can give the pow() method these two pieces of information and it will return the result of the base number to the power of the exponent. For example, if you wanted to know the result of $7^2$, then you would give the pow() method 7 and 2. The pow() method would then return the value 49.

Math methods are **static** methods. This means that to invoke them or use them, you need to put the name of the class in front of the method name with a dot or period. For example, to use the pow() method, you refer to it as **Math.pow()**.

the exponent. These inputs are often called **arguments** or **parameters**. The pow() method always knows that the first argument you give it is the base number and the second argument is the exponent! When learning about a new method, you need to check its documentation to see if it requires inputs, and if so, what order those inputs should go in.

If a method returns a value or gives you a result, you need to store that result somewhere. The pow() method returns a double value that is the result of the first argument to the power of the second argument. You will need to store this result in a variable most of the time:

```
double power = Math.pow(7, 2);
```

You can also output the result instead:

```
System.out.println(Math.pow(7, 2));
```

Both of these are correct, but if you need to "remember" the result for use later, you'll need to use a variable as in the first example.

What does the following main method do?

```
public static void main(String[] args) {
      Math.pow(7, 2);
}
```

## Mathematical Functions

Java provides many useful methods in the **Math** class for performing common mathematical functions.

## The `Math` Class

- Class constants:
  - `PI`
  - `E`
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

## Trigonometric Methods

- `sin(double a)`
- `cos(double a)`
- `tan(double a)`
- `acos(double a)`
- `asin(double a)`
- `atan(double a)`

```
Examples:

Math.sin(0) returns 0.0
Math.sin(Math.PI / 6) returns 0.5
Math.sin(Math.PI / 2) returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6) returns
  0.866
Math.cos(Math.PI / 2) returns 0
```

## Exponent Methods

- **`exp(double a)`**
  Returns `e` raised to the power of `a`.
- **`log(double a)`**
  Returns the natural logarithm of `a`.
- **`log10(double a)`**
  Returns the 10-based logarithm of `a`.
- **`pow(double a, double b)`**
  Returns `a` raised to the power of `b`.
- **`sqrt(double a)`**
  Returns the square root of `a`.

```
Examples:


Math.exp(1) returns 2.71
Math.log(2.71) returns 1.0
Math.pow(2, 3) returns 8.0
Math.pow(3, 2) returns 9.0
Math.pow(3.5, 2.5) returns 22.91765
Math.sqrt(4) returns 2.0
Math.sqrt(10.5) returns 3.24
```

# Rounding Methods

- **`double ceil(double x)`**

  x rounded up to its nearest integer. This integer is  returned as a double value.

- **`double floor(double x)`**

  x is rounded down to its nearest integer. This integer is  returned as a double value.

- **`double rint(double x)`**

  x is rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double.

- **`int round(float x)`**

  Return (int)Math.floor(x+0.5).

- **`long round(double x)`**

  Return (long)Math.floor(x+0.5).

# Rounding Methods Examples

```
Math.ceil(2.1) returns 3.0

Math.ceil(2.0) returns 2.0

Math.ceil(-2.0) returns –2.0

Math.ceil(-2.1) returns -2.0

Math.floor(2.1) returns 2.0

Math.floor(2.0) returns 2.0

Math.floor(-2.0) returns –2.0

Math.floor(-2.1) returns -3.0

Math.rint(2.1) returns 2.0

Math.rint(2.0) returns 2.0

Math.rint(-2.0) returns –2.0

Math.rint(-2.1) returns -2.0

Math.rint(2.5) returns 2.0

Math.rint(-2.5) returns -2.0

Math.round(2.6f) returns 3

Math.round(2.0) returns 2

Math.round(-2.0f) returns -2

Math.round(-2.6) returns -3
```

# min, max, and abs

- `max(a, b)` and `min(a, b)`

  Returns the maximum or minimum of two parameters.

- `abs(a)`

  Returns the absolute value of the parameter.

