جامعة الجلالة
GALALA UNIVERSITY

# CSE110 Principles of Programming

## Lecture 3: Data Types and Input output

Professor Shaker El-Sappagh

Shaker.elsappagh@gu.edu.eg

Fall 2023

# Outline

1. Variables and Literals
2. Primitive Data Types
3. Arithmetic Operators
4. Combined Assignment Operators
5. Conversion between Primitive Data Types
6. Creating Named Constants with final
7. The String Class
8. Scope
9. Programming Style

# Variables and Literals

- **A variable** is a named storage location in the computer's memory. **A literal** is a value that is written into the code of a program.
- Every variable has a **name**, a **type**, a **size** (in bytes) and a **value**.

```
1  // This program has a variable.
2
3  public class Variable2
4  {
5     public static void main(String[] args)
6     {
7        int value;
8
9        value = 5;
10       System.out.print("The value is ");
11       System.out.println("value");
12    }
13 }
```

**Program Output**

```
The value is value
```

```
1  // This program has a variable.
2
3  public class Variable
4  {
5     public static void main(String[] args)
6     {
7        int value;
8
9        value = 5;
10       System.out.print("The value is ");
11       System.out.println(value);
12    }
13 }
```

**Program Output**

```
The value is 5
```

# Variables and Literals

**Displaying Multiple items with the + operator**

- When the + operator is used with strings, it is known as the string concatenation operator.

```
System.out.println("This is " + "one string.");
```

This statement will print:

```
This is one string.
```

- You can also use the + operator to concatenate the contents of a variable to a string.

```
number = 5;
System.out.println("The value is " + number);
The value is 5
```

- Although *number* is not a string, the + operator converts its value to a string and then concatenates that value with the first string.

# Variables and Literals

**Displaying Multiple items with the + operator**

- A string literal cannot begin on one line and end on another.

```
// This is an error!
System.out.println("Enter a value that is greater than zero
                    and less than 10." );
```

- Remedy this problem by breaking the argument up into smaller string literals, and then using the string concatenation operator to spread them out over more than one line.

```
System.out.println("Enter a value that is " +
                   "greater than zero and less " +
                   "than 10." );

sum = 249;
System.out.println("The sum of the three " +
                   "numbers is " + sum);
```

# Variables and Literals

**Be careful with Quotation Marks**

- placing double quotation marks around anything that is not intended to be a string literal will create an error of some type.

```
int value;

value = "5";    // Error!
```

- In this statement, 5 is no longer an integer, but a string literal. Because value was declared as an integer variable, you can only store integers in it. In other words, 5 and “5” are not the same thing.

# Variables and Literals

## More about Literals

- A literal is a **value** that is **written in the code of a program**. Literals are commonly **assigned to variables** or **displayed**.

```
1  // This program has literals and a variable.
2
3  public class Literals
4  {
5     public static void main(String[] args)
6     {
7        int apples;
8
9        apples = 20;
10       System.out.println("Today we sold " + apples +
11                          " bushels of apples.");
12    }
13 }
```

| Literal | Type of Literal |
|---|---|
| 20 | Integer literal |
| "Today we sold " | String literal |
| " bushels of apples." | String literal |

**Program Output**

```
Today we sold 20 bushels of apples.
```

# Variables and Literals

## Identifiers

- An identifier is a **programmer-defined name** that represents some **element** of a program.
- **Variable, Constants, Methods,** and **Class names** are examples of identifiers.
- You may **choose your own variable names** and class names in Java, as long as you do **not use any of the Java key words**.
- You should always choose names for your variables that give an indication of what they are used for. This method of coding helps produce *self-documenting programs*.

```
int x;

int itemsOrdered;
```

- The first character must be one of the letters a–z or A–Z, an underscore (_), or a dollar sign ($).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (_), or dollar signs ($).
- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Identifiers cannot include spaces.

# Variables and Literals

Examine the following program.

```
// This program uses variables and literals.
public class BigLittle
{
   public static void main(String[] args)
   {
      int little;
      int big;
      little = 2;
      big = 2000;
      System.out.println("The little number is " + little);
      System.out.println("The big number is " + big);
   }
}
```

List the variables and literals found in the program.

# Variables and Literals

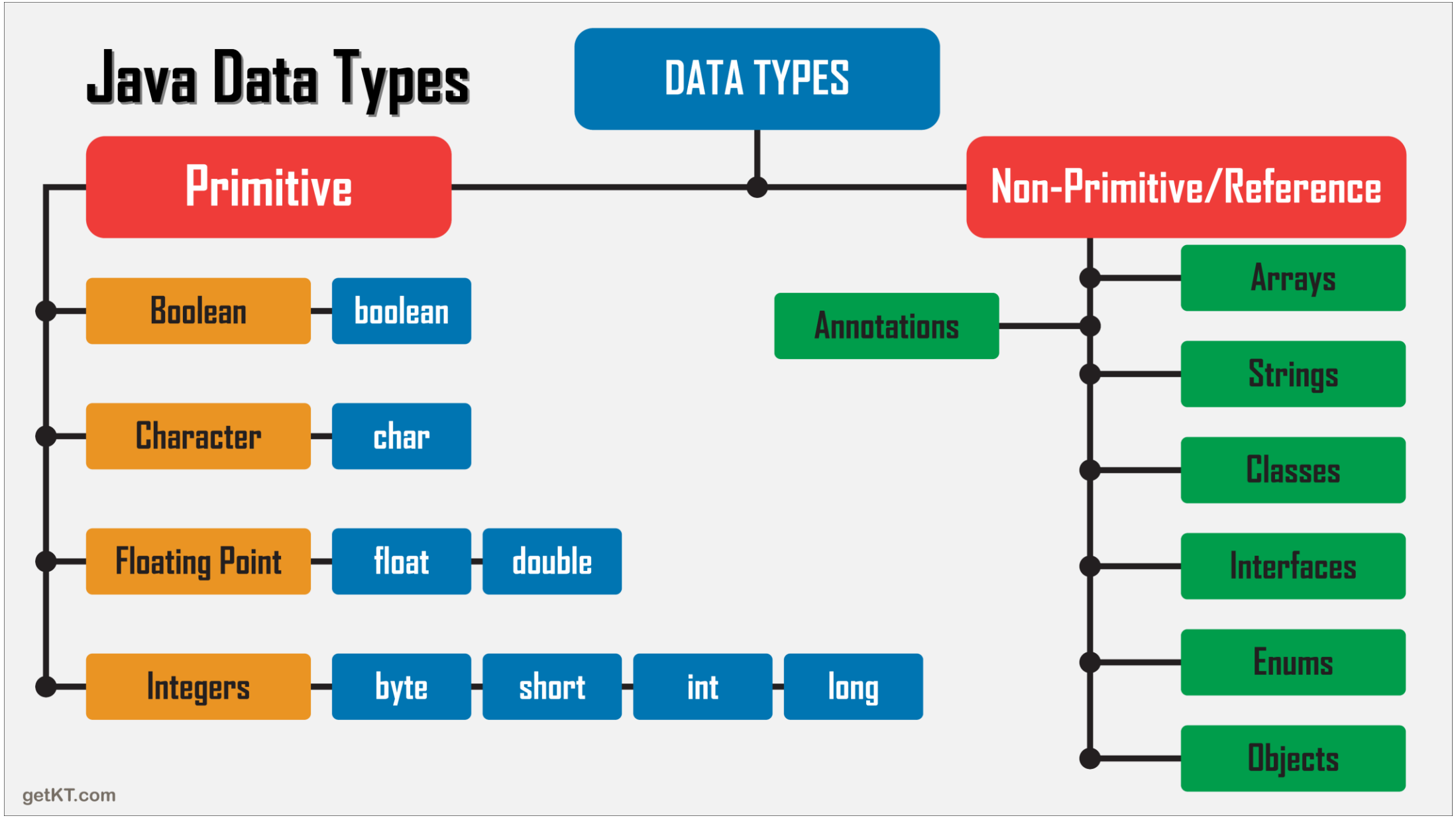
What will the following program display on the screen?

```
public class CheckPoint
{
   public static void main(String[] args)
   {
      int number;
      number = 712;
      System.out.println("The value is " + "number");
   }
}
```

# Primitive Data Types

- Computer programs **collect pieces of data** from the real world and manipulate them in various ways. There are many different **types of data**. In the realm of **numeric data**, for example, there are whole and fractional numbers, negative and positive numbers, and numbers so large and others so small that they don't even have a name. Then there is **textual information**. Names and addresses, for instance, are stored as strings of characters.
- When you write a program you **must determine what types of data** it is likely to encounter.
- There are **many different types of data**. Variables are **classified according to** their data type, which determines the kind of data that may be stored in them.
- **Each variable has a data type**, which is the type of data that the variable can hold. Selecting
- Data type determines the **amount of memory** the variable uses, and the way the variable formats and stores data.
- Select a data type that is appropriate for the type of data that your program.

