# JAVA PROGRAMMING

Chapter 2

***Data Types and Operators***

# Objectives

- Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables.
- Write code that correctly applies the appropriate operators including assignment operators, arithmetic operators, relational operators, the instanceof operator, logical operators, and the conditional operator.
- Write code that determines the equality of two objects or two primitives.

# Data Types and Operators

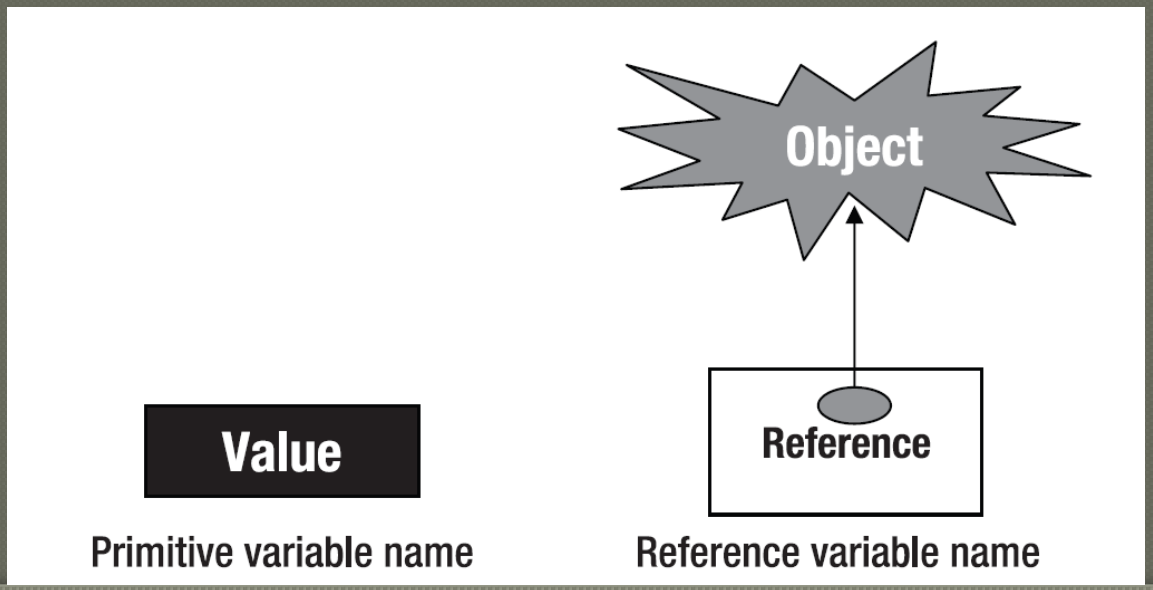
- **Data-Related Concepts**
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Data-Related Concepts

- Understanding Variables,Data Types, and Operators
- Naming the Variables: Legal Identifiers
- Reserved Names: The Keywords

# Understanding Variables,Data Types, and Operators

- 2 kinds of data types: *primitive* and *non-primitive*
- 2 corresponding kinds of variables: *primitive variables,* and *reference variables – object references*



# Naming the Variables: Legal Identifiers

- Each variable has a name, called an *identifier.*
- Rules to name a variable:
  - The first character must be a letter, a dollar sign ($), or an underscore (_).
  - A character other than the first character in an identifier may be a letter, a dollar sign, an underscore, or a digit.
  - None of the Java language keywords (or reserved words) can be used as identifiers.

# Reserved Names: The Keywords

**Table 2-1.** *Java Keywords*

| | | | | | |
|---|---|---|---|---|---|
| abstract | const | final | int | public | throw |
| assert | continue | finally | interface | return | throws |
| boolean | default | float | long | short | transient |
| break | do | for | native | static | true |
| byte | double | goto | new | strictfp | try |
| case | else | if | null | super | void |
| catch | enum | implements | package | switch | volatile |
| char | extends | import | private | synchronized | while |
| class | false | instanceof | protected | this | |

# Data Types and Operators

- Data-Related Concepts
- **Working with Primitive Data Types**
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Working with Primitive Data Types