- `random()`

  Returns a random `double` value in the range [0.0, 1.0).

```
Examples:

Math.max(2, 3) returns 3
Math.max(2.5, 3) returns 3.0
Math.min(2.5, 3.6) returns
  2.5
Math.abs(-2) returns 2
Math.abs(-2.1) returns 2.1
```

## The random Method

Generates a random double value greater than or equal to 0.0 and less than 1.0 (0 <= Math.random() < 1.0).

Examples:

`(int)(Math.random() * 10)` $\longrightarrow$ Returns a random integer between 0 and 9.

`50 + (int)(Math.random() * 50)` $\longrightarrow$ Returns a random integer between 50 and 99.

In general,

`a + Math.random() * b` $\longrightarrow$ Returns a random number between a and a + b, excluding a + b.

## The String Type

In programming, when we are referring to textual data, we use the word **string**.

A string is any set of letters, numbers, or symbols that are considered "text" and are not for calculation.

The char type only represents one character. To represent a string of characters, use the data type called String. For example,

```
String message = "Welcome to Java";
```

String is actually a predefined class in the Java library just like the System class and JOptionPane class. The String type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in Chapter 8, “Objects and Classes.” For the time being, you just need to know how to declare a String variable, how to assign a string to the variable, and how to concatenate strings.

## Strings

A customer's phone number is usually stored as a string; even though a phone number can be stored as just a set of digits (your program code could format a phone number with brackets and dashes, so you don't need to store those), you don't calculate with phone numbers (e.g. you don't add two phone numbers together or multiply a phone number by some other value).

The same is sometimes true of ID numbers such as a product ID or student ID. Sometimes if a programmer is concerned with saving storage space, s/he might store a phone number or ID value as an integer instead of a string.

To help differentiate between strings and chars, study these examples:

Examples of **char** types:

'a'
'Z'
'2'
'\n'

- '\0' (backslash-zero, the null value for char)

Examples of strings:

"hello"
"Hello, World!"
"123.456"
"123 is a number but this is a string."

- "\n"

Note that it doesn't matter if the character or string is made up of digits! Any digits enclosed in quotes is either a character or a string, not a number. For example, 123.456 is a number, but "123.456" is a string. Note that **'123.456' is invalid** because a **char** type can consist of only one character or escape sequence.

## String Concatenation

Recall that you can perform the **concatenation** operation on a String using the + operator. In programming, using + on Strings simply places the strings next to each other. For example:

```
String si = "Hello";
String s2 = "World";
String s3 = si + s2;
System.out.println(s3);
```

## String Concatenation

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB

String s4 = ”10 plus 2 is ” + 10 + 2;
System.out.println(s4); // 10 plus 2 is 102
```

## String Concatenation

```
String s4 = ”10 plus 2 is ” + (10 + 2);
//You should then see the proper output of:
// 10 plus 2 is 12
```