# Primitive Data Types



# Primitive Data Types

| Data Type | Size | Range |
|---|---|---|
| `byte` | 1 byte | Integers in the range of –128 to +127 |
| `short` | 2 bytes | Integers in the range of –32,768 to +32,767 |
| `int` | 4 bytes | Integers in the range of –2,147,483,648 to +2,147,483,647 |
| `long` | 8 bytes | Integers in the range of –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 |
| `float` | 4 bytes | Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy |
| `double` | 8 bytes | Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy |

***DataType* is the name of the data type and *VariableName* is the name of the variable. Examples:**

```
byte inches;
int speed;
short month;
float salesCommission;
double distance;
```

# Primitive Data Types: The integer Data Types

```
1  // This program has variables of several of the integer types.
2
3  public class IntegerVariables
4  {
5     public static void main(String[] args)
6     {
7        int checking; // Declare an int variable named checking.
8        byte miles; // Declare a byte variable named miles.
9        short minutes; // Declare a short variable named minutes.
10       long days; // Declare a long variable named days.
11
12       checking = -20;
13       miles = 105;
14       minutes = 120;
15       days = 189000;
16       System.out.println("We have made a journey of " + miles +
17                          " miles.");
18       System.out.println("It took us " + minutes + " minutes.");
19       System.out.println("Our account balance is $" + checking);
20       System.out.println("About " + days + " days ago Columbus " +
21                          "stood on this spot.");
22    }
23 }
```

**Program Output**

```
We have made a journey of 105 miles.
It took us 120 minutes.
Our account balance is $-20
About 189000 days ago Columbus stood on this spot.
```

# Primitive Data Types

- In most programs you will need more than one variable of any given data type. If a program uses three integers, length, width, and area, they could be declared separately, as follows:

```
int length;
int width;
int area;
```

- It is easier, however, to combine the three variable declarations:

```
int length, width, area;
```

- integer Literals:

```
int checking= -20;
```

- You can **force** an integer literal to be treated as a long by **suffixing** it with the letter L. For example, the value 57L would be treated as a long. you can use either an **uppercase or a lowercase L**, but upper case is preferred.

# Primitive Data Types: Floating-point Data Types

- They are float and double.
- The float data type is considered a **single precision data type**. It can store a floating-point number with **7 digits of accuracy**.
- The double data type is considered a **double precision data type**. It can store a floating-point number with **15 digits of accuracy**.
- The double data type uses **twice as much memory** as the float data type.
- A float variable occupies **4 bytes** of memory, whereas a double variable uses **8 bytes**.

# Primitive Data Types: Floating-point Data Types

```
1  // This program demonstrates the double data type.
2
3  public class Sale
4  {
5     public static void main(String[] args)
6     {
7        double price, tax, total;
8
9        price = 29.75;
10       tax = 1.76;
11       total = 31.51;
12       System.out.println("The price of the item " +
13                          "is " + price);
14       System.out.println("The tax is " + tax);
15       System.out.println("The total is " + total);
16    }
17 }
```

**Program Output**

```
The price of the item is 29.75
The tax is 1.76
The total is 31.51
```

# Primitive Data Types: Floating-point Data Types

**Floating-point Literals:**

- When you write a floating-point **literal** in your program code, Java assumes it to be of the double data type.
- Because of this, a **problem can arise** when assigning a floating-point literal to a float variable.

```
float number;
number = 23.5;              // Error!
```

- Java is a **strongly typed language**, which means that it only allows you to store values of **compatible** data types in variables.
- A double value **is not compatible with** a float variable because a double can be much larger or much smaller than the allowable range for a float.
- You can force a double literal to be treated as a float, however, by suffixing it with the letter F or f.

```
float number;
number = 23.5F;             // This will work.
```

# Primitive Data Types: Floating-point Data Types

## scientific and E notation

Floating-point literals can be represented in scientific notation. Take the number 47,281.97. In scientific notation this number is $4.728197 \times 10^4$. ($10^4$ is equal to 10,000, and $4.728197 \times 10{,}000$ is 47,281.97.)

Java uses E notation to represent values in scientific notation. In E notation, the number $4.728197 \times 10^4$ would be 4.728197E4. Table 2-6 shows other numbers represented in scientific and E notation. **NOTE:** The E can be uppercase or lowercase.

| Decimal Notation | Scientific Notation | E Notation |
|---|---|---|
| 247.91 | $2.4791 \times 10^2$ | 2.4791E2 |
| 0.00072 | $7.2 \times 10^{-4}$ | 7.2E–4 |
| 2,900,000 | $2.9 \times 10^6$ | 2.9E6 |

# Primitive Data Types: The boolean Data Type

- The boolean data type allows you to create variables that may hold one of two possible values: true or false.

```
1  // A program for demonstrating boolean variables
2
3  public class TrueFalse
4  {
5     public static void main(String[] args)
6     {
7        boolean bool;
8
9        bool = true;
10       System.out.println(bool);
11       bool = false;
12       System.out.println(bool);
13    }
14 }
```

**Program Output**

```
true
false
```

- Variables of the boolean data type are useful for **evaluating conditions** that are either true or false.
- boolean variables may hold **only** the value true or false.
- The contents of a boolean variable **may not be copied** to a variable of any type other than boolean.

# Primitive Data Types: The char Data Type

- The char data type is used to store characters. A variable of the char data type can hold **one character at a time**.
- Character literals are enclosed in **single quotation marks**.

```
1  // This program demonstrates the char data type.
2
3  public class Letters
4  {
5     public static void main(String[] args)
6     {
7        char letter;
8
9        letter = 'A';
10       System.out.println(letter);
11       letter = 'B';
12       System.out.println(letter);
13    }
14 }
```

**Program Output**

```
A
B
```

# Primitive Data Types: The char Data Type

**Unicode**

- Characters are **internally represented by numbers**.
- Each printable character, as well as many non-printable characters, is assigned a unique number.
- **Java uses Unicode**, which is a set of numbers that are used as codes for representing characters.
- Each Unicode number **requires two bytes** of memory, so char variables occupy two bytes.
- When a character is stored in memory, it is **actually the numeric code that is stored**.
- When the computer is instructed to print the value on the screen, it displays the **character that corresponds with the numeric code**.

# Primitive Data Types: The char Data Type

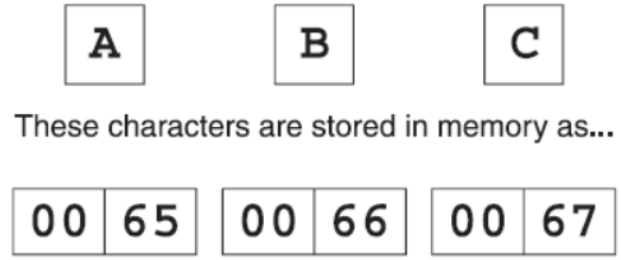
## Unicode

```
1  // This program demonstrates the close relationship between
2  // characters and integers.
3
4  public class Letters2
5  {
6     public static void main(String[] args)
7     {
8        char letter;
9
10       letter = 65;
11       System.out.println(letter);
12       letter = 66;
13       System.out.println(letter);
14    }
15 }
```

**Program Output**

```
A
B
```

Characters and how they are stored in memory



# Primitive Data Types: The char Data Type

## Unicode

| Code | Character | Code | Character | Code | Character | Code | Character | Code | Character |
|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | 26 | SUB | 52 | 4 | 78 | N | 104 | h |
| 1 | SOH | 27 | Escape | 53 | 5 | 79 | O | 105 | i |
| 2 | STX | 28 | FS | 54 | 6 | 80 | P | 106 | j |
| 3 | ETX | 29 | GS | 55 | 7 | 81 | Q | 107 | k |
| 4 | EOT | 30 | RS | 56 | 8 | 82 | R | 108 | l |
| 5 | ENQ | 31 | US | 57 | 9 | 83 | S | 109 | m |
| 6 | ACK | 32 | (Space) | 58 | : | 84 | T | 110 | n |
| 7 | BEL | 33 | ! | 59 | ; | 85 | U | 111 | o |
| 8 | Backspace | 34 | " | 60 | < | 86 | V | 112 | p |
| 9 | HTab | 35 | # | 61 | = | 87 | W | 113 | q |
| 10 | Line Feed | 36 | $ | 62 | > | 88 | X | 114 | r |
| 11 | VTab | 37 | % | 63 | ? | 89 | Y | 115 | s |
| 12 | Form Feed | 38 | & | 64 | @ | 90 | Z | 116 | t |
| 13 | CR | 39 | ' | 65 | A | 91 | [ | 117 | u |
| 14 | SO | 40 | ( | 66 | B | 92 | \ | 118 | v |
| 15 | SI | 41 | ) | 67 | C | 93 | ] | 119 | w |
| 16 | DLE | 42 | * | 68 | D | 94 | ^ | 120 | x |
| 17 | DC1 | 43 | + | 69 | E | 95 | _ | 121 | y |
| 18 | DC2 | 44 | , | 70 | F | 96 | ` | 122 | z |
| 19 | DC3 | 45 | - | 71 | G | 97 | a | 123 | { |
| 20 | DC4 | 46 | . | 72 | H | 98 | b | 124 | \| |
| 21 | NAK | 47 | / | 73 | I | 99 | c | 125 | } |
| 22 | SYN | 48 | 0 | 74 | J | 100 | d | 126 | ~ |
| 23 | ETB | 49 | 1 | 75 | K | 101 | e | 127 | DEL |
| 24 | CAN | 50 | 2 | 76 | L | 102 | f | | |
| 25 | EM | 51 | 3 | 77 | M | 103 | g | | |

