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
- 2 kinds of data types: *primitive* and *non-primitive*
- 2 corresponding kinds of variables: *primitive variables*, and *reference variables* – *object references*
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>
<thead>
<tr>
<th>keyword</th>
<th>keyword</th>
<th>keyword</th>
<th>keyword</th>
<th>keyword</th>
<th>keyword</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>const</td>
<td>final</td>
<td>int</td>
<td>public</td>
<td>throw</td>
<td></td>
</tr>
<tr>
<td>assert</td>
<td>continue</td>
<td>finally</td>
<td>interface</td>
<td>return</td>
<td>throws</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>default</td>
<td>float</td>
<td>long</td>
<td>short</td>
<td>transient</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>do</td>
<td>for</td>
<td>native</td>
<td>static</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>double</td>
<td>goto</td>
<td>new</td>
<td>strictfp</td>
<td>try</td>
<td></td>
</tr>
<tr>
<td>case</td>
<td>else</td>
<td>if</td>
<td>null</td>
<td>super</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td>catch</td>
<td>enum</td>
<td>implements</td>
<td>package</td>
<td>switch</td>
<td>volatile</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>extends</td>
<td>import</td>
<td>private</td>
<td>synchronized</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>false</td>
<td>instanceof</td>
<td>protected</td>
<td>this</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
### Table 2-2. Primitive Data Types, Their Sizes, and the Ranges of Their Values

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size in Bits</th>
<th>Range of Values</th>
<th>Signed/Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>true or false</td>
<td>NA</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>$-2^7$ to $2^7 - 1$</td>
<td>Signed</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>$-2^{15}$ to $2^{16} - 1$</td>
<td>Signed</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>0 to $2^{15} - 1$</td>
<td>Unsigned</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
<td>Signed</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>$-2^{31}$ to $2^{31} - 1$</td>
<td>Signed</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
<td>Signed</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>$-2^{63}$ to $2^{63} - 1$</td>
<td>Signed</td>
</tr>
</tbody>
</table>
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
The general syntax for declaring and initializing a variable:

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

```java
private int id = 10;
```

```java
int id;
```
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*
A literal is a value assigned to a variable in the source code

```java
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`
Only the instance variables acquire the default values if not explicitly initialized

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>char</td>
<td><code>\u0000</code></td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
</tbody>
</table>
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
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):

  ```java
  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:

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>op1 + op2</td>
<td>Adds the values of op1 and op2</td>
</tr>
<tr>
<td>++</td>
<td>++op</td>
<td>Increments the value of op by 1</td>
</tr>
<tr>
<td></td>
<td>op++</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>op1 - op2</td>
<td>Subtracts the value of op2 from that of op1</td>
</tr>
<tr>
<td></td>
<td>--op</td>
<td>Decrements the value of op by 1</td>
</tr>
<tr>
<td></td>
<td>op--</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>op1 * op2</td>
<td>Multiplies value of op1 by that of op2</td>
</tr>
<tr>
<td>/</td>
<td>op1 / op2</td>
<td>Divides the value of op1 by that of op2</td>
</tr>
<tr>
<td>%</td>
<td>op1 % op2</td>
<td>Computes the remainder of dividing the value of op1 by that of op2</td>
</tr>
</tbody>
</table>
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*