```
String s4 = 10 + 2 + ” is 10 plus 2”;
System.out.println(s4); // 12 is 10 plus 2
```

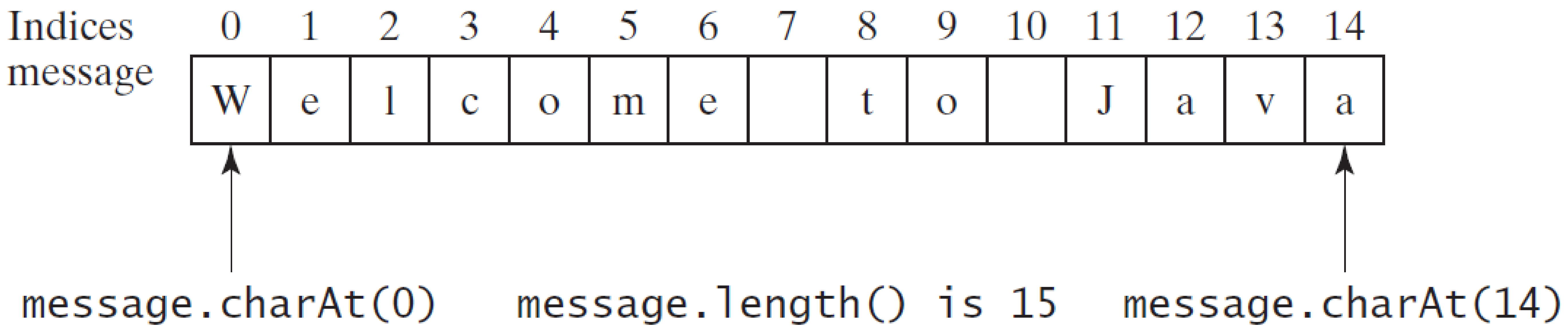
# Methods in the Character Class

| **Method** | **Description** |
|---|---|
| `isDigit(ch)` | Returns true if the specified character is a digit. |
| `isLetter(ch)` | Returns true if the specified character is a letter. |
| `isLetterOfDigit(ch)` | Returns true if the specified character is a letter or digit. |
| `isLowerCase(ch)` | Returns true if the specified character is a lowercase letter. |
| `isUpperCase(ch)` | Returns true if the specified character is an uppercase letter. |
| `toLowerCase(ch)` | Returns the lowercase of the specified character. |
| `toUpperCase(ch)` | Returns the uppercase of the specified character. |

# Simple Methods for **String** Objects

| Method | Description |
|---|---|
| `length()` | Returns the number of characters in this string. |
| `charAt(index)` | Returns the character at the specified index from this string. |
| `concat(s1)` | Returns a new string that concatenates this string with string s1. |
| `toUpperCase()` | Returns a new string with all letters in uppercase. |
| `toLowerCase()` | Returns a new string with all letters in lowercase. |
| `trim()` | Returns a new string with whitespace characters trimmed on both sides. |

# Getting Characters from a String



String message = "**Welcome to Java**";

System.out.println("**The first character in message is “** + message.charAt(0));

## Converting Strings

"Welcome".toLowerCase() returns a new string, welcome.

"Welcome".toUpperCase() returns a new string, WELCOME.

"  Welcome  ".trim() returns a new string, Welcome.

# Reading a String from the Console

```
Scanner input = new Scanner(System.in);
System.out.print("Enter three words separated by spaces: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

# Reading a Character from the Console

```
Scanner input = new Scanner(System.in);
System.out.print("Enter a character: ");
String s = input.nextLine();
char ch = s.charAt(0);
System.out.println("The character entered is " + ch);
```

# Comparing Strings

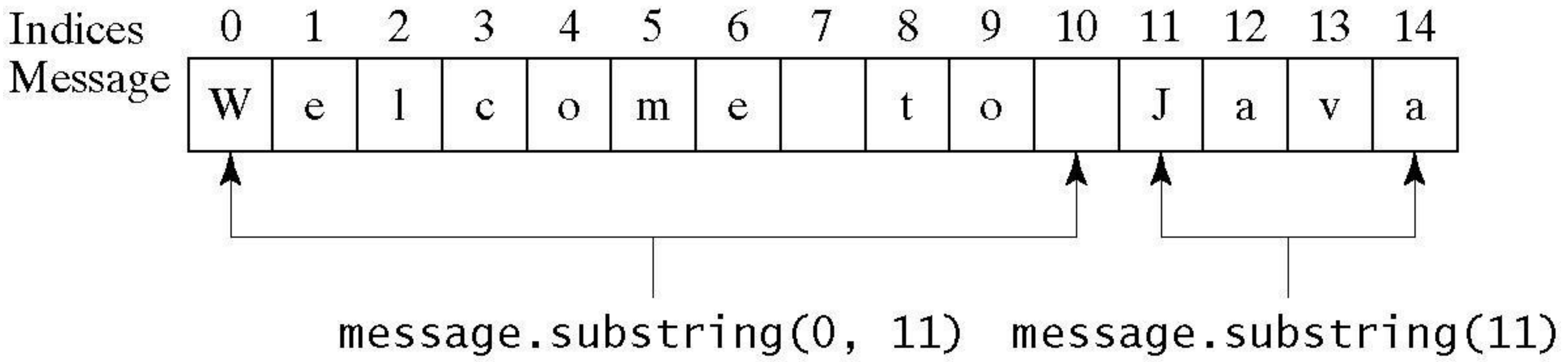
| Method | Description |
|---|---|
| `equals(s1)` | Returns true if this string is equal to string `s1`. |
| `equalsIgnoreCase(s1)` | Returns true if this string is equal to string `s1`; it is case insensitive. |
| `compareTo(s1)` | Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or less than `s1`. |
| `compareToIgnoreCase(s1)` | Same as `compareTo` except that the comparison is case insensitive. |
| `startsWith(prefix)` | Returns true if this string starts with the specified prefix. |
| `endsWith(suffix)` | Returns true if this string ends with the specified suffix. |

