The Genesis of JAVA

- Java is related to C++, which is a direct descendent of C.
- From C, Java derives its syntax.
- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- This language was initially called “Oak” but was renamed “Java”in 1995.

- primary motivation →the need for a platform-independent language Web, too, demanded portable programs.
- By 1993, the focus of Java switch from consumer electronics to Internet programming.
- Architecture-Neutral Programming language.
- Computer languages evolve for two reasons:
  - → to adapt to changes in environment
  - → to implement advances in the art of programming.

- Java
  - Environmental change → platform-independent programs
  - Advances in programming → enhancement of object-oriented paradigm used by C++.
- Java → Internet Programming
- C → systems programming

Java Applets and Applications

- Java can be used to create two types of programs: **applications and applets.**
- An ***application*** *is a program that runs on your computer, under the operating system of that* computer.
- An ***applet*** *is an application designed to be transmitted over the Internet and executed by* a Java-compatible Web browser.
- An applet is an ***intelligent program***, that can react to user input and dynamically change.

Java's Magic: The Bytecode

- The output of a Java compiler is not executable code. Rather, it is bytecode.
- *Bytecode is a highly optimized set of instructions designed to be executed* by the Java run-time system, which is called the *Java Virtual Machine (JVM).*
- JVM is an *interpreter for bytecode.*
- Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments, only the JVM needs to be implemented for each platform.
- Sun supplies its Just In Time (JIT) compiler for bytecode, which is a part of JVM. It compiles bytecode into executable code in realtime, on a piece-by-piece, demand basis.

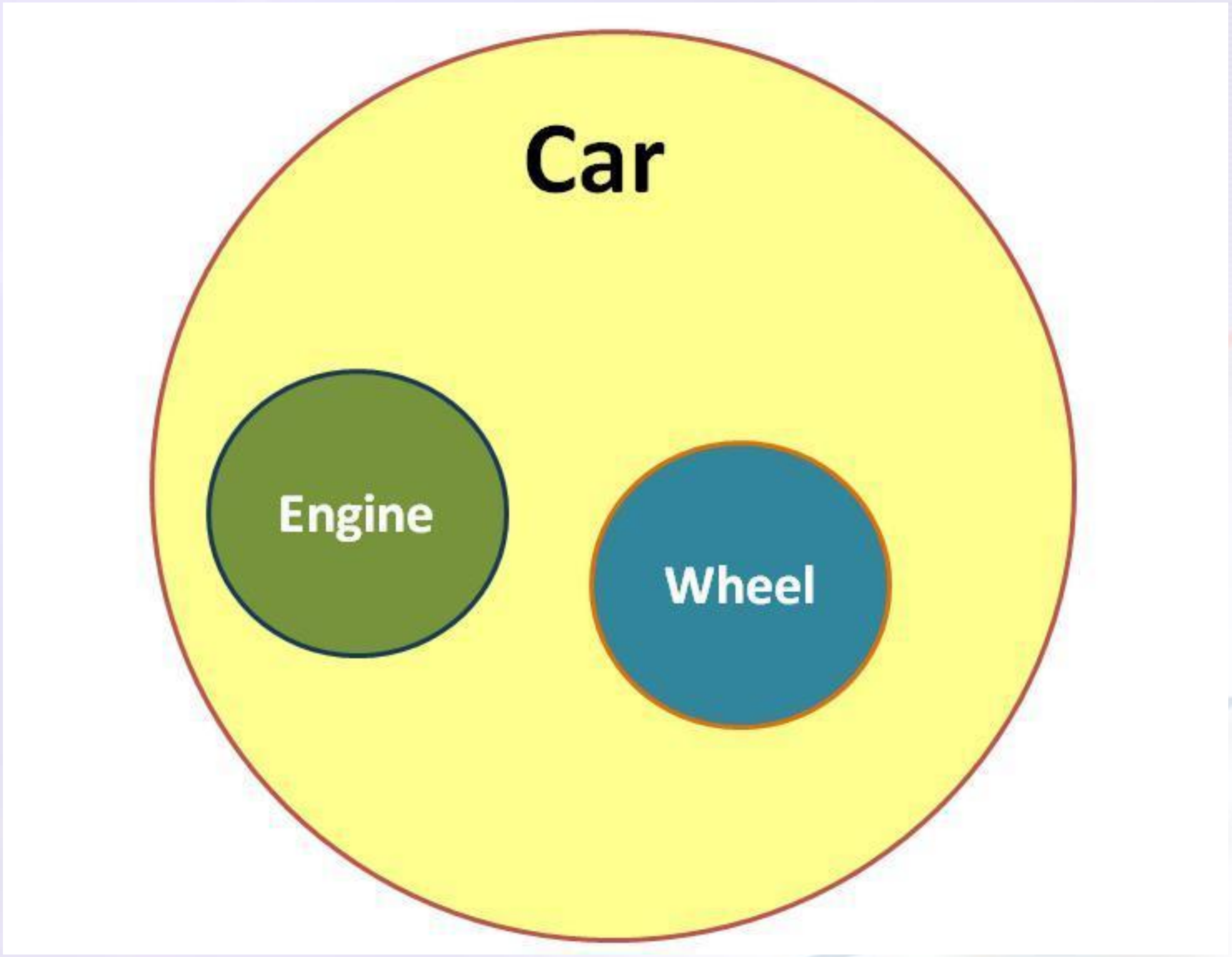
The Java Buzzwords

- Simple
- Secure
- Portable
- Object-oriented
- Robust → checks code at compile time & run time.
- Multithreaded
- Architecture-neutral → "write once; run anywhere, any time, forever."
- Interpreted
- High performance → perform well on low power CPUs.
- Distributed→handles TCP/IP protocols; Java-RMI package
  -client/server programming
- Dynamic