**Table 2-2.** *Primitive Data Types, Their Sizes, and the Ranges of Their Values*

| Data Type | Size in Bits | Range of Values | Signed/Unsigned |
|---|---|---|---|
| `boolean` | 1 | `true` or `false` | NA |
| `byte` | 8 | $-2^7$ to $2^7 - 1$ | Signed |
| `short` | 16 | $-2^{15}$ to $2^{16} - 1$ | Signed |
| `char` | 16 | 0 to $2^{15} - 1$ | Unsigned |
| `int` | 32 | $-2^{31}$ to $2^{31} - 1$ | Signed |
| `float` | 32 | $-2^{31}$ to $2^{31} - 1$ | Signed |
| `double` | 64 | $-2^{63}$ to $2^{63} - 1$ | Signed |
| `long` | 64 | $-2^{63}$ to $2^{63} - 1$ | Signed |

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- **Declaring and Initializing Primitive Variables**
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Declaring and Initializing Primitive Variables

- Syntax for Declaring Variables
- Accessing Variables
- Literals
- Default Initial Values

# Syntax for Declaring Variables

- The general syntax for declaring and initializing a variable:

```
<modifier> <dataType>
<variableName> = <initialValue>;

private int id = 10;

int id;
```

# Accessing Variables

- Once you declare a variable, you can access it by referring to it by its name:

```
x = y;
```

- Variables can be classified into three categories:
  - *Local variables*
  - *Instance variables*
  - *Static variables*

# Literals

- A literal is a value assigned to a variable in the source code

```
int id = 10;
```

- The **`boolean`** Literals: **`true`** or **`false`**
- The **`char`** Literals: **`'L'`**, **`'\u4567'`**, **`'\n'`**
- The Integral Literals: **`43`**, **`43L`**, **`053`**, **`0x2b`**
- The Floating-Point Literals: **`12.33`**, **`1.25E+8`**, **`1.2534f`**

# Default Initial Values

- Only the instance variables acquire the default values if not explicitly initialized

**Table 2-3.** *Default Initial Values for Variables of Different Data Types*

| Type | Default Initial Value |
|---|---|
| boolean | false |
| byte | 0 |
| short | 0 |
| char | `\u0000` |
| int | 0 |
| float | 0.0f |
| double | 0.0d |
| long | 0L |

# Example

**Listing 2-1.** *InitialTest.java*

```
1. public class InitialTest {
2.       int x;
3.    public static void main(String[] args) {
4.        new InitialTest().printIt();
5.     }
6.     public void printIt(){
7.        int y;

8.        int z;
9.        y=2;
10.      System.out.println(x +" "+ y);
11.    // System.out.println(z);
12.    }
13. }
```

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- **Working with Nonprimitive Data Types**
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Working with Nonprimitive Data Types

- All non-primitive data types in Java are objects
- You create an object by instantiating a class
- When you declare a variable of a non-primitive data type, you actually declare a variable that is a reference - *reference variable / object reference* - to the memory where an object lives

# Working with Nonprimitive Data Types

- Objects
- Arrays
- The Data Type enum

# Objects

- An object reference (a reference variable) is declared:

  ```
  Student studentOne;
  ```

- You create an object with the new operator

  ```
  studentOne = new Student();
  ```

- The declaration of the object reference variable, object creation, and initialization of the reference variable:

  ```
  Student studentOne = new Student();
  ```

# Arrays

- Objects that are used to store multiple variables of the same type

- Making an array of data items consists of three logical steps:
  1. Declare an array variable.
  2. Create an array of a certain size and assign it to the array variable.
  3. Assign a value to each array element.

# Declaring an Array Variable

- declare an array by specifying the data type of the elements that the array will hold, followed by the identifier, plus a pair of square brackets before or after the identifier

- Example:

```
int[] scores;
int scores [];
Student[] students;
```

# Creating an Array

- create an array with the new operator
- An array of primitives is created and assigned to an already declared array variable:

```
scores = new int[3];
```

- An array of a non-primitive data type is created and assigned to an already declared array variable:

```
students = new Student[3];
```

# Assigning Values to Array Elements

- Each element of an array needs to be assigned a value (primitive type / object reference):

```
scores[0] = 75;
scores[1] = 80;
scores[2] = 100;
students[0] = new Student();
students[1] = new Student();
students[2] = new Student();
```

# The Data Type enum

- use enums any time you need a fixed set of constants such as days of the week
- define an enum variable in two steps:
  1. Define the enum type with a set of named values.
  2. Define a variable to hold one of those values.
- Example:

```
enum AllowedCreditCard {VISA,
  MASTER_CARD, AMERICAN_EXPRESS};
AllowedCreditCard visa =
  AllowedCreditCard.VISA;
```

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- **Understanding Operations on Data**
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Understanding Operations on Data