OrderTwoCities Run

https://liveexample.pearsoncmg.com/html/OrderTwoCities.html

# Obtaining Substrings

| Method | Description |
|---|---|
| `substring(beginIndex)` | Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2. |
| `substring(beginIndex, endIndex)` | Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring. |

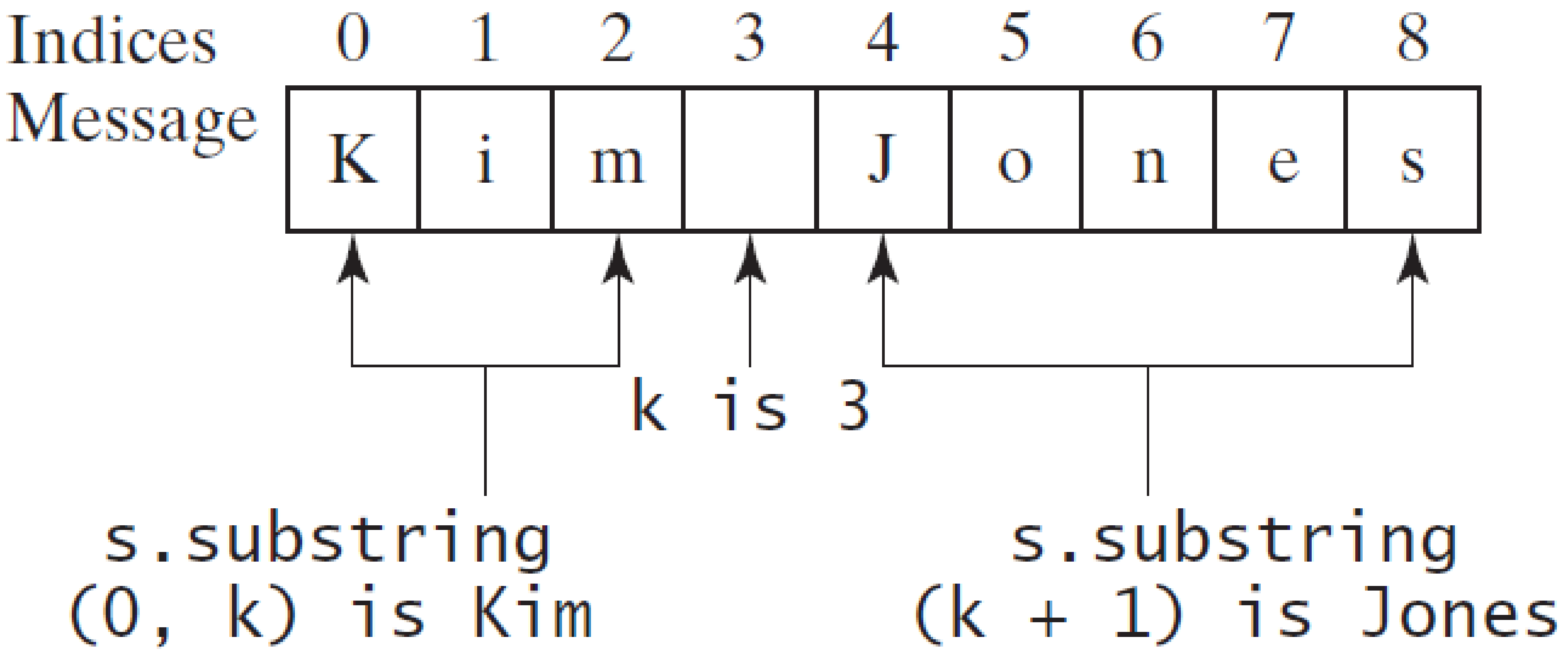


# Finding a Character or a Substring in a String

| Method | Description |
|---|---|
| `indexOf(ch)` | Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched. |
| `indexOf(ch, fromIndex)` | Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched. |
| `indexOf(s)` | Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched. |
| `indexOf(s, fromIndex)` | Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched. |
| `lastIndexOf(ch)` | Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched. |
| `lastIndexOf(ch, fromIndex)` | Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched. |
| `lastIndexOf(s)` | Returns the index of the last occurrence of string `s`. Returns `-1` if not matched. |
| `lastIndexOf(s, fromIndex)` | Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched. |

# Finding a Character or a Substring in a String

```
int k = s.indexOf(' ');
String firstName = s.substring(0, k);
String lastName = s.substring(k + 1);
```