JAVA OVERVIEW

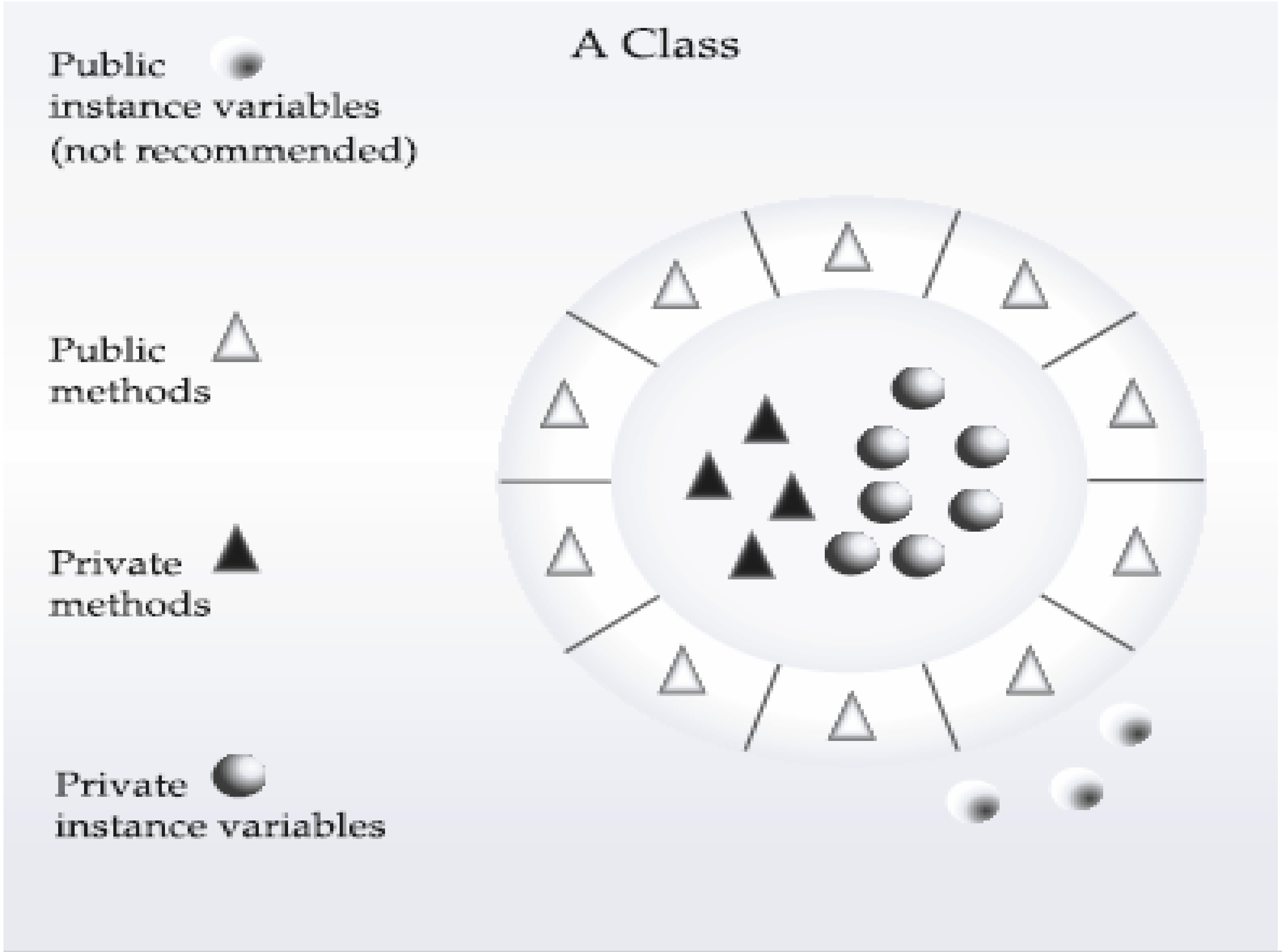
- Object-Oriented Programming
  - All Java programs are object oriented.
- Two Paradigms
  - all computer programs consist of two elements:
    - Code → 'what is happening'
    - Data → 'who is being affected'
  - Process-oriented model
    - Code acting on data
  - Object-oriented model
    - Data controlling access to data

- **Abstraction**
  - An essential element of object-oriented programming is *abstraction.*
  - A powerful way to manage abstraction is through the use of hierarchical classifications.
  - This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.
  - The data from a traditional process-oriented program can be transformed by abstraction into its component objects.


Car
Engine
Wheel

