

303.1.2

Java Programming Fundamentals

Learning Objectives

2

In this section, we will discuss how to create, compile, and run a Java program, and how to solve practical problems programmatically. Through the example problems, we will learn Java primitive data types and related subjects, such as variables, constants, operators, expressions, type casting, and input and output.

- ❑ By the end of this presentation, learners will be able to:
 - Explain and demonstrate variable and data type.
 - Explain and demonstrate Java Literal.
 - Describe Java Scanner Class.
 - Explain Java Literal.
 - Demonstrate Operators.
 - Understand and demonstrate Type Casting.
- ❑ There are practice assignments, labs, and practice presentations available, and learners can use office hours to complete the labs and assignments. If you have any technical problems, please contact your instructor.

Outline

3

- ❖ Lesson 1: Overview of Java Variable and Data Type.
- ❖ Lesson 2: Introduction to Java Literal.
- ❖ Lesson 3: Introduction to Java Scanner Class.
- ❖ Lesson 4: Overview of Operators
- ❖ Lesson 5: Introduction to Type Casting.

Overview of Java Variable and Data Type

- Variable Declaration/Memory Allocation.
- Declare Names of Variables (Identifiers).
- Final Variable (Named Constants) in Java.
- Java is Statically Typed.
- Java Data Types.
- Primitive Data Types.
- Non-Primitive Data Types



Overview of Java Variable and Data Types

5

- ❑ As the name suggests, the word “variable” can be broken down into two words: “**vary**” + “**able**,” which means that the value of variables can change.
- ❑ A Java variable is a **memory location** that can contain a **Value**. A variable, thus has a **data type**. Data types are broken down into two groups:
 - Primitive data types.
 - Object references/reference variables.
- ❑ A variable takes up a certain amount of space in memory and how much memory a variable takes depending on its data type.

Overview of Java Variable and Data Types (continued)

6

Let's understand *variable* by using an example:

- Problem: Computing the Area of a circle.
 - Let's see the example on the next slide, which is how to compute the area of the circle.

Variable Declaration/Memory Allocation

7

```
public class AreaComputer {  
    /** Main method */  
    public static void main(String[] args) {  
        double radius;  
        double area;  
  
        // Assign a radius  
        radius = 20;  
  
        // Compute area  
        area = radius * radius * 3.14159;  
  
        // Display results  
        System.out.println("The area for the circle of  
radius " +  
            radius + " is " + area);  
    }  
}
```

Allocate memory
for radius.

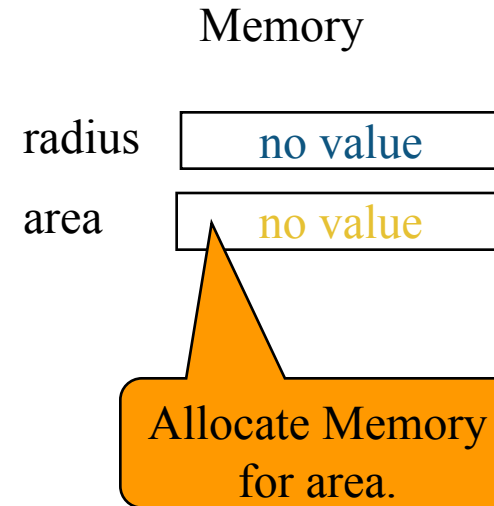
radius

no value

Variable Declaration/Memory Allocation (continued)

8

```
public class AreaComputer {  
    /** Main method */  
    public static void main(String[] args) {  
        double radius;  
        double area;  
  
        // Assign a radius  
        radius = 20;  
  
        // Compute area  
        area = radius * radius * 3.14159;  
  
        // Display results  
        System.out.println("The area for the circle of radius " +  
            radius + " is " + area);  
    }  
}
```



Assignment Statement

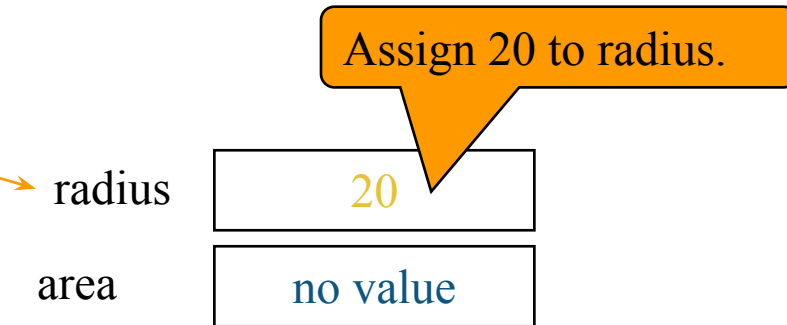
9

```
public class AreaComputer {  
    /** Main method */  
    public static void main(String[] args) {  
        double radius;  
        double area;  
  
        // Assign a radius  
        radius = 20;  
        // Compute area  
        area = radius * radius * 3.14159;  
  
        // Display results  
        System.out.println("The area for the circle of radius  
        " +  
        radius + " is " + area);  
    }  
}
```

In Java, the **only** way to get data into a variable -- that is, into the box that the variable names -- is with an **Assignment Statement**. An assignment statement takes this form:

variable = expression;

A where **expression** represents anything that computes a data value. When the computer sees an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. Consider the simple assignment statement:

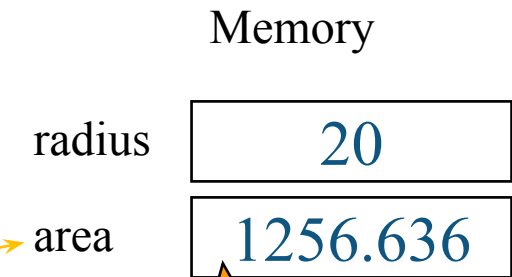


The **variable** in this assignment statement is **radius**, and the **expression** is the number **20**.