## Conversion between Strings and Numbers

```
int intValue = Integer.parseInt(intString);
double doubleValue = Double.parseDouble(doubleString);

String s = number + "";
```

## Formatting Output

Use the printf statement.

```
System.out.printf(format, items);
```

Where format is a string that may consist of substrings and format specifiers.

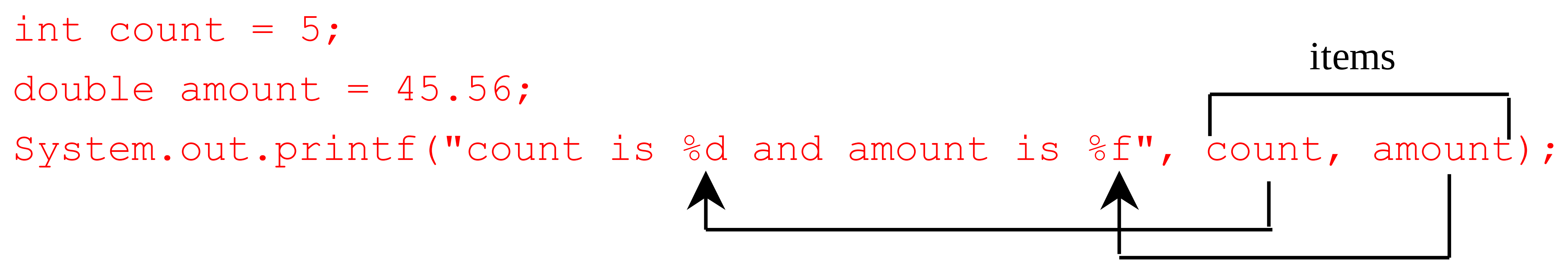
A format specifier specifies how an item should be displayed.

An item may be a numeric value, character, boolean value, or a string.

Each specifier begins with a percent sign.

# Frequently-Used Specifiers

| Specifier | Output | Example |
|---|---|---|
| %b | a boolean value | true or false |
| %c | a character | 'a' |
| %d | a decimal integer | 200 |
| %f | a floating-point number | 45.460000 |
| %e | a number in standard scientific notation | 4.556000e+01 |
| %s | a string | "Java is cool" |



```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

```
display             count is 5 and amount is 45.560000
```

## Selections

## The `boolean` Type and Operators

Often in a program you need to compare two values, such as whether i is greater than j. Java provides six comparison operators (also known as relational operators) that can be used to compare two values. The result of the comparison is a Boolean value: true or false.