- *Unary operators*: Require only one operand. For example, ++ increments the value of its operand by one.
- *Binary operators*: Require two operands. For example, + adds the values of its two operands.
- *Ternary operators*: Operate on three operands. The Java programming language has one ternary operator, ?:

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- **Arithmetic Operators**
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Arithmetic Operators

**Table 2-4.** *Arithmetic Operators Supported by Java*

| Operator | Use | Description |
|---|---|---|
| + | op1 + op2 | Adds the values of op1 and op2 |
| ++ | ++op<br>op++ | Increments the value of op by 1 |
| – | op1 – op2 | Subtracts the value of op2 from that of op1 |
| -- | --op<br>op-- | Decrements the value of op by 1 |
| * | op1 * op2 | Multiplies value of op1 by that of op2 |
| / | op1 / op2 | Divides the value of op1 by that of op2 |
| % | op1 % op2 | Computes the remainder of dividing the value of op1 by that of op2 |

# The Unary Arithmetic Operators

- The Sign Unary Operators: + and –
- The Increment and Decrement Operators: ++ and –

**Table 2-5.** *Examples of Using Increment and Decrement Unary Operators*

| Initial Value of x | Code Statement | Final Value of y | Final Value of x |
|---|---|---|---|
| 7 | `y = ++x;` | 8 | 8 |
| 7 | `y = x++;` | 7 | 8 |
| 7 | `y = --x;` | 6 | 6 |
| 7 | `y = x--;` | 7 | 6 |

# The Multiplication and Division Operators: * and /

- operate on all primitive numeric types and the type `char`
- The result of dividing an integer by another integer will be an integer
- In case of integer types, division by zero would generate an `ArithmeticException` at execution time
- Division by zero in case of float and double types would generate `IPOSITIVE_INFINITY` or `NEGATIVE_INFINITY`
- The square root of a negative number of float or double type would generate an NaN (Not a Number) value: `Float.NaN`, and `Double.NaN`.

# The Modulo Operator: %

- gives the value that is the remainder of a division
- The sign of the result is always the sign of the first (from the left) operand

**Table 2-6.** *Examples of Using the Modulo Operator*

| Value of x | Value of y | Expression | Final Value of z |
|---|---|---|---|
| 11 | 3 | `z = x%y` | 2 |
| 11 | 3 | `z = x%(-y)` | 2 |
| 11 | 3 | `z = -x%y` | –2 |
| 11 | 3 | `z = -x%(-y)` | –2 |
| 3.8 | 1.2 | `z=x%y` | 0.2 |

# The Addition and Subtraction Operators: + and –

- perform arithmetic addition and subtraction
- If the result overflows, the truncation of bits happens the same way as in multiplication
- The + operator is overloaded in the Java language to concatenate strings

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- **Relational Operators**
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Relational Operators

- also called a comparison operator, compares the values of two operands and returns a `boolean` value: `true` or `false`

**Table 2-7.** *Comparison Operators*

| Operator | Use | Result |
|---|---|---|
| > | op1 > op2 | `true` if op1 is greater than op2, otherwise `false` |
| >= | op1 >= op2 | `true` if op1 is greater than or equal to op2, otherwise `false` |
| < | op1 < op2 | `true` if op1 is less than op2, otherwise `false` |
| <= | op1 <= op2 | `true` if op1 is less than or equal to op2, otherwise `false` |
| == | op1 == op2 | `true` if op1 and op2 are equal, otherwise `false` |
| != | op1 != op2 | `true` if op1 and op2 are not equal, otherwise `false` |

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- **Logical Operators**
- Using Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Logical Operators

- used to combine more than one condition that may be true or false
- deal with connecting the boolean values
- operate at bit level
- two kinds of logical operators:
  - bitwise logical operators
  - short-circuit logical operators

# Bitwise Logical Operators

- manipulate the bits of an integer (byte, short, char, int, long) value

**Table 2-8.** *Bitwise Logical Operators*

| Operator | Use | Operation |
|---|---|---|
| & | op1 & op2 | AND |
| \| | op1 \| op2 | OR |
| ^ | op1 ^ op2 | XOR |
| ~ | ~op | Bitwise inversion |
| ! | !op | NOT (Boolean inversion) |

# Bitwise Logical Operators

```
byte x = 117;
byte y = 89;
byte z = (byte) (x&y);
System.out.println("Value of z: " + z );
```

```
  01110101
& 01011001
-----------
  01010001   =  81.
```

```
  01110101
^ 01011001
-----------
  00101100   =  44.
```

```
  01110101
| 01011001
-----------
  01111101   =  125.
```

# Short-Circuit Logical Operators

- operate on the `boolean` types
- The outcome of these operators is a `boolean`

**Table 2-12.** *Short-Circuit Logical Operators*

| Operator | Name | Usage | Outcome |
|---|---|---|---|
| && | Short-circuit logical AND | op1 && op2 | true if op1 and op2 are both true, otherwise false. Conditionally evaluates op2. |
| \|\| | Short-circuit logical OR | op1 \|\| op2 | `true` if either op1 or op2 is true, otherwise `false`. Conditionally evaluates op2. |

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- **Using Assignment Operators**
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Using Assignment Operators

- used to set (or reset) the value of a variable:

```
x = 7;
```

- *shortcut assignment operators:*

**Table 2-13.** *Shortcut Assignment Operators*

| Operator | Use | Equivalent To |
|---|---|---|
| += | op1 += op2 | op1 = op1 + op2 |
| -= | op1 -= op2 | op1 = op1 - op2 |
| *= | op1 *= op2 | op1 = op1 * op2 |
| /= | op1 /= op2 | op1 = op1 / op2 |
| %= | op1 %= op2 | op1 = op1 % op2 |
| &= | op1 &= op2 | op1 = op1 & op2 |
| \|= | op1 \|= op2 | op1 = op1 \| op2 |
| ^= | op1 ^= op2 | op1 = op1 ^ op2 |
| <<= | op1 <<= op2 | o1 = op1 << op2 |
| >>= | op1 >>= op2 | o1 = op1 >> op2 |
| >>>= | op1 >>>= op2 | o1 = op1 >>> op2 |

# Arithmetic Promotion

- involves binary operation between two operands of different types or of types narrower in size than `int`
- the compiler may convert the type of one operand to the type of the other operand, or the types of both operands to entirely different types
- *Arithmetic promotion* is performed before any calculation is done

# Arithmetic Promotion Rules

- If both the operands are of a type narrower than int (that is `byte`, `short`, or `char`), then both of them are promoted to type `int` before the calculation is performed.
- If one of the operands is of type `double`, then the other operand is converted to `double` as well.
- If none of the operands is of type `double`, and one of the operands is of type `float`, then the other operand is converted to type `float` as well.

# Arithmetic Promotion Rules

- If none of the operands is of type `double` or `float`, and one of the operands is of type `long`, then the other operand is converted to type `long` as well.
- If none of the operands is of type `double`, `float`, or `long`, then both the operands are converted to type `int`, if they already are not.

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- **Advanced Operators**
- Equality of Two Objects or Two Primitives

# Advanced Operators

**Table 2-14.** *Advanced Operators Offered by Java*

| Operator | Description |
|---|---|
| ?: | Shortcut if-else statement |
| [ ] | Used to declare arrays, create arrays, and access array elements |
| . | Used to form qualified names of class members |
| ( <*params*> ) | Delimits a comma-separated list of parameters; used, for example, in a method declaration |
| ( <*type*> ) | Casts (converts) a value to a specified type |
| new | Creates a new object or a new array |
| instanceof | Determines whether its first operand is an instance of its second operand |

# The Shortcut if-else Operator: ?:

```
if (x) {
     a=b;
else {
   a=c;
}
```

```
a = x ? b : c;
```

# Advanced Operators

- The cast operator: (<*type*>) explicitly converts a value to the specified type

```
byte z = (byte) (x/y);
```

- The `new` operator: instantiate a class and to create an array
- The `instanceof` operator: determines if a given object is of the type of a specific class

```
<op1> instanceof <op2>
```

# Data Types and Operators

- Data-Related Concepts
- Working with Primitive Data Types
- Declaring and Initializing Primitive Variables
- Working with Nonprimitive Data Types
- Understanding Operations on Data
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Using Assignment Operators
- Advanced Operators
- **Equality of Two Objects or Two Primitives**

# Equality of Two Objects or Two Primitives

Three kinds of elements that can be compared to test the equality:

- Primitive variables:
  - hold the same value
  - can be tested with the == operator
- Reference variables:
  - can be compared by using the == operator
  - hold the same value
- Objects:
  - tested with the `equals()` method of the Object class.

# Codewalk Quicklet

**Listing 2-3.** *CodeWalkOne.java*

```
1. class CodeWalkOne {
2.     public static void main(String [] args) {
3.       int [] counts = {1,2,3,4,5};
4.       counts[1] = (counts[2] == 2) ? counts[3] : 99;
5.       System.out.println(counts[1]);
6.     }
7. }
```