Computation and Assignment Statement

10

```
public class AreaComputer {  
    /** Main method */  
    public static void main(String[] args) {  
        double radius;  
        double area;  
  
        // Assign a radius  
        radius = 20;  
  
        // Compute area  
        area = radius * radius * 3.14159;  
  
        // Display results  
        System.out.println("The area for the circle of radius " +  
            radius + " is " + area);  
    }  
}
```



Compute area and assign
it to variable area.

Output Statement of Variable

11

```
public class AreaComputer {  
    /** Main method */  
    public static void main(String[] args) {  
        double radius;  
        double area;  
  
        // Assign a radius  
        radius = 20;  
  
        // Compute area  
        area = radius * radius * 3.14159;  
  
        // Display results  
        System.out.println("The area for the circle of radius " +  
            radius + " is " + area);  
    }  
}
```

Memory

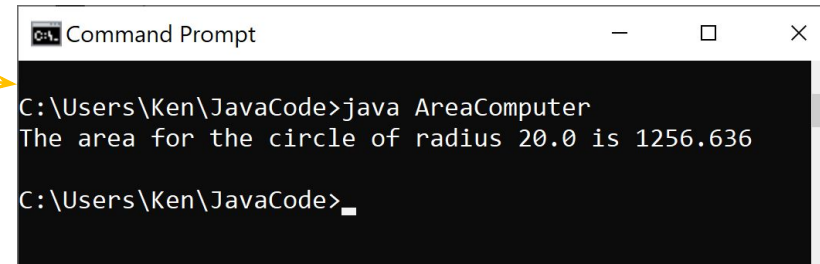
radius

20

area

1256.636

Print a message to the console.



```
Command Prompt  
C:\Users\Ken\JavaCode>java AreaComputer  
The area for the circle of radius 20.0 is 1256.636  
C:\Users\Ken\JavaCode>
```

Declare Names of Variables (Identifiers)

12

- ❖ An identifier is needed to name a variable, and an identifier is a sequence of characters that consist of letters, digits, underscores (`_`), and dollar signs (`$`).
- ❖ Java imposes the following rules on identifiers:
 - An identifier must start with a letter, underscore (`_`), or dollar sign (`$`).
 - Identifiers cannot start with a digit.
 - An identifier cannot be a reserved word.
 - An identifier cannot be the words, `true`, `false`, or `null`.
 - An identifier can be any number of characters in length.
 - Identifiers are case-sensitive. "A rose is NOT a Rose, and is NOT a ROSE."

Final Variable (Named Constants) in Java

13

- ❑ Constants are *non-modifiable (immutable)* variables, declared with a keyword `final`.
- ❑ Once we declare a **variable** with the **final** keyword, we cannot change the variable's **value**. If we attempt to change the **value** (**final** keyword), we will get a **compilation error**.
- ❑ It is used to define constants:

➤ The syntax is: **final datatype CONSTANTNAME = <expression>;**

```
final double PI = 3.14159;  
final int SIZE = 32;  
final double RADIUS = SIZE*SIZE*PI;
```

- ❑ It is a common practice to use **ALL_CAPS** for named constants.

For more information, see: [Use constant types for safer and cleaner code](#).

Java is Statically Typed

14

Java is a **Statically Typed** language. Every variable used in a Java program has a distinct data type.

- ❑ Variables cannot hold values inconsistent with their data type.
 - ❑ A variable declared as **int** cannot be assigned a floating-point value.
 - ❑ A variable declared as **Boolean** cannot hold a character value.
 - ❑ A variable declared as **Scanner** cannot be assigned a numeric value.

In contrast, many languages, such as Python and Javascript are **dynamically typed**, which means that any variable can take on any value. These languages are generally interpreted, not compiled.

In exchange for the restrictions imposed by static typing, a compiler can make programs run more efficiently and with fewer runtime errors.

□ Java has data types:

- **Primitive data types** - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`.
- **Non-primitive data type** are also called **Object references**, **Reference variables**, and **Reference data types** (e.g., `String`, `Arrays` and `Classes`).

Primitive Data Types

16

- ❑ A primitive data type variable contains the value of the variable directly in the memory allocated to the variable.
- ❑ Data uses a small fixed amount of memory.
- ❑ **Primitive data types** are built into the Java language. Java has eight built-in primitive data types:

boolean	byte
char	double
float	int
long	short

- ❑ Numeric integer types (4): **byte, short, int, and long.**
- ❑ Floating-point numeric types (2): **float and double.**
- ❑ Non-numeric (2): **boolean and char.**

Primitive Data Types - Integer Data Types

17

- ❑ The simplest numeric types hold a number with no fractional part: *integers*.
- ❑ When we store an integer, we can opt to use one, two, four, or eight bytes.
 - The smaller the amount of storage, the smaller the numerical range we can use.
- ❑ The standard Java integer data types are:

Data Type	Default Value	Storage Size	Value Range	Example
byte	0	1 byte (8 bits)	-128 to 127	<code>byte b = 10;</code>
short	0	2 bytes (16 bits)	-32,768 to 32,767	<code>short s = 10;</code>
int	0	4 bytes (32 bits)	-2,147,483,648 to 2,147,483,647	<code>int i = 10;</code>
long	0L	8 bytes (64 bits)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	<code>long l = 12345678910L;</code>

Primitive Data Types - Floating-Point Data Types

18

- Often, we need to represent numbers with a fractional part.
- The term “floating-point” means that a number’s decimal point can occur anywhere within the digits that represent the number. This allows a computer to represent a wide range of values, from the very tiny to the very large.
- Java provides two built-in, floating-point types:

Data Type	Default Value	Storage Size	Example
<code>float</code>	<code>0.0f</code>	4 bytes (32 bit floating point value)	<code>float f = 1.001f;</code>
<code>double</code>	<code>0.0</code>	8 bytes (64 bit floating point value)	<code>double d = 1.001;</code>

- If we need to perform exact financial calculations, we need to use more advanced data types, such as the [BigDecimal](#) class. These are not primitive data types but classes that implement more sophisticated ways of representing and storing numbers.