```
boolean b = (1 > 2);
```

# Relational Operators

| Java Operator | Mathematics Symbol | Name | Example (radius is 5) | Result |
|---|---|---|---|---|
| `<` | $<$ | less than | `radius < 0` | `false` |
| `<=` | $\leq$ | less than or equal to | `radius <= 0` | `false` |
| `>` | $>$ | greater than | `radius > 0` | `true` |
| `>=` | $\geq$ | greater than or equal to | `radius >= 0` | `true` |
| `==` | $=$ | equal to | `radius == 0` | `false` |
| `!=` | $\neq$ | not equal to | `radius != 0` | `true` |

## Note

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

# Simple if Demo

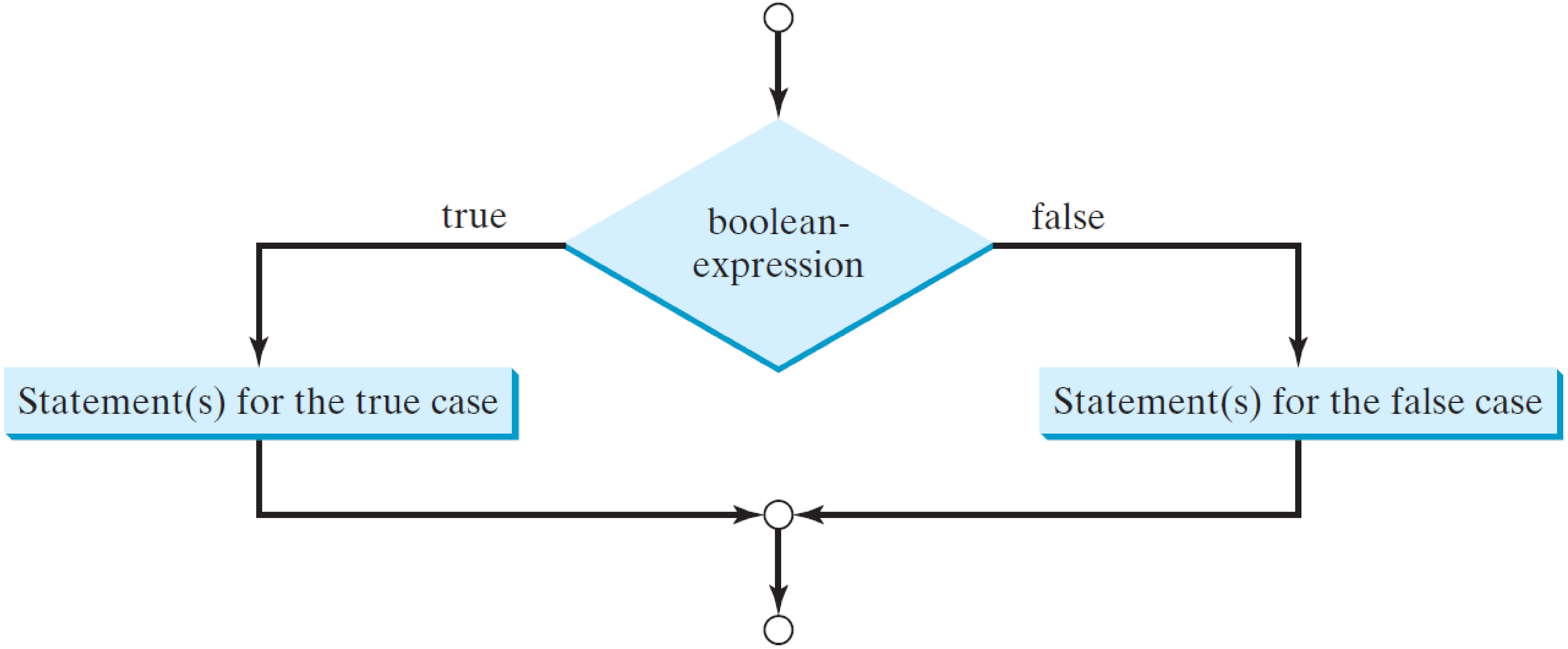
Write a program that prompts the user to enter an integer. If the number is a multiple of 5, print HiFive. If the number is divisible by 2, print HiEven.

SimpleIfDemo Run

**https://liveexample.pearsoncmg.com/html/SimpleIfDemo.html**

## The Two-way `if` Statement

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
```


true
boolean-expression
false
Statement(s) for the true case
Statement(s) for the false case