<table>
<thead>
<tr>
<th>Initial Value of x</th>
<th>Code Statement</th>
<th>Final Value of y</th>
<th>Final Value of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>y = ++x;</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>y = x++;</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>y = --x;</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>y = x--;</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Value of x</th>
<th>Value of y</th>
<th>Expression</th>
<th>Final Value of z</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>3</td>
<td>$z = x % y$</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>$z = x % (-y)$</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>$z = -x % y$</td>
<td>-2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>$z = -x % (-y)$</td>
<td>-2</td>
</tr>
<tr>
<td>3.8</td>
<td>1.2</td>
<td>$z = x % y$</td>
<td>0.2</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>op1 &gt; op2</td>
<td>true if op1 is greater than op2, otherwise false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>op1 &gt;= op2</td>
<td>true if op1 is greater than or equal to op2, otherwise false</td>
</tr>
<tr>
<td>&lt;</td>
<td>op1 &lt; op2</td>
<td>true if op1 is less than op2, otherwise false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>op1 &lt;= op2</td>
<td>true if op1 is less than or equal to op2, otherwise false</td>
</tr>
<tr>
<td>==</td>
<td>op1 == op2</td>
<td>true if op1 and op2 are equal, otherwise false</td>
</tr>
<tr>
<td>!=</td>
<td>op1 != op2</td>
<td>true if op1 and op2 are not equal, otherwise false</td>
</tr>
</tbody>
</table>
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
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>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>op1 &amp; op2</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>op1</td>
</tr>
<tr>
<td>^</td>
<td>op1 ^ op2</td>
<td>XOR</td>
</tr>
<tr>
<td>~</td>
<td>~op</td>
<td>Bitwise inversion</td>
</tr>
<tr>
<td>!</td>
<td>!op</td>
<td>NOT (Boolean inversion)</td>
</tr>
</tbody>
</table>
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.
operate on the boolean types
The outcome of these operators is a boolean

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Usage</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Short-circuit logical AND</td>
<td>op1 &amp;&amp; op2</td>
<td>true if op1 and op2 are both true, otherwise false. Conditionally evaluates op2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Short-circuit logical OR</td>
</tr>
</tbody>
</table>
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
used to set (or reset) the value of a variable:

\[ x = 7; \]

**shortcut assignment operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>( \text{op1} += \text{op2} )</td>
<td>( \text{op1} = \text{op1} + \text{op2} )</td>
</tr>
<tr>
<td>-=</td>
<td>( \text{op1} -= \text{op2} )</td>
<td>( \text{op1} = \text{op1} - \text{op2} )</td>
</tr>
<tr>
<td>*=</td>
<td>( \text{op1} *= \text{op2} )</td>
<td>( \text{op1} = \text{op1} \times \text{op2} )</td>
</tr>
<tr>
<td>/=</td>
<td>( \text{op1} /= \text{op2} )</td>
<td>( \text{op1} = \text{op1} / \text{op2} )</td>
</tr>
<tr>
<td>%=</td>
<td>( \text{op1} %= \text{op2} )</td>
<td>( \text{op1} = \text{op1} % \text{op2} )</td>
</tr>
<tr>
<td>&amp;=</td>
<td>( \text{op1} &amp;= \text{op2} )</td>
<td>( \text{op1} = \text{op1} &amp; \text{op2} )</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>( \text{op1}</td>
</tr>
<tr>
<td>^=</td>
<td>( \text{op1} ^= \text{op2} )</td>
<td>( \text{op1} = \text{op1} ^ \text{op2} )</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>( \text{op1} &lt;&lt;= \text{op2} )</td>
<td>( \text{op1} = \text{op1} &lt;&lt; \text{op2} )</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>( \text{op1} &gt;&gt;= \text{op2} )</td>
<td>( \text{op1} = \text{op1} &gt;&gt; \text{op2} )</td>
</tr>
<tr>
<td>&gt;&gt;&gt;=</td>
<td>( \text{op1} &gt;&gt;&gt;= \text{op2} )</td>
<td>( \text{op1} = \text{op1} &gt;&gt;&gt; \text{op2} )</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?:</code></td>
<td>Shortcut if-else statement</td>
</tr>
<tr>
<td><code>[ ]</code></td>
<td>Used to declare arrays, create arrays, and access array elements</td>
</tr>
<tr>
<td>.</td>
<td>Used to form qualified names of class members</td>
</tr>
<tr>
<td><code>&lt;params&gt;</code></td>
<td>Delimits a comma-separated list of parameters; used, for example, in a method declaration</td>
</tr>
<tr>
<td><code>&lt;type&gt;</code></td>
<td>Casts (converts) a value to a specified type</td>
</tr>
<tr>
<td><code>new</code></td>
<td>Creates a new object or a new array</td>
</tr>
<tr>
<td><code>instanceof</code></td>
<td>Determines whether its first operand is an instance of its second operand</td>
</tr>
</tbody>
</table>
The Shortcut if-else Operator: ?:

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

```
a = x ? b : c;
```
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**
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.
Listing 2-3. CodeWalkOne.java

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