Additional Primitive Data Types

19

Java has two additional Primitive data types; both of which are non-numeric: boolean and char.

boolean type

- Represents one bit of information.
- There are only two possible values: **true** and **false**.
- Is used for simple flags that track true/false conditions.
 - Default value is false.
 - Example: **boolean flag = true;**

[For more information about Boolean Data Types, see our Wiki document.](#)

char type

- Is a single 16-bit unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535).
- Is used to store any character.
 - Example: `char letterA = 'A';`
- Can often be used in integer arithmetic expressions. This is valid Java:
 - `char letterB = letterA + 1;`

[Click here for Unicode Character Table.](#)

Example - Primitive Data type

20

```
1. public class datatypedemo {
2.     public static void main(String[] args) {
3.         byte num;    // This can hold whole number between -128 and 127.
4.         num = 113;
5.         System.out.println(num);
6.
7.         short snum;
8.         snum = 150;
9.         System.out.println(snum);
10.
11.         long lnum = -12332252626L;
12.         System.out.println(lnum);
13.
14.         double dnum = -42937737.9d;
15.         System.out.println(dnum);
16.
17.         float fnum = 19.98f;
18.         System.out.println(fnum);
19.
20.         boolean b = false;
21.         System.out.println(b);
22.
23.         char ch = 'Z';
24.         System.out.println(ch);
25.     }
26. }
```

In this example, you will demonstrate how to declare variables with primitive Data types

Non-Primitive Data Types

21

Non-primitive data types are also called **Object references**, **Reference variables**, and **Reference data types**.

- ❑ We can extend the Java type system by defining classes.
- ❑ Once defined, a class can serve as a template for creating objects with capabilities and behaviors that go far beyond the built-in types.
- ❑ While non-primitive types have exact memory requirements, the memory needed for class-based types depends on the class definition.
- ❑ Memory to store objects is allocated in a memory area called the [heap](#).
- ❑ Reference variables reference, or point to this memory location.

Non-Primitive Data Types (continued)

22

- ❑ The **new** keyword is used to create objects.
- ❑ When objects are created, special methods called *constructors* run to initialize the object.
- ❑ The default value of any reference variable is *null*, which means that there is nothing there.

Example: `Integer Obj = new Integer(45);`

3. Assign the result of **new** to the variable.

1. Use the **new** operator to allocate memory in the heap.

2. Invoke a constructor defined by the Integer class, and initializing the value

Introduction to Java Literal

- Integer Literals
- Character Literals
- String Literals.
- Boolean Literals.
- Float Literals



- ❑ A *literal* is a number, text, or other information that directly represents a value.
 - A literal is not a value of any type that we read from the console, a file, or any other information source.
- ❑ *Literal* is a programming term that essentially means that what we type is what we get. The following assignment statement uses a literal:

```
int radius = 20;
```

- ❑ The literal is 20, because it directly represents the integer value 20.
- ❑ Numbers, characters, and strings can all be represented as literals.

Integer (int) Literals

25

An integer literal can be assigned to an integer variable, provided it is within the range of the variable's type.

- An integer literal is assumed to be an **int**, whose value is between -2^{31} (-2,147,483,648) to $2^{31}-1$ (2,147,483,647).

```
int i = 312;  
byte b = 58; // valid, since 58 < 128.
```

- To denote a long literal, append it with the letter "L." Lowercase "l" is allowed, but is strongly discouraged because it is easily be confused with "1" (the digit).

```
long lval = 3885L;
```

Floating-Point Literals

26

- ❑ Floating-point literals use a period character (.) for the decimal point. The following statement uses a literal to initialize a double variable:

```
double length = 1.5;
```

- ❑ All floating-point literals are considered type *double*.
- ❑ To specify a literal of float, append the letter F (F or f) to the literal:

```
float pi = 3.1415927F;
```

- ❑ Floating point literals can be written using scientific notation.

```
double avogadrosNumber = 6.022e23;  
double plancksConstant = 6.626e-34;
```

- ❑ The 'e' may be either lower- or upper-case.

Character (char) Literals

27

- ❑ Character literals are enclosed in single quotation marks. Any printable character, except for the backslash (\), can be specified this way.

```
char a = 'A', nine = '9', plus = '+', tilde = '~';
```

- ❑ Alternatively, we can specify a *char* literal as an integer literal. This code is identical to the code above:

```
char a = 0101, nine = 57, plus = 0x2b, tilde = 126;
```

- ❑ The integer can be specified using either decimal, octal, or hexadecimal forms.
- ❑ The allowed range is 0 to 65535.

Character Literals - Escape Sequences

28

- ❑ A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.
- ❑ The table on the right lists the Java escape sequences.
- ❑ Examples:

```
System.out.println("This will print on\r\ntwo lines.");  
System.out.println("She said \"Hello\" to me!");  
System.out.println("This will print two back-slashes: \\");
```

Result:

```
This will print on  
two lines.  
She said "Hello" to me!  
This will print two back-slashes: \
```

Escape Sequence	Description
\t	Tab character.
\b	Backspace character.
\n	Newline character.
\r	Carriage-return character.
\f	Form feed character.
'\'	Single-quote character.
'\"'	Double-quote character.
\\	Backslash character.