# `if-else` Example

```
if (radius >= 0) {
  area = radius * radius * 3.14159;

 System.out.println("The area for the “
    + “circle of radius " + radius +
    " is " + area);
}
else {
  System.out.println("Negative input");
}
```

## Multiple Alternative if Statements

```
if (score >= 90.0)
  System.out.print("A");
else
  if (score >= 80.0)
    System.out.print("B");
  else
    if (score >= 70.0)
      System.out.print("C");
    else
      if (score >= 60.0)
        System.out.print("D");
      else
        System.out.print("F");
```

(a)

Equivalent

This is better

```
if (score >= 90.0)
  System.out.print("A");
else if (score >= 80.0)
  System.out.print("B");
else if (score >= 70.0)
  System.out.print("C");
else if (score >= 60.0)
  System.out.print("D");
else
  System.out.print("F");
```

(b)

animation

# Trace if-else statement

Suppose score is 70.0

The condition is false

```
if (score >= 90.0)
  System.out.print("A");
else if (score >= 80.0)
  System.out.print("B");
else if (score >= 70.0)
  System.out.print("C");
else if (score >= 60.0)
  System.out.print("D");
else
  System.out.print("F");
```

# Common Errors

Adding a semicolon at the end of an if clause is a common mistake.

```
if (radius >= 0);    ← Wrong
{
  area = radius*radius*PI;
  System.out.println(
    "The area for the circle of radius " +
    radius + " is " + area);
}
```

This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.

This error often occurs when you use the next-line block style.

# CAUTION

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

## Logical Operators

| Operator | Name | Description |
|---|---|---|
| **!** | not | logical negation |
| **&&** | and | logical conjunction |
| \|\| | or | logical disjunction |
| ^ | exclusive or | logical exclusion |

## Examples

Here is a program that checks whether a number is divisible by 2 and 3, whether a number is divisible by 2 or 3, and whether a number is divisible by 2 or 3 but not both:

TestBooleanOperators

Run

**https://liveexample.pearsoncmg.com//html/SubtractionQuiz.html**

## Examples

```
System.out.println("Is " + number + " divisible by 2 and 3? " +
  ((number % 2 == 0) && (number % 3 == 0)));

System.out.println("Is " + number + " divisible by 2 or 3? " +
  ((number % 2 == 0) || (number % 3 == 0)));

System.out.println("Is " + number +
  " divisible by 2 or 3, but not both? " +
  ((number % 2 == 0) ^ (number % 3 == 0)));
```

TestBooleanOperators

Run

https://liveexample.pearsoncmg.com//html/SubtractionQuiz.html

## Comparing and Testing Characters

```
if (ch >= 'A' && ch <= 'Z')
  System.out.println(ch + " is an uppercase letter");
else
      if (ch >= 'a' && ch <= 'z')
            System.out.println(ch + " is a lowercase letter");
      else
           if (ch >= '0' && ch <= '9')
                 System.out.println(ch + " is a numeric character");
```

# Operator Precedence

- `var++, var--`
- `+, - (Unary plus and minus), ++var,--var`
- `(type) Casting`
- `! (Not)`
- `*, /, % (Multiplication, division, and remainder)`
- `+, - (Binary addition and subtraction)`
- `<, <=, >, >= (Relational operators)`
- `==, !=; (Equality)`
- `^ (Exclusive OR)`
- `&& (Conditional AND) Short-circuit AND`
- `|| (Conditional OR) Short-circuit OR`
- `=, +=, -=, *=, /=, %= (Assignment operator)`