# Primitive Data Types: Variable assignment and initialization

- A value is put into a variable with an assignment statement.

```
unitsSold = 12;
```

- The = symbol is called the **assignment operator**. Operators perform operations on data. The data that operators work with are called **operands**.
- In an assignment statement, the **name of the variable** receiving the assignment must appear on the **left side** of the operator, and the **value** being assigned must appear on the **right side**.

```
12 = unitsSold;      // ERROR!
```

- The operand on **the left side** of the = operator must be a **variable name**. The operand on the **right side** of the = symbol must be **an expression** that has a value.

```
length = 20;
width = length;
```

- You may also assign values to variables **as part of the declaration statement**. This is known as **initialization**.

```
int month = 2, days = 28;
```

# Primitive Data Types: Variable assignment and initialization

- Assignment operators are used to "assign" (give) values to variables.

| | |
|---|---|
| SYNTAX | variable = literal; |
| Example 1 | x = 10; // x is previously declared |
| SYNTAX | variable1 = variable2; |
| Example 2 | x = 10; // x is previously declared<br>y = 15; // y is previously declared<br>x = y; // x=15 y=15 |
| SYNTAX | variable = expression; |
| Example 3 | x = 10; // x is previously declared<br>y = 15; // y is previously declared<br>x = x * y; // x=10*15 ➔ x=150 |

# Primitive Data Types: Variable assignment and initialization

- **Variables Hold only one Value at a Time**
- A variable can hold **only one value at a time**.
- When you assign a new value to a variable, the new value **takes the place of** the variable's previous contents.

```
int x = 5;
System.out.println(x);
x = 99;
System.out.println(x);
```

# Primitive Data Types: Variable assignment and initialization

2.12 Which of the following are illegal variable names and why?

```
x
99bottles
july97
theSalesFigureForFiscalYear98
r&d
grade_report
```

2.13 Is the variable name `Sales` the same as `sales`? Why or why not?

2.14 Refer to the Java primitive data types listed in Table 2-5 for this question.

a) If a variable needs to hold whole numbers in the range 32 to 6,000, what primitive data type would be best?

b) If a variable needs to hold whole numbers in the range –40,000 to +40,000, what primitive data type would be best?

c) Which of the following literals use more memory? `22.1` or `22.1F`?

2.15 How would the number $6.31 \times 10^{17}$ be represented in E notation?

2.16 A program declares a `float` variable named `number`, and the following statement causes an error. What can be done to fix the error?

```
number = 7.4;
```

# Primitive Data Types: Variable assignment and initialization

2.17 What values can `boolean` variables hold?

2.18 Write statements that do the following:

a) Declare a `char` variable named `letter`.

b) Assign the letter A to the `letter` variable.

c) Display the contents of the `letter` variable.

2.19 What are the Unicode codes for the characters 'C', 'F', and 'W'? (You may need to refer to Appendix B on the book's companion Web site, at www.pearsonglobaleditions.com/Gaddis.)

2.20 Which is a character literal, `'B'` or `"B"`?

2.21 What is wrong with the following statement?

```
char letter = "Z";
```

# Arithmetic operators

| Operator | Meaning | Type | Example |
|---|---|---|---|
| + | Addition | Binary | `total = cost + tax;` |
| - | Subtraction | Binary | `cost = total - tax;` |
| * | Multiplication | Binary | `tax = cost * rate;` |
| / | Division | Binary | `salePrice = original / 2;` |
| % | Modulus | Binary | `remainder = value % 3;` |

# Associativity of arithmetic operators

| Operator | Associativity |
|---|---|
| - (unary negation) | Right to left |
| * / % | Left to right |
| + - | Left to right |

# Arithmetic operators

| | | |
|---|---|---|
| Example 1 | 3 * 7 – 6 + 2 * 5 / 4 + 6 | No parenthesis, look for * / % |
| Step 1 | 3 * 7 – 6 + 2 * 5 / 4 + 6 | Evaluate from left to right |
| Step 2 | 21 – 6 + 10 / 4 + 6 | This is an integer division |
| Step 3 | 21 – 6 + 2 + 6 | Evaluate from left to right |
| Step 4 | 21 – 6 + 2 + 6 | Evaluate from left to right |
| Step 5 | 15 + 2 + 6 | Evaluate from left to right |
| Step 6 | 17 + 6 | Evaluate from left to right |
| Step 7 | 23 | Final result |

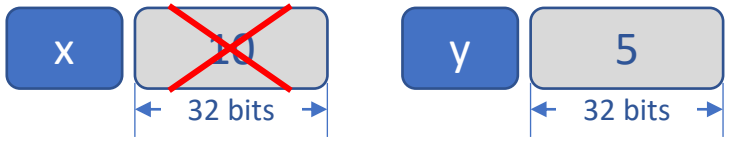
This is equivalent to: (((3*7)-6)+((2*5)/4))+6

# Combined assignment operators

| Operator | Example Usage | Equivalent To |
|---|---|---|
| += | x += 5; | x = x + 5; |
| -= | y -= 2; | y = y - 2; |
| *= | z *= 10; | z = z * 10; |
| /= | a /= b; | a = a / b; |
| %= | c %= 3; | c = c % 3; |

Example 1

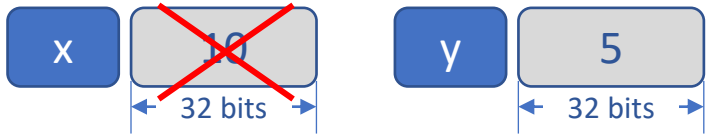
```
int x = 10;
int y = 5;
x*= y;                    //x = x * y
```

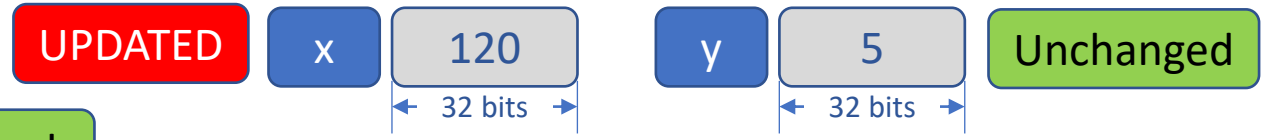




Example 2

```
int x = 10;
int y = 5;
x*= y + 7; //x = x * (y + 7)
```



# Increment/Decrement

- Two increment operators are used
  - Pre-increment ++x increment then act
  - Post-increment x++ act then increment
- Two decrement operators are used
  - Pre-decrement --x decrement then act
  - Post-decrement x-- act then decrement

# Increment/Decrement

**PRE-INCREMENT (INCREMENT THEN ACT)**

SYNTAX

```
++variable;
```

Example 1

```
counter = 0;        // counter is previously declared
++counter;          // counter = counter + 1
                    // counter should previously have a value
```

After this example, counter = 1

Example 2

```
x = 3;              // x is previously declared
y = ++x;    // y is previously declared
                    // x already has a value
```

- In Example 2, (y = ++x) is equivalent to the following statements in this order:

```
1   x = x + 1;          //increment
2   y = x;                      //act (assign)
```

After this example, x = 4 and y = 4

# Increment/Decrement

**PRE-INCREMENT (INCREMENT THEN ACT)**

Example 3

```
a = 7;                                  // a is previously declared
System.out.println (++a);       //increment then act (print)
```

- In Example 3, System.out.println (++a) is equivalent to the following two statements in this order:

```
1  a = a + 1;                  //increment
2  System.out.println (a);          //act (print)
```

The program output is 8

Example 4

```
a = 5;                  // a is previously declared
b = ++a % 2;            // b is previously declared
                        // a has already a value
```

- In Example 4, b = ++a % 2 is equivalent to the following two statements in this order:

```
1  a = a + 1;                  //increment
2  b = a % 2;                  //act (mod)
```

After this example, a = 6 and b = 0

# Increment/Decrement

**POST-INCREMENT (ACT THEN INCREMENT)**

SYNTAX

```
variable++;
```

Example 1

```
counter = 0;          // counter is previously declared
counter++;            // counter = counter + 1
                      // counter should previously have a value
```

After this example, counter = 1

Example 2

```
x = 3;                // x is previously declared
y = x++;   // y is previously declared
                      // x already has a value
```

- In Example 2, (y = x++) is equivalent to the following statements in this order:

```
1  y = x;                    //act (assign)
2  x = x + 1;          //increment
```

After this example, x = 4 and y = 3

# Increment/Decrement

**POST-INCREMENT (ACT THEN INCREMENT)**

Example 3

```
a = 7;                                  // a is previously declared
System.out.println (a++);       //act (print) then increment
```

- In Example 3, System.out.println (a++) is equivalent to the following two statements in this order:

```
1  System.out.println (a);          //act (print)
2  a = a + 1;
```

The program output is 7

Example 4

```
a = 5;                  // a is previously declared
b = a++ % 2;            // b is previously declared
                        // a already has a value
```

- In Example 4, b = a++ % 2 is equivalent to the following two statements in this order:

```
1  b = a % 2;           //act (mod)
2  a = a + 1;           //increment
```

After this example, a = 6 and b = 1

# Increment/Decrement

**NOTES**

- When a variable is used by itself, there is no difference between the post-increment and the pre-increment:
  - counter++;
  - ++counter;

- The following statements have all the same effect:
  - counter = counter + 1;
  - counter++;
  - ++counter;
  - counter +=1;

# Increment/Decrement

**NOTES**

- When a variable is used by itself, there is no difference between the post-decrement and the pre-decrement:
  - counter--;
  - --counter;

- The following statements have all the same effect:
  - counter = counter – 1;
  - counter--;
  - --counter;
  - counter -=1;

# Increment/Decrement

**PRE-DECREMENT (DECREMENT THEN ACT)**

- The same rules of the increment operator. However, it decrements rather than increments.

SYNTAX

```
--variable;
```

Example 1

```
x = 3;                  // x is previously declared
y = --x;    // y is previously declared
                        // x already has a value
```

- In Example 2, (y = --x) is equivalent to the following statements in this order:

```
1  x = x - 1;          //decrement
2  y = x;                       //act (assign)
```

After this example, x = 2 and y = 2

# Increment/Decrement

**POST-DECREMENT (ACT THEN DECREMENT)**

SYNTAX

```
variable--;
```

Example 1

```
x = 3;                  // x is previously declared
y = x--;    // y is previously declared
                        // x already has a value
```

- In Example 2, (y = x--) is equivalent to the following statements in this order:

```
1   y = x;                      //act (assign)
2   x = x -1;                   //decrement
```

After this example, x = 2 and y = 3

# ORDER OF PRECEDENCE

| | |
|---|---|
| Parenthesis ( ) | inside-out |
| Increment (++), Decrement (--) | from left to right |
| * / % | from left to right |
| + - | from left to right |
| < > <= >= | from left to right |
| == != | from left to right |
| && | from left to right |
| \|\| | from left to right |
| = += -= *= /= %= | from left to right |

# The Math class

- The Math.pow Method: The method takes two double arguments. It raises the first argument to the power of the second argument, and returns the result as a double.

```
result = Math.pow(4.0, 2.0);
```

$result = 4^2$

```
x = 3 * Math.pow(6.0, 3.0);

System.out.println(Math.pow(5.0, 4.0));
```

- The Math.sqrt Method: accepts a double value as its argument and returns the square root of the value.

```
result = Math.sqrt(9.0);
System.out.println(Math.sqrt(25.0));
```

# Conversion between primitive Data Types

- Before a value can be stored in a variable, the value's data type **must be compatible** with the variable's data type.
- Java performs some conversions between data types **automatically**, but does **not automatically perform any conversion** that can result in the **loss of data**.
- Java also **follows a set of rules** when evaluating arithmetic expressions containing **mixed data types**.

# Conversion between primitive Data Types

- Java is a strongly typed language. This means that before a value is assigned to a variable, Java checks the data types of the variable and the value being assigned to it to determine whether they are compatible. For example, look at the following statements:

```
int x;
double y = 2.5;
x = y;
```

- The assignment statement is attempting to **store a double value** (2.5) in an **int** variable. When the Java compiler encounters this line of code, **it will respond with an error message**.

# Conversion between primitive Data Types

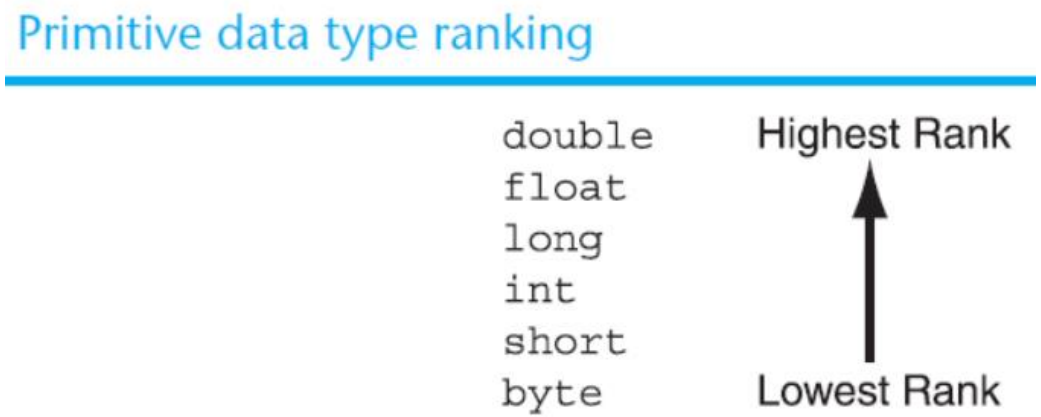
- Not all assignment statements that mix data types are rejected by the compiler, however.

```
int x;
short y = 2;
x = y;
```

- This assignment statement, which stores a short in an int, will work with **no problems**.
- So, why does Java permit a short to be stored in an int, but does not permit a double to be
- stored in an int? The obvious reason is that a **double can store fractional numbers** and can **hold values much larger than an int** can hold.
- If Java were to permit a double to be assigned to an int, a **loss of data would be likely**.

# Conversion between primitive Data Types

- The primitive data types are **ranked**. One data type outranks another if it can hold a larger number.
- For example, a float outranks an int, and an int outranks a short.
- In assignment statements where values of lower-ranked data types are stored in variables of higher-ranked data types, **Java automatically** converts the lower-ranked value to the higherranked type. This is called a ***widening conversion***.




```
double x;
int y = 10;
x = y;          // Performs a widening conversion
```

- **A *narrowing conversion*** is the conversion of a value to a lower-ranked type. For example, converting a double to an int. Narrowing conversions can potentially cause a loss of data, so Java **does not automatically perform them**.

# Cast operators

- The **cast operator** lets you manually convert a value, even if it means that a **narrowing conversion** will take place.
- Cast operators are **unary operators** that appear as a **data type name enclosed in a set of parentheses**.
- The operator precedes the value being converted.
- Here is an example: `x = (int)number;`
- The cast operator in this statement is the **word int inside the parentheses**.
- It returns the value in number, **converted to an int**. This converted value is then stored in x.
- If number were a floating-point variable, such as a float or a double, the value that is returned would be truncated, which means the fractional part of the number is lost.
- **The original value in the number variable is not changed, however.**

# Cast operators

**Examples of casting:**

| Statement | Description |
|---|---|
| `littleNum = (short)bigNum;` | The cast operator returns the value in `bigNum`, converted to a `short`. The converted value is assigned to the variable `littleNum`. |
| `x = (long)3.7;` | The cast operator is applied to the expression 3.7. The operator returns the value 3, which is assigned to the variable `x`. |
| `number = (int)72.567;` | The cast operator is applied to the expression 72.567. The operator returns 72, which is used to initialize the variable `number`. |
| `value = (float)x;` | The cast operator returns the value in `x`, converted to a `float`. The converted value is assigned to the variable `value`. |
| `value = (byte)number;` | The cast operator returns the value in `number`, converted to a `byte`. The converted value is assigned to the variable `value`. |

# Cast operators

## Examples of casting:

SYNTAX

(dataTypeName) expression

Examples

```
(int) (7.9);                                  // = 7
(double) (25);                                // = 25.0
(double) (5 + 3);                             // = (double) (8) = 8.0
(double) (15) / 2;                 // = 15.0 / 2   = 7.5
(double) (15 / 2);                 // = (double) (7)   = 7.0
(int) (7.8 + (double) (15) / 2);
                                   //=(int)(7.8+15.0/2)=(int)(7.8+7.5) = 15
(int) (7.8 + (double) (15 / 2)); //(int)(7.8+7.0) = 14
```

# Cast operators

## Examples of casting:

Examples

```
char ch = ‘a’;                    //Unicode of ‘a’ = 97
int unicode;
unicode = (int)(ch); //unicode = 97
System.out.println (unicode);
```

```
97
```

Examples

```
int x = 98;
char ch;
ch = (char)(x);
System.out.println (ch);
```

```
b
```

# Cast operators

- Recall that when **both operands of a division are integers**, the operation will result in integer division. This means that the result of the division will be an integer, with **any fractional part of the result thrown away**. For example, look at the following code:

```
int pies = 10, people = 4;
double piesPerPerson;
piesPerPerson = pies / people;
```

Although 10 divided by 4 is 2.5, this code will store 2 in the `piesPerPerson` variable. Because both `pies` and `people` are `int` variables, the result will be an `int`, and the fractional part will be thrown away. We can modify the code with a cast operator, however, so it gives the correct result as a floating-point value:

```
piesPerPerson = (double)pies / people;
```

The variable `pies` is an `int` and holds the value 10. The expression `(double)pies` returns the value in `pies` converted to a `double`. This means that one of the division operator's operands is a `double`, so the result of the division will be a `double`. The statement could also have been written as follows:

```
piesPerPerson = pies / (double)people;
```

# Cast operators

Example 1

```
int x = 15;
int y = 2;
int z = x / y;   // z = 7 (the decimal part is truncated)
```

Example 2

```
int x = 15;
int y = 2;
double  z = x / y;   // z = 7.0 (since x & y are integers)
```

Example 3

```
int x = 15;
double y = 2.0;
double  z = x / y;   // z = 7.5 (since y is double)
```

Example 4

```
double x = 15.0;
double y = 2.0;
double  z = x / y;   // z = 7.5 (since x & y are double)
```

# Cast operators

**WARNING!** The cast operator can be applied to an entire expression enclosed in parentheses. For example, look at the following statement:

```
piesPerPerson = (double)(pies / people);
```

This statement does not convert the value in `pies` or `people` to a `double`, but converts the result of the expression `pies / people`. If this statement were used, an integer division operation would still have been performed. Here's why: The result of the expression `pies / people` is 2 (because integer division takes place). The value 2 converted to a `double` is 2.0. To prevent the integer division from taking place, one of the operands must be converted to a `double`.

# Cast operators: Mixed integer operations

- When values of the byte or short data types are used in arithmetic expressions, they are **temporarily converted to int values**.
- The result of an arithmetic operation using only a mixture of byte, short, or int values **will always be an int**.
- For example, assume that b and c in the following expression are short variables: **b + c** Although both b and c are short variables, **the result of the expression b + c is an int.** This means that when the result of such an expression is stored in a variable, the variable must be an int or **higher data type**.

```
short firstNumber = 10,
      secondNumber = 20,
      thirdNumber;
// The following statement causes an error!
thirdNumber = firstNumber + secondNumber;
```

- The error results from the fact that **thirdNumber is a short**. Although firstNumber and secondNumber are also short variables, **the expression firstNumber + secondNumber results in an int value**. The program can be corrected if thirdNumber is declared as an int, or if a cast operator is used in the assignment statement:

```
thirdNumber = (short)(firstNumber + secondNumber);
```

# Cast operators: other Mixed Mathematical expressions

- If a mathematical expression has one or more values of the double, float, or long data types, Java converts all the operands in the expression to **the same data type**. The rules are as follows:

1. If one of an operator's operands is a `double`, the value of the other operand will be converted to a `double`. The result of the expression will be a `double`. For example, in the following statement assume that `b` is a `double` and `c` is an `int`:

```
a = b + c;
```

   The value in `c` will be converted to a `double` prior to the addition. The result of the addition will be a `double`, so the variable `a` must also be a `double`.

2. If one of an operator's operands is a `float`, the value of the other operand will be converted to a `float`. The result of the expression will be a `float`. For example, in the following statement assume that `x` is a `short` and `y` is a `float`:

```
z = x * y;
```

   The value in `x` will be converted to a `float` prior to the multiplication. The result of the multiplication will be a `float`, so the variable `z` must also be either a `double` or a `float`.

3. If one of an operator's operands is a `long`, the value of the other operand will be converted to a `long`. The result of the expression will be a `long`. For example, in the following statement assume that `a` is a `long` and `b` is a `short`:

```
c = a - b;
```

   The variable `b` will be converted to a `long` prior to the subtraction. The result of the subtraction will be a `long`, so the variable `c` must also be a `long`, `float`, or `double`.

# Checkpoint

2.25 The following declaration appears in a program:

```
short totalPay, basePay = 500, bonus = 1000;
```

The following statement appears in the same program:

```
totalPay = basePay + bonus;
```

a) Will the statement compile properly or cause an error?
b) If the statement causes an error, why? How can you fix it?

2.26 The variable `a` is a `float` and the variable `b` is a `double`. Write a statement that will assign the value of `b` to `a` without causing an error when the program is compiled.

# creating named constants with final

- The final key word can be used in a variable declaration to make the variable a **named constant**. Named constants are **initialized** with a value, and that value **cannot change** during the execution of the program.
- In `amount = balance * 0.069;` What does 0.069 means for a reader of the code?
- Better method is as follows:

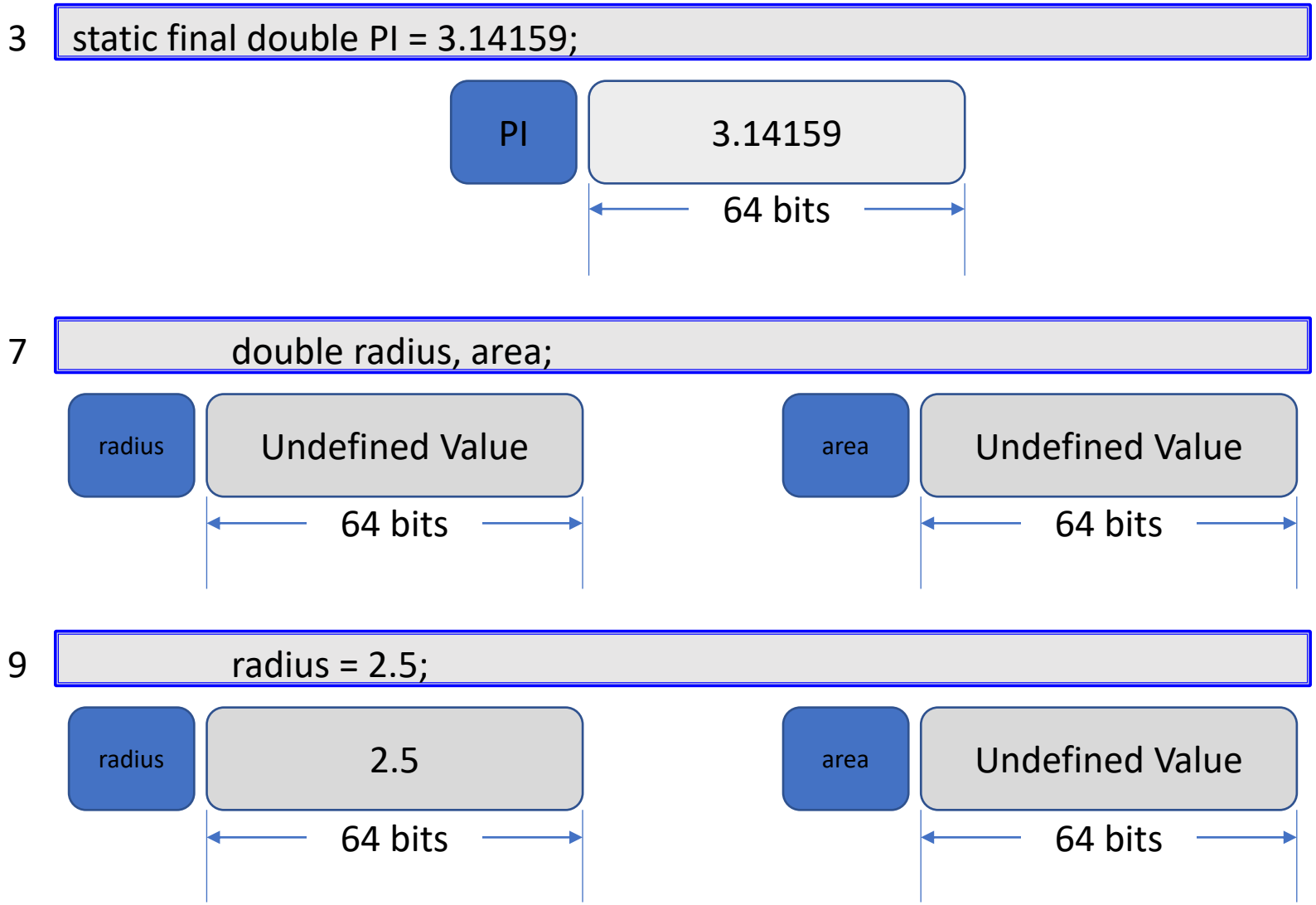
```
final double INTEREST_RATE = 0.069;
amount = balance * INTEREST_RATE;
```

- **A new programmer can read the second statement and know what is happening.**
- **The Math.PI named constant (***3.14159265358979323846***)**

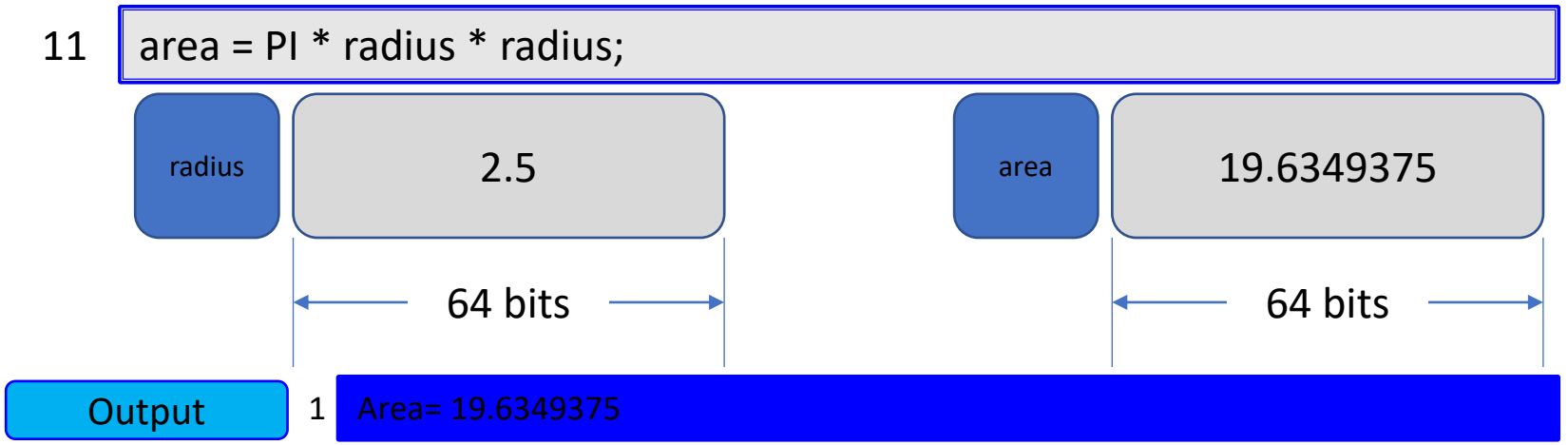
# creating named constants with final

```
1  public class AssignmentOperator
2  {
3    static final double PI = 3.14159;//constant declaration
4    public static void main (String[] args)
5      {
6        // Declaration section: to declare needed variables
7              double radius, area;
8        // Input section: to enter values of used variables
9              radius = 2.5;
10       // Processing section: processing statements
11             area = PI * radius * radius;
12       // Output section: display program output
13             System.out.println (“Area= “ + area);
14     } // end main
15 } // end class
```

# creating named constants with final



# creating named constants with final



# The String class

- The String class allows you to create objects for holding strings. It also has various methods that allow you to work with strings.
- A string is a sequence of characters. It can be used to **represent any type of data that contains text**, such as names, addresses, warning messages, and so forth.
- **String literals** are enclosed in **double quotation marks**, such as the following:

```
"Hello World"
"Joe Mahoney"
```

- Java does **not have a primitive data type** for storing them in memory. Instead, the Java API provides **a class** for handling strings. You use this class to **create objects** that are capable of **storing** strings and **performing operations** on them.

```
String name;
```

**TIP:** The `S` in `String` is written in an uppercase letter. By convention, the first character of a class name is always written in an uppercase letter.

# The String class

```
1  // A simple program demonstrating String objects.
2
3  public class StringDemo
4  {
5     public static void main(String[] args)
6     {
7        String greeting = "Good morning, ";
8        String name = "Herman";
9
10       System.out.println(greeting + name);
11    }
12 }
```

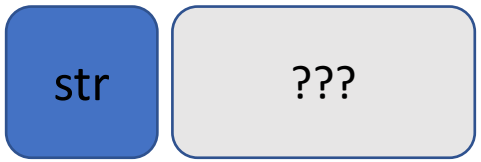
**Program Output**

```
Good morning, Herman
```

# The String class

- The following statement declares a variable str of type String:

```
1  String str;
```



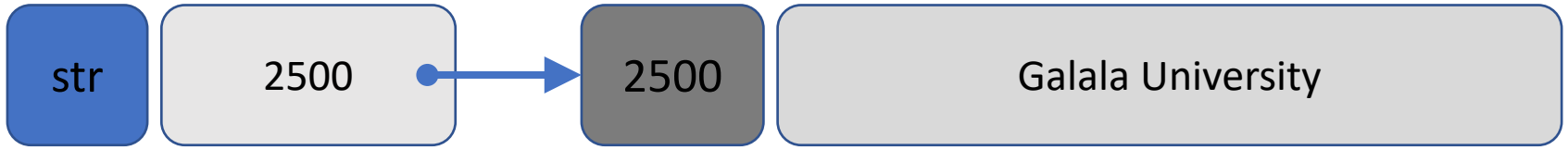

- In Java, String is NOT a primitive data type.
- A variable declared as a String class is known as a reference variable.
- A reference variable stores an address rather than a value.
- This is more illustrated in the next slide.

# The String class: String initialization

- The following statement initializes a variable str of type String:

```
1  String str = “Galala University”;
```

- After this statement, the memory layout is as follows:



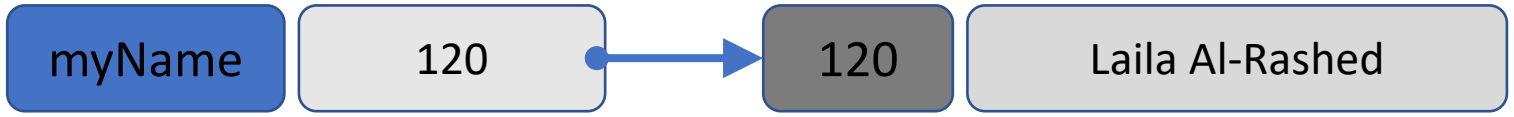

- In the example above, 2500 is the address that stores “Galala University”.

- We say that str “points to” the memory location (address) that contains the string “Galala University”.
- The value of the memory location (address) - 2500 in this example - is specified by the operating system.
- Whenever we want to refer to “Galala University”, we just use the variable str.
- Since variables declared as String “refer to” a memory location, they are known as reference variables.
- The following statement also initializes a variable country of type String:

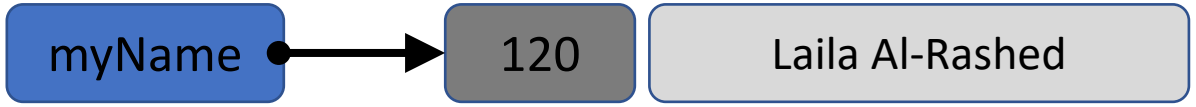
```
country = new String (“Galala University”);
```

# The String class: String update

- Let us consider another example:



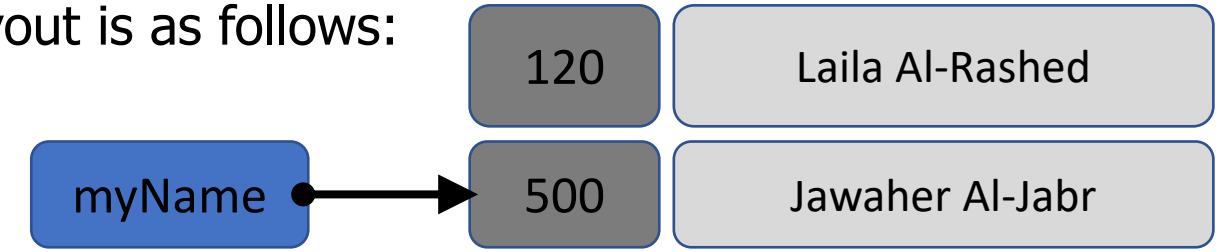

- For simplicity, we'll write it as follows:




- Also, consider this statement:

```
1  myName = "Jawaher Al-Jabr";  //changes the address stored in myName
```

- The memory layout is as follows:




- The old address (120) is no more accessed by the variable myName.
- The variable myName refers to the new address (500) that contains the new value "Jawaher Al-Jabr".

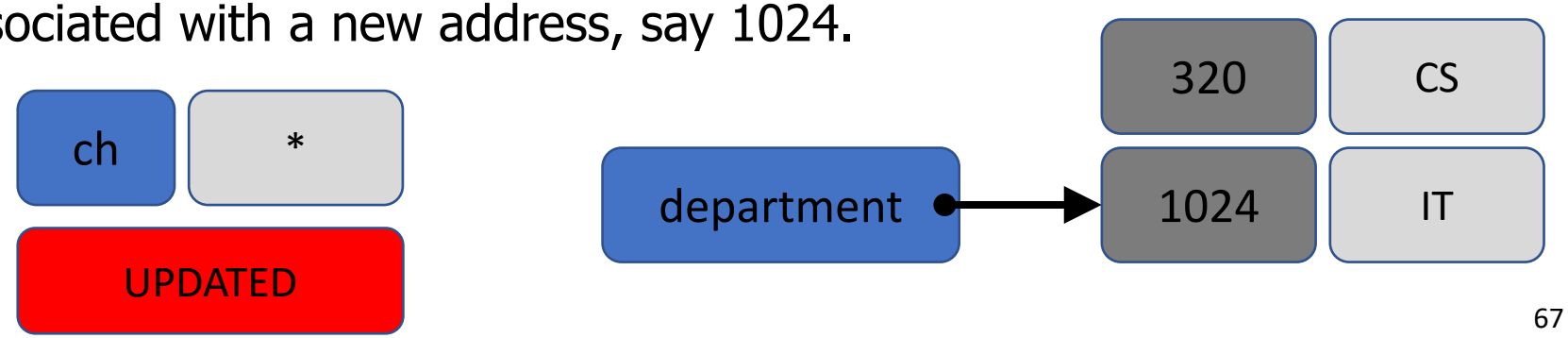
# The String class: String update

- Consider the following statements:

```
1  char ch = 'x';                                  //This is a data variable
2  String department = new String("CS");  //This is a reference variable
3  department = "IT"; //changes the address stored in department
4  ch = '*';                  //changes the value (not address) stored in ch
```

- After the execution of lines 1 & 2, ch is initialized to 'x'. In addition, department is associated with a new address, say 320.


- After the execution of lines 3 & 4, the value of ch is overwritten; department is no more associated with the address 320. It is rather associated with a new address, say 1024.



# The String class

- String class provides numerous methods for working with strings.

```
stringSize = name.length();
```

This statement calls the `length` method of the object that `name` refers to. To *call* a method means to execute it. The general form of a method call is as follows:

```
referenceVariable.method(arguments. . .)
```

*`referenceVariable`* is the name of a variable that references an object, *`method`* is the name of a method, and *`arguments. . .`* is zero or more arguments that are passed to the method. If no arguments are passed to the method, as is the case with the `length` method, a set of empty parentheses must follow the name of the method.

1. https://www.javatpoint.com/java-string
2. https://www.w3schools.com/java/java_ref_string.asp

# The String class

```
1  // This program demonstrates the String class's length method.
2
3  public class StringLength
4  {
5     public static void main(String[] args)
6     {
7        String name = "Herman";
8        int stringSize;
9
10       stringSize = name.length();
11       System.out.println(name + " has " + stringSize +
12                          " characters.");
13    }
14 }
```

**Program Output**

```
Herman has 6 characters.
```

**NOTE:** The `String` class's `length` method returns the number of characters in the string, including spaces.

# The String class: converting string input to numbers

| Method | Use This Method To . . . | Example Code |
|---|---|---|
| `Byte.parseByte` | Convert a string to a byte. | `byte num;`<br>`num = Byte.parseByte(str);` |
| `Double.parseDouble` | Convert a string to a double. | `double num;`<br>`num = Double.parseDouble(str);` |
| `Float.parseFloat` | Convert a string to a float. | `float num;`<br>`num = Float.parseFloat(str);` |
| `Integer.parseInt` | Convert a string to an int. | `int num;`<br>`num = Integer.parseInt(str);` |
| `Long.parseLong` | Convert a string to a long. | `long num;`<br>`num = Long.parseLong(str);` |
| `Short.parseShort` | Convert a string to a short. | `short num;`<br>`num = Short.parseShort(str);` |

# The String class: Examples

| Method | Description and Example |
|---|---|
| `charAt(index)` | The argument *index* is an `int` value and specifies a character position in the string. The first character is at position 0, the second character is at position 1, and so forth. The method returns the character at the specified position. The return value is of the type `char`.<br>**Example:**<br>`char letter;`<br>`String name = "Herman";`<br>`letter = name.charAt(3);`<br>After this code executes, the variable `letter` will hold the character 'm'. |
| `length()` | This method returns the number of characters in the string. The return value is of the type `int`.<br>**Example:**<br>`int stringSize;`<br>`String name = "Herman";`<br>`stringSize = name.length();`<br>After this code executes, the `stringSize` variable will hold the value 6. |

# The String class: Examples

`toLowerCase()` This method returns a new string that is the lowercase equivalent of the string contained in the calling object.

**Example:**

```
String bigName = "HERMAN";
String littleName = bigName.toLowerCase();
```

After this code executes, the object referenced by `littleName` will hold the string "herman".

`toUpperCase()` This method returns a new string that is the uppercase equivalent of the string contained in the calling object.

**Example:**

```
String littleName = "herman";
String bigName = littleName.toUpperCase();
```

After this code executes, the object referenced by `bigName` will hold the string "HERMAN".

# The String class: Examples

| String | "How are you?" | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Character in the String | 'H' | 'o' | 'w' | ' ' | 'a' | 'r' | 'e' | ' ' | 'y' | 'o' | 'u' | '?' |
| Position of the character | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

- Note that the space character ' ' has a position number.
- Also, special characters '?' have a position number.
- Maximum position = 11. Number of characters = 12.

| String | "I am fine." | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Character in the String | 'I' | ' ' | 'a' | 'm' | ' ' | 'f' | 'i' | 'n' | 'e' | '.' |
| Position of the character | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Maximum position = 9. Number of characters = 10.

Number of characters (length) = Maximum position + 1

# The String class: String concatenation

**THE '+' OPERATOR**

- String concatenation appends the second string to the first.
- The + operator is used to concatenate two strings.
- Examples:
  - "Programming with " + "Java I" = "Programming with Java I"
  - "My name is " + "Sara" = "My name is Sara"

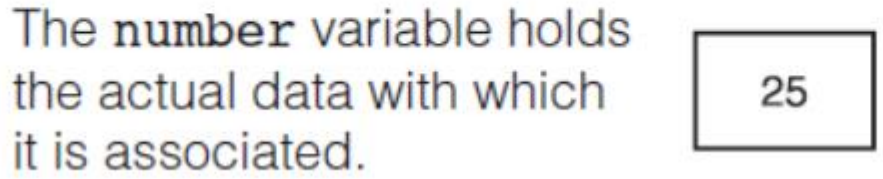
- When a String is concatenated with a numeric value, the latter is converted into a String.
- Examples:
  - "Price is SR" + 28 = "Price is SR28"
  - "Pay rate is SR " + 30.5 = "Pay rate is SR 30.5"
  - "The sum is " + 12 + 24 = "The sum is 1224"
  - "The sum is " + (12 + 24) = The sum is 26

# Primitive Type Variables and class Type Variables

- Primitive type variables hold **the actual data items** with which they are associated.
- For example, assume that *number* is an int variable. The following statement stores the value 25 in the variable:

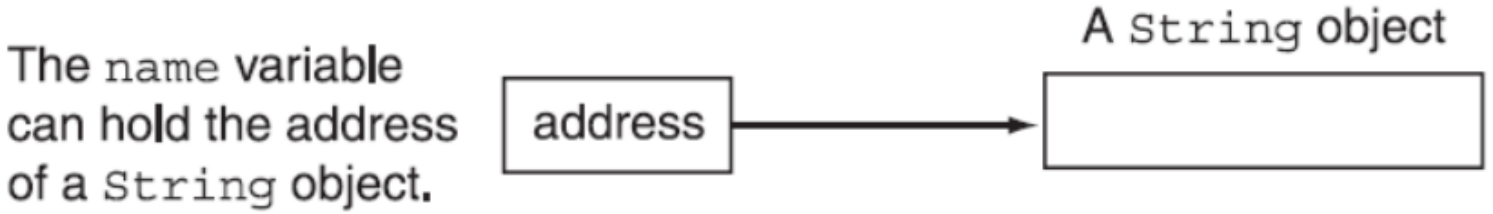
```
number = 25;
```

A primitive type variable holds the data with which it is associated




- A **class type variable** does not hold the actual data item that it is associated with, but holds the **memory address** of the data item it is associated with.
- If *name* is a String class variable, then *name* can hold the memory address of a String object.

A `String` class variable can hold the address of a `String` object



# Scope of a variable

- A variable's scope is the **part of the program** that has **access to the variable**.
- Every variable has a **scope**. A variable is visible only to statements inside the variable's scope.
- So far, you have only seen variables declared inside the **main** method. Variables that are declared inside a method are called **local variables**.

```
// This program can't find its variable.

public class Scope
{
   public static void main(String[] args)
   {
      System.out.println(value); // ERROR!
      int value = 100;
   }
}
```

The compiler reads the program from **top to bottom**. If it encounters a statement that uses a variable **before** the variable is declared, an **error** will result. To correct the program, the variable declaration must be written **before** any statement that uses it.

# Scope of a variable

- You **cannot** have two local variables with the same name in the same scope:

```
public static void main(String[] args)
{
   // Declare a variable named number and
   // display its value.
   int number = 7;
   System.out.println(number);
   // Declare another variable named number and
   // display its value.
   int number = 100;                  // ERROR!!!
   System.out.println(number);        // ERROR!!!
}
```

# Comments

- Comments are **notes** of explanation that document lines or sections of a program. Comments are **part of the program**, but the **compiler ignores them**. They are intended for **people** who may be reading the source code.

**Single-Line comments**

**and Multi-Line comments**

```
/*
   PROGRAM: Comment2.java
   Written by Herbert Dorfmann
   This program calculates company payroll
*/

public class Comment2
{
   public static void main(String[] args)
   {
      double payRate;        // Holds the hourly pay rate
      double hours;          // Holds the hours worked
      int employeeNumber;    // Holds the employee number

      // The Remainder of This Program is Omitted.
   }
}
```

# Programming style

- Programming style refers to the way a programmer uses spaces, indentations, blank lines, and punctuation characters to visually arrange a program's source code..

```
1  public class Compact {public static void main(String [] args){int
2  shares=220; double averagePrice=14.67; System.out.println(
3  "There were "+shares+" shares sold at $"+averagePrice+
4  " per share.");}}
```

**Program Output**

```
There were 220 shares sold at $14.67 per share.
```

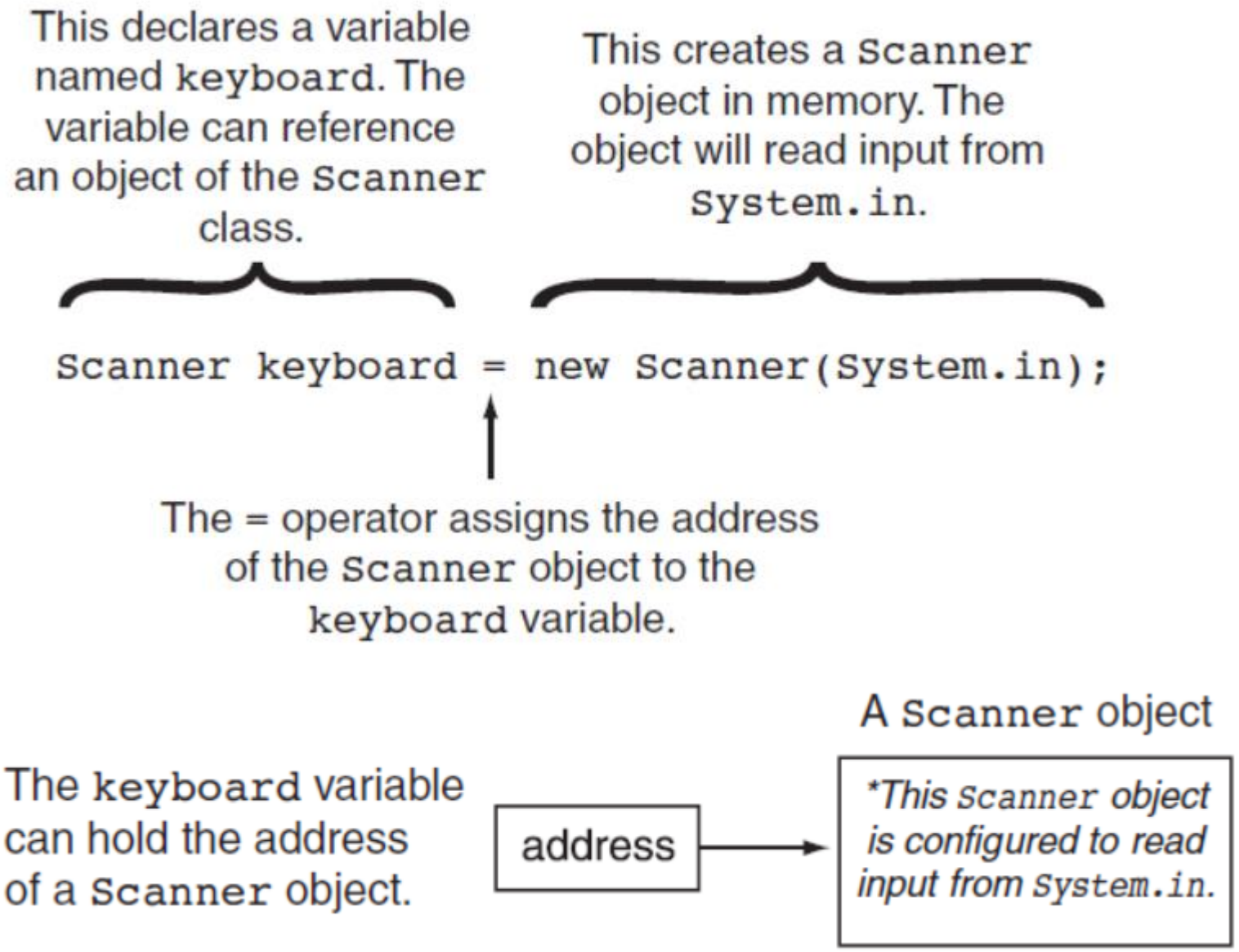
# Programming style

```
/**
   This example is much more readable than Compact.java.
*/

public class Readable
{
   public static void main(String[] args)
   {
      int shares = 220;
      double averagePrice = 14.67;

      System.out.println("There were " + shares +
                         " shares sold at $" +
                         averagePrice + " per share.");
   }
}
```

**Program Output**

```
There were 220 shares sold at $14.67 per share.
```

# A note about the Scanner class



# A note about the Scanner class

| Method | Example and Description |
|---|---|
| nextByte | Example Usage:<br>`byte x;`<br>`Scanner keyboard = new Scanner(System.in);`<br>`System.out.print("Enter a byte value: ");`<br>`x = keyboard.nextByte();`<br>Description: Returns input as a byte. |
| nextDouble | Example Usage:<br>`double number;`<br>`Scanner keyboard = new Scanner(System.in);`<br>`System.out.print("Enter a double value: ");`<br>`number = keyboard.nextDouble();`<br>Description: Returns input as a double. |

# A note about the Scanner class

`nextFloat`

Example Usage:

```
float number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a float value: ");
number = keyboard.nextFloat();
```

Description: Returns input as a `float`.

`nextInt`

Example Usage:

```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer value: ");
number = keyboard.nextInt();
```

Description: Returns input as an `int`.

`nextLine`

Example Usage:

```
String name;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your name: ");
name = keyboard.nextLine();
```

Description: Returns input as a `String`.

# A note about the Scanner class

- nextLong

Example Usage:

```
long number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a long value: ");
number = keyboard.nextLong();
```

Description: Returns input as a long.

nextShort

Example Usage:

```
short number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a short value: ");
number = keyboard.nextShort();
```

Description: Returns input as a short.

# A note about the Scanner class: reading a character

The Scanner class does not have a method for reading a single character.

```
String input;  // To hold a line of input
char answer;   // To hold a single character

// Create a Scanner object for keyboard input.
Scanner keyboard = new Scanner(System.in);

// Ask the user a question.
System.out.print("Are you having fun? (Y=yes, N=no) ");
input = keyboard.nextLine();  // Get a line of input.
answer = input.charAt(0);     // Get the first character.
```

# A note about the Scanner class: Mixing calls to nextLine with calls to other Scanner Methods

When you call one of the Scanner class's methods to (1) read a primitive value, such as nextInt or nextDouble, and then (2) call the nextLine method to read a string, an annoying and hard to-find problem can occur.

# A note about the Scanner class: Mixing calls to nextLine with calls to other Scanner Methods

```
7  public class InputProblem
8  {
9     public static void main(String[] args)
10    {
11       String name;    // To hold the user's name
12       int age;        // To hold the user's age
13       double income;  // To hold the user's income
14
15       // Create a Scanner object to read input.
16       Scanner keyboard = new Scanner(System.in);
17
18       // Get the user's age.
19       System.out.print("What is your age? ");
20       age = keyboard.nextInt();
21
22       // Get the user's income
23       System.out.print("What is your annual income? ");
24       income = keyboard.nextDouble();
25
26       // Get the user's name.
27       System.out.print("What is your name? ");
28       name = keyboard.nextLine();
29
30       // Display the information back to the user.
31       System.out.println("Hello, " + name + ". Your age is " +
32                          age + " and your income is $" +
33                          income);
34    }
35 }
```

**Program Output with Example Input Shown in Bold**

```
What is your age? 24 [Enter]
What is your annual income? 50000.00 [Enter]
What is your name? Hello, . Your age is 24 and your income is $50000.0
```

It appears that the statement in line 28 is **skipped**. The name is **never read** from the keyboard. This happens because of **a slight difference in behavior** between the nextLine method and the other Scanner class methods.

## The keyboard buffer concept:

- Pressing the e key causes a newline character to be stored in the keyboard buffer
- next**X** methods read the value 24 from the keyboard buffer, and then stopped when it encountered the **newline character. And** The newline character remained in the keyboard buffer.
- nextLine is not designed to skip over an initial newline character.
- What is the solution?

# A note about the Scanner class: Mixing calls to nextLine with calls to other Scanner Methods

```
7  public class CorrectedInputProblem
8  {
9     public static void main(String[] args)
10    {
11       String name;   // To hold the user's name
12       int age;       // To hold the user's age
13       double income; // To hold the user's income
14
15       // Create a Scanner object to read input.
16       Scanner keyboard = new Scanner(System.in);
17
18       // Get the user's age.
19       System.out.print("What is your age? ");
20       age = keyboard.nextInt();
21
22       // Get the user's income
23       System.out.print("What is your annual income? ");
24       income = keyboard.nextDouble();
25
26       // Consume the remaining newline.
27       keyboard.nextLine();
28
29       // Get the user's name.
30       System.out.print("What is your name? ");
31       name = keyboard.nextLine();
32
33       // Display the information back to the user.
34       System.out.println("Hello, " + name + ". Your age is " +
35                          age + " and your income is $" +
36                          income);
37    }
38 }
```

**Program Output with Example Input Shown in Bold**

```
What is your age? 24 [Enter]
What is your annual income? 50000.00 [Enter]
What is your name? Mary Simpson [Enter]
Hello, Mary Simpson. Your age is 24 and your income is $50000.0
```

Thank you