Character Literals - Unicode Characters

29

- ❑ We can specify char literals in Unicode representation (“\uxxxx”) where xxxx represents four hexadecimal digits.

```
char ch = '\u0061'; // \u0061 represents 'a'
```

- ❑ Unicode is not especially useful in console/terminal applications because terminals do not render characters outside of the standard ASCII character set.
- ❑ Unicode characters written into text files and displayed by GUI (Graphical User Interface) applications can take advantage of the full Unicode character set.

[Click here for Unicode Character Table.](#)

String Literals

30

- ❑ A string in Java is an object, not a primitive. Any sequence of characters within double quotes is treated as a String literal:

```
String username = "popcorn";  
String password = "123456";
```

- ❑ String literals are first-class objects. They have access to all of the methods defined by the String class:

```
String username = "pop".concat("corn");
```

- ❑ String literals are stored in the String Constant pool.

For more information, visit: [String Constant Pool](#).

Boolean Literals

31

Only two values are allowed for boolean literals: *true* and *false*.

```
boolean t = true;  
boolean f = false;
```

Practice Assignment - 2

32

Complete this assignment: **303.1.2 - Practice Assignment - Core Java - Variables.**

You can find this assignment on Canvas, under the **Assignment** section.

Use your office hours to complete this assignment. If you have technical questions while performing the practice assignment, ask your instructors for assistance.

Introduction to Java Scanner Class

- ❖ Scanner Class
- ❖ Popular Scanner Class Objects in Java



Reading input From the Console–

Let's return to the Scanner Class and look at it in more detail.

- ❑ Accepting input from a console, terminal, or file is a common task.
- ❑ Java has various ways to read input from various sources. The [Scanner](#) Class is one of them.
 - The Scanner class is used to parse string and primitive types from text input (int, double, etc. and strings, input streams, users, files, etc.).
 - The Scanner Class provides a variety of methods that perform these read operations.

For more information, visit: The Scanner Class of the [java.util.Scanner](#) package.

Scanner Class (continued)

35

- ❑ To use Scanner Class, include near to top of your Java file: `import java.util.Scanner;`
- ❑ Before a Java program can read input from the keyboard, the program must instantiate a Scanner object.
- ❑ The **System.in** parameter is used to take input from the standard input. It works just like taking inputs from the keyboard.

```
Scanner input = new Scanner(System.in); //System.in represents the keyboard, input is an identifier
int radius = input.nextInt() // Instead of hard-coding a radius
```

Method	Description
<code>nextLine()</code>	Reads an entire line of input as a <code>string</code> and advances to the next line.
<code>nextByte()</code>	Reads an integer of the <code>byte</code> type.
<code>nextShort()</code>	Reads an integer of the <code>short</code> type/
<code>nextInt()</code>	Reads an integer of the <code>int</code> type.
<code>nextLong()</code>	Reads an integer of the <code>long</code> type.
<code>nextDouble()</code>	Reads a number of the <code>double</code> type.
<code>nextFloat()</code>	Reads a number of the <code>float</code> type.
<code>next()</code>	<code>next()</code> method reads input up to the whitespace character. Once the whitespace is encountered, it returns the string (excluding the whitespace).

Popular Scanner Class Objects in Java

36

```
// read input from the input stream  
Scanner sc1 = new Scanner(InputStream input);
```

```
// read input from files  
Scanner sc2 = new Scanner(File file);
```

```
// read input from a string  
Scanner sc3 = new Scanner(String str);
```


Example: Scanner Class

37

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        // create an Object of Scanner class
        Scanner input = new Scanner(System.in);
        System.out.print("Enter your name: ");
        // read input(line of text) from the keyboard
        String name = input.nextLine();
        // prints the name
        System.out.println("My name is " + name);
        // closes the scanner
        input.close();
    } }
```

Output:

Enter your name: Kelvin

My name is Kelvin

Practice Assignment

38

Complete this assignment **303.1.1 - Practice Assignment - Scanner Class**

You can find this assignment on Canvas, under the **Assignment** section. If you have technical questions while performing the practice assignment, ask your instructors for assistance.

Note: It is NOT a mandatory assignment. This assignment does not count toward the final grade

Overview of Operators

- Arithmetic Operators.
- Relational Operators.
- Conditional Operators.
- Assignment Operators.
- Unary Operators.
- Bitwise and Bit Shift Operators.
- The Ternary Operator.
- Division and Modulus (Reminder).
- Remainder Operator.



Overview of Operators

40

- ❑ Operators are used to manipulate primitive data types. For example: To calculate an area, we used the expression:

```
area = radius * radius * 3.14159;
```

- ❑ Here, we are performing successive multiplications, and then assigning the result into a variable.
 - The symbol * is an arithmetic (type: multiplication) operator.
 - The symbol = is an assignment operator.
- ❑ Java operators can be classified as:
 - Arithmetic operators.
 - Relational operators.
 - Conditional operators.
 - Bitwise and Bit Shift operators.
 - Assignment operators.
 - Unary operators.
 - Ternary operators.

Arithmetic Operators

41

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Description	Example (Assume a,b,v are double)
+ Addition	Adds values on each side of the operator.	$v = a + b;$
- Subtraction	Subtracts right-hand operand from left-hand operand.	$v = a - b;$
* Multiplication	Multiplies values on each side of the operator.	$v = a * b$
/ Division	Divides left-hand operand by right-hand operand.	$v = a / b;$
% Modulus	Divides left-hand operand by right-hand operand and returns the remainder.	$v = b \% a;$

Relational Operators

42

Java has seven relational operators that compare two numbers and return a boolean value.

Careful: Do not confuse the assignment operator `=` with the equality operator `==`.

Operator	Description	Example (assume a=3, b=4, r is boolean)
== (equal to)	If the values of two operands are equal, then the condition becomes true.	<code>r = (a == b); // false</code>
!= (not equal to)	If values of two operands are not equal, then condition becomes true.	<code>r = (a != b); // true</code>
> (greater than)	If the value of left-hand operand is greater than the value of right-hand operand, then condition becomes true.	<code>r = (a > b); // false</code>
< (less than)	If the value of the left-hand operand is less than the value of the right-hand operand, then condition becomes true.	<code>r = (a < b); // true</code>
>= (greater than or equal to)	If the value of the left-hand operand is greater than or equal to the value of right-hand operand, then condition becomes true.	<code>r = (a >= b); // false</code>
<= (less than or equal to)	If the value of the left-hand operand is less than or equal to the value of the right-hand operand, then condition becomes true.	<code>r = (a <= b); // true</code>
instanceOf	Test whether the object is an instance of the specified type (class or subclass or interface).	<pre>class Student{ public static void main(String args[]){ Student s = new Student(); System.out.println(s instanceof Student); //true } }</pre>

Conditional Operators

43

Conditional operators are used to combine conditional statements and return a boolean value. These operators [short circuit](#), as described here:

Operator	Description	Example (a=3, b=4)
&& Logical AND	The conditional && operator does not check the second condition if the first condition is false. It checks the second condition only if the first one is <i>true</i> .	<pre>System.out.print (a<b && b<a); // true && false = false System.out.print (a>b && a<b); //false && true = false</pre>
 Logical OR	The conditional operator does not check the second condition if the first condition is true. It checks second condition only if the first one is <i>false</i> .	<pre>System.out.print (a<b b<a); // true false = true System.out.print (a>b a<b); // false true = true</pre>

Assignment Operators

44

Assignment operators are used in Java to assign the result of an operation on a variable back to the same variable:

Operator	Example	Equivalent
+= Addition Assignment	x += 5	x = x + 5
-= Subtraction Assignment	x -= 5	x = x - 5
*= Multiplication Assignment	x *= 5	x = x * 5
/= Division Assignment	x /= 5	x = x / 5
%= Remainder Assignment	x %= 5	x = x % 5
<<= Left Shift Assignment	x <<= 5	x = x << 5
>>= Right Shift Assignment	x >>= 5	x = x >> 5
&= Bitwise AND Assignment	x &= 5	x = x & 5
^= Bitwise XOR Assignment	x ^= 5	x = x ^ 5
 = Bitwise OR Assignment	x = 5	x = x 5

Unary Operators

45

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

Operator	Description	Example (a = 20, b = 10, c = 0, d = 20, e = 40, f = 30)
- Unary Minus	Used for negating the values.	<pre>int result = -a; // result is now -20 System.out.println(result); //-20</pre>
+ Unary Plus	Used to retain the current sign. This operator has no effect on a value.	<pre>int result = +a; // result is now 20 System.out.println(result); //20</pre>
++ Increment Operator	Used to incrementing a value by 1.	<pre>c = b++; System.out.println("Value of c (b++) = " + c); //Value of c (b++) = 10 c = ++a; System.out.println("Value of c (++a) = " + c); //Value of c (++a) = 21</pre>
	Postfix-Increment: Value is first used for computing a result and then incremented.	
-- Decrement Operator	Used for decrementing the value by 1.	<pre>c = e--; System.out.println("Value of c (e--) = " + c); //Value of c (e--) = 40 c = --d; System.out.println("Value of c (--d) = " + c); //Value of c (--d) = 19</pre>
	Postfix-Decrement: Value is first used for computing the result and then decremented.	
	Prefix-Decrement: Value is decremented first and then result is computed.	
! Logical Not Operator	Used for inverting a boolean value.	<pre>boolean condition = true; System.out.println("Value of !condition = " + !condition); //Value of !condition = false</pre>

Bitwise and Bit Shift Operators

46

Bitwise and Bit Shift operators work on bits and perform bit-by-bit operations. Unlike the Conditional Operators, both operands are evaluated before an operator is applied.

[For more information about Bitwise and Bit Shift Operators, see our **Wiki** document.](#)

The Ternary Operator

47

- The ternary operator is a shorthand version of the *if-else* statement. It has three operands; hence the name ternary. The name *ternary* refers to the fact that the operator takes three operands.

General format is: **condition ? exprTrue : exprFalse;**

- The above statement means that if the condition evaluates to *true*, then execute the expression after the “?”, else execute the expression after the “:,”

```
int age = 18;
String result;
if(age < 100){
    result = "Less than 100";
}else {
    result = "Greater than 100";
}
System.out.println(result);
```

Equivalent to

```
int age = 18;
String result = age < 100 ?
    "Less than 100" : "Greater than 100";
System.out.println(result); //Less than 100
```

Floating-Point Arithmetic

48

- ❑ Calculations involving floating-point numbers are approximated because these numbers cannot be stored with perfect precision. For example:

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);  
//displays 0.5000000000000001, not 0.5  
System.out.println(1.0 - 0.9);  
//displays 0.09999999999999998, not 0.1
```

- ❑ Integers are stored exactly. Therefore, calculations with integers yield an exact integer result.

Operator Precedence and Associativity

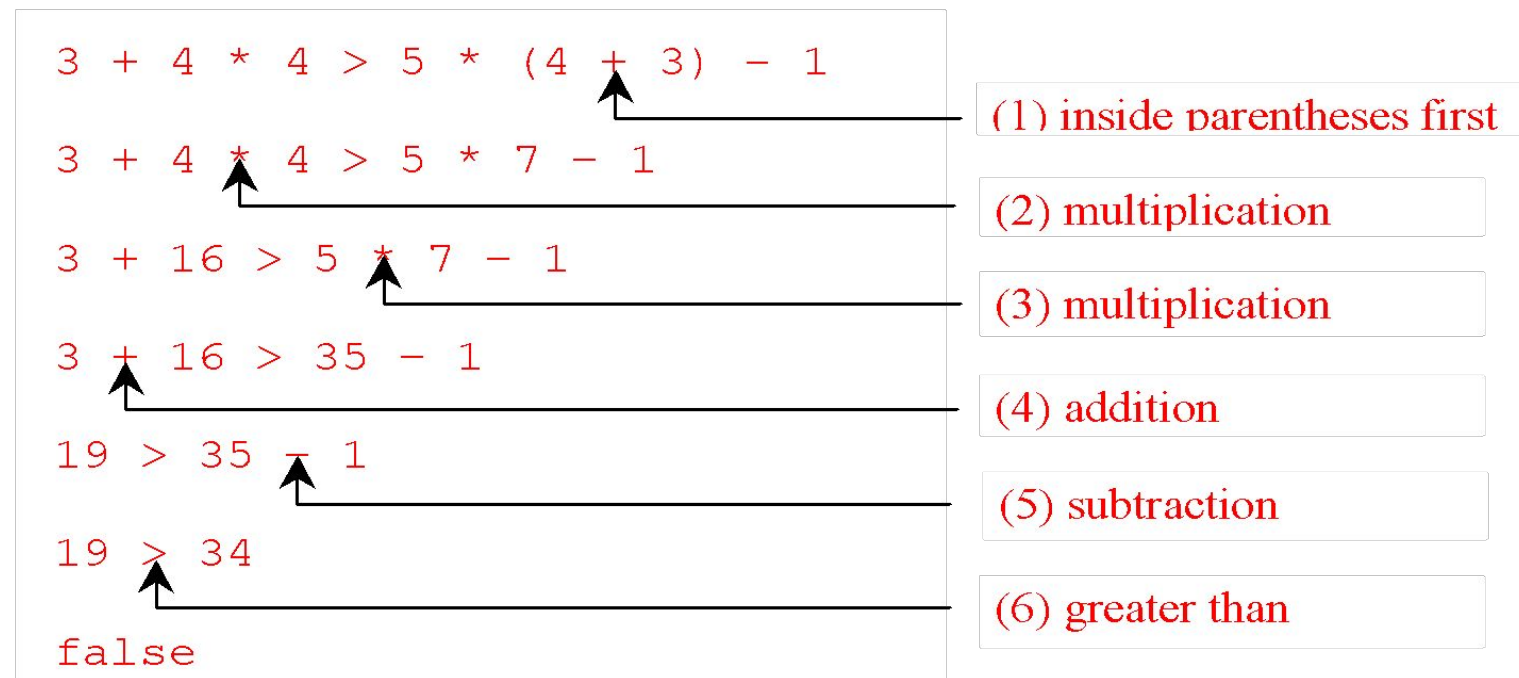
49

- ❑ The expression in the parentheses is evaluated first. Parentheses can be nested; in which case, the expression in the innermost parentheses is executed first.
- ❑ When evaluating an expression without (or within) parentheses, the operators are applied according to the precedence rule and the associativity rule.
- ❑ If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators, except assignment operators, are left-associative.

Example: Operator Precedence and Associativity

50

Applying the operator precedence and associativity rule, the expression $3 + 4 * 4 > 5 * (4 + 3) - 1$ is evaluated as follows:



Practice Lab - Operators

51

- Please go to [“Lab 303.1.2 - Core Java: Operators”](#) for learning more about Operators.
- You can find this lab on Canvas under the **Guided Lab section**. If you have technical questions while performing the lab, ask your instructors for assistance.

Practice Assignment - 3

52

Complete this assignment **303.1.3 - Practice Assignment - Operators and Numbers**, You can find this assignment on Canvas, under the **Assignment** section.

Use your office hours to complete this assignment. If you have technical questions while performing the practice assignment, ask your instructors for assistance.

Note: It is NOT a mandatory assignment. This assignment does not count toward the final grade

- ❑ Integer Division:

- $5 / 2$ yields an integer 2.

- ❑ Floating-point Division:

- $5.0 / 2$ yields an floating-point 2.5.

- ❑ Integer Modulus:

- $5 \% 2$ yields 1 (the remainder of the division).

- ❑ Floating-point Modulus:

- $3.6 \% 0.55$ yields 0.1 (the remainder of the division).

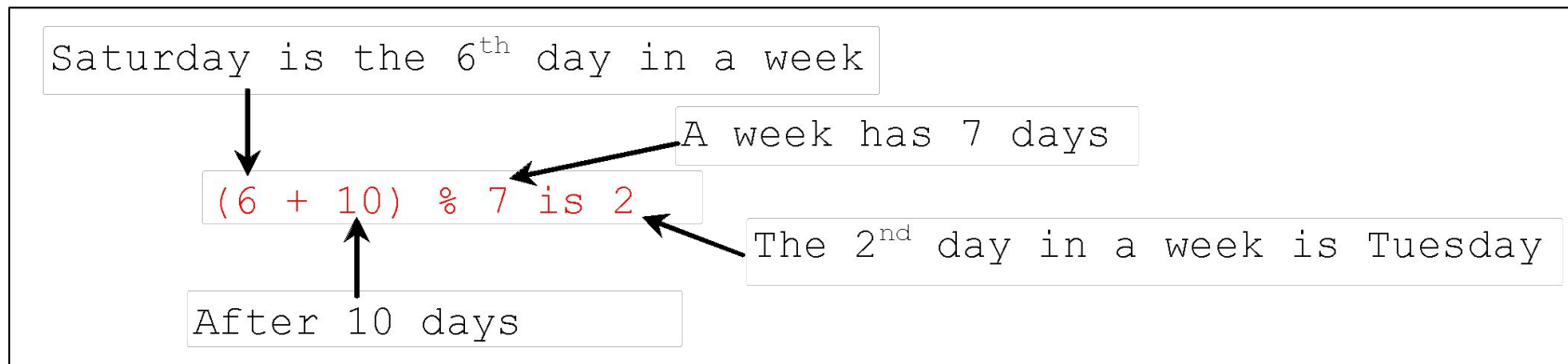
Remainder Operator

54

- ❑ Remainder is very useful in programming. For example, an even number $\% 2$ is always 0 and an odd number $\% 2$ is always 1. So you can use this property to determine whether a number is even or odd. Because odd, negative numbers return a negative remainder, always use the 0 test:

```
boolean isEven = (myValue % 2) == 0;
```

- ❑ Suppose today is Saturday and you and your friends are going to meet in 10 days. What day is it in 10 days? You can find that the day is Tuesday using the following expression:



Problem: Displaying Time

55

Write a Java program to convert a total amount of seconds to the hour, minute, and seconds.

Example case:

- ❑ Input seconds: 86399
- ❑ Output: 23:59:59

Solution : Displaying Time

56

```
public static void main(String args[]) {  
    int seconds = 86399;  
    int p1 = seconds % 60;  
    int p2 = seconds / 60;  
    int p3 = p2 % 60;  
    p2 = p2 / 60;  
    System.out.print( p2 + ":" + p3 + ":" + p1);  
    System.out.print("\n");  
}
```

- ❖ **Introduction to Type Casting**
 - Implicit Type Casting.
 - Explicit Type Casting.
- ❖ **Common Errors and Pitfalls in Java Program.**
- ❖ **Redundant/Unused Variables.**



Overview of Type Casting

58

- ❑ Assigning a value of one type to a variable of another type is known as **Type Casting**.
- ❑ Here, we discuss four types of Type Casting in Java:
 - Implicit Type Casting, Widening, or Automatic Type Conversion.
 - Explicit Type Casting or Narrowing.
 - Automatic Type Promotion in Expressions.
 - Explicit Type Casting in Expressions.

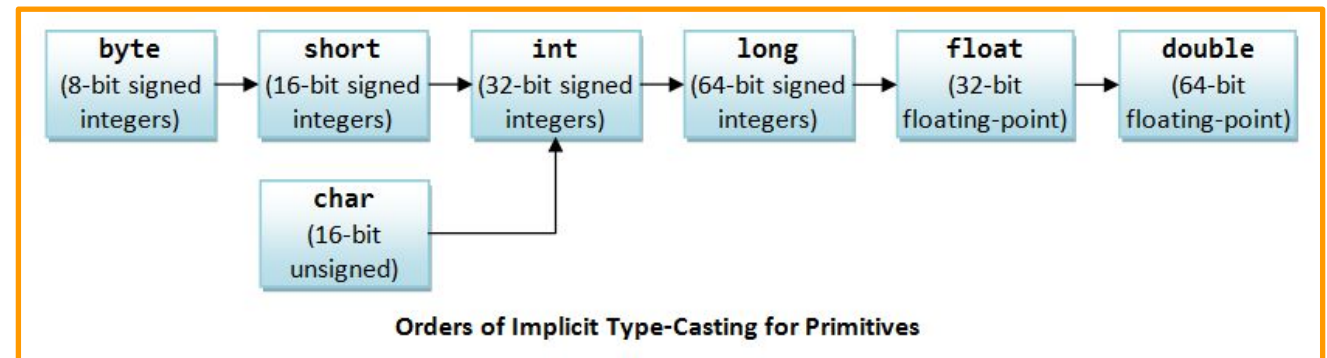
Implicit Type Casting

59

- ❖ Implicit type casting takes place when two data types are **automatically converted** by Java compiler internally. This happens when:
 - The two data types are compatible.
 - We assign value of a smaller data type to a bigger data type.
 - It is also called Widening or **Automatic Type Conversion**.
- ❖ The Java numeric data types are compatible with each other and with char.
- ❖ Boolean is not compatible with any other types.

Examples:

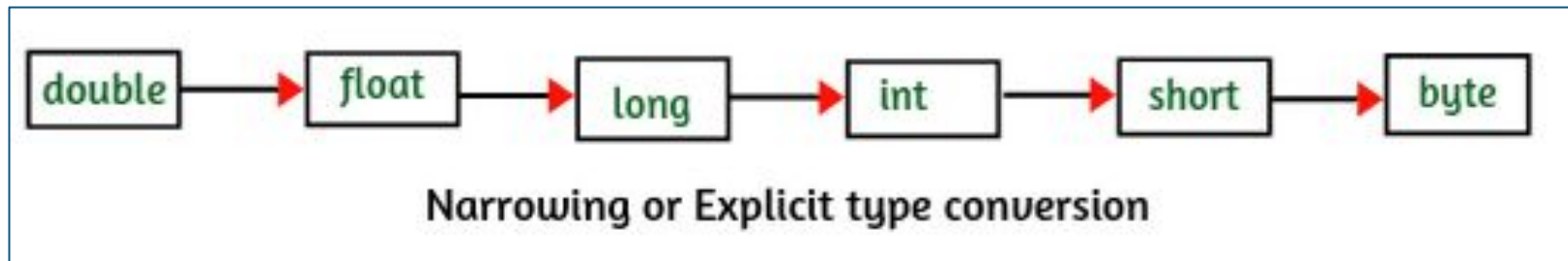
```
int radius = 20; //20
long l = radius; //20
float f = l; //20.0
```



Explicit Type Casting

60

- If we want to assign a value of a **larger data type with a smaller range**, we perform explicit type casting. This kind of conversion is also known as **narrowing conversion** in Java.



- The explicit cast operation has **syntax**: **(target-type) value;**
- Let's review an example to better understand the concept of Java explicit type casting.

```
int x;  
double y = 9.99;  
x = (int)y;    // It will compile in Java and the resulting value will simply be 9.
```

Basic Example

61

```
public static void main(String args[]) {  
    byte i = 40;  
    // No casting needed for below conversion  
    short j = i;  
    int k = j;  
    long l = k;  
    float m = l;  
    double n = m;  
    System.out.println("byte value : "+i);  
    System.out.println("short value : "+j);  
    System.out.println("int value : "+k);  
    System.out.println("long value : "+l);  
    System.out.println("float value : "+m);  
    System.out.println("double value : "+n);  
    s = short(k); // Not Allowed - narrowing  
    c = (char)(k); // Not Allowed - narrowing  
    float f = 1.5e3f;  
    k = (int)f; // Explicit - ok  
    k = f // Not Allowed - narrowing  
}
```

Automatic Type Promotion in Expressions

62

- ❑ Implicit (automatic) type conversions may occur in arithmetic in expressions.
- ❑ In an expression, the range required of an intermediate value will sometimes exceed the range of either operand.
- ❑ For example, consider the following statements:

```
byte a = 40, b = 50, c = 100;  
int d = a * b / c;
```

- ❑ The result of **a * b** exceeds the range of **byte**. To handle this, the compiler automatically promotes each **byte**, **short** or **char** operand to **int**.
- ❑ **a * b** is performed using integers.

Type-Promotion Rules

63

- ❑ All byte, short, and char values are promoted.
- ❑ Promotion to *int* is the default behavior, except:
 - ❑ If any operand is a long, the whole expression is promoted to long.
 - ❑ If any operand is a float, the entire expression is promoted to float.
 - ❑ If any of the operands is double, the result is double.

Explicit Type Casting in Expressions

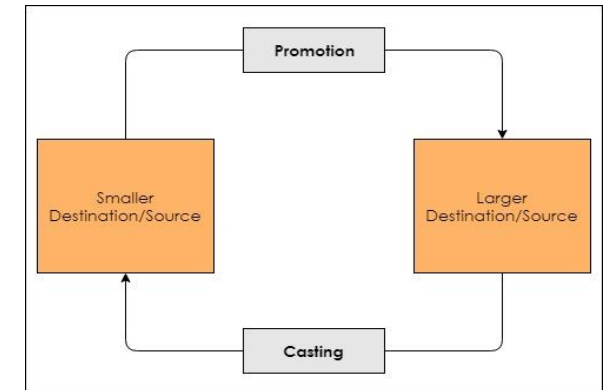
64

- ❑ Automatic promotions can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;  
b = b * 2; // Error! Cannot assign an int to a byte!
```

- ❑ The code is attempting to store 50×2 , a perfectly valid **byte** value, back into a **byte** variable.
- ❑ However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**.
- ❑ The result cannot be assigned to a **byte** without the use of a **cast**.
- ❑ This explicit cast solves the problem:

```
byte b = 50;  
b = (byte) b*2; //legal
```



Lab: Type Casting

65

- Go to “[LAB 303.1.3 - Core Java: Type Casting](#)” for more Type Casting practice
- You can find this lab on Canvas under the **Guided Lab section**. If you have technical questions while performing the lab activity, ask your instructors for assistance.

❑ Common Errors:

1. Undeclared/Uninitialized Variables.
2. Floating-point Roundoff Errors.
3. Unintended Integer Division.
4. Integer Overflow.

❑ Common Pitfalls:

1. Redundant/Unused Variables.

Undeclared/Uninitialized Variables

67

Attempting to use an undeclared or uninitialized variable is a compiler error.

Undeclared Variable:

```
135  
136 private static void errorsExamples() {  
137  
    interestRate = 0.05; // ????
```

✖ interestRate cannot be resolved to a variable

4 quick fixes available:

- [Create local variable 'interestRate'](#)
- [Create field 'interestRate'](#)
- [Create parameter 'interestRate'](#)
- ✖ [Remove assignment](#)

Uninitialized Variable:

```
private static void errorsExamples() {  
  
    double interestRate, principal;  
    double interest = principal * interestRate;  
  
}
```

✖ The local variable principal may not have been initialized

1 quick fix available:

- ➔ [Initialize variable](#)

Press 'F2' for focus

Redundant/Unused Variables

68

Compare this code:

```
int a = 1, b = 2, x = 5, y = x;  
System.out.println(a * x);  
System.out.println(b * y);
```

To this code:

```
int a = 1, b = 2, x = 5;;  
System.out.println(a * x);  
System.out.println(b * x);
```

- ❑ The first code example asks the compiler to allocate stack space for a redundant variable, **y**.
- ❑ This is not a critical error because it is neither a compiler error nor a logic error.
- ❑ As your coding skills improve, it is important to develop awareness of even minor errors like this.

A Variable in Java is a data container that stores the data values during Java program execution.

Data Types in Java are defined as specifiers that allocate different sizes and types of values that can be stored in the variable or an identifier. Java can be divided into two parts:

1. **Primitive Data Types**, which include integers, characters, booleans, and floats.
2. **Non-primitive Data Types**, which include classes, arrays, and interfaces.

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators.
- Relational Operators.
- Bitwise Operators.
- Logical Operators.
- Assignment Operators.

References

70

<https://beginnersbook.com/2017/08/data-types-in-java/>

<https://jenkov.com/tutorials/java/variables.html>

<https://www.scaler.com/topics/java/operators-in-java/#1--unary-operators>

<https://www.scientecheasy.com/2020/07/type-conversion-casting-java.html/>

Questions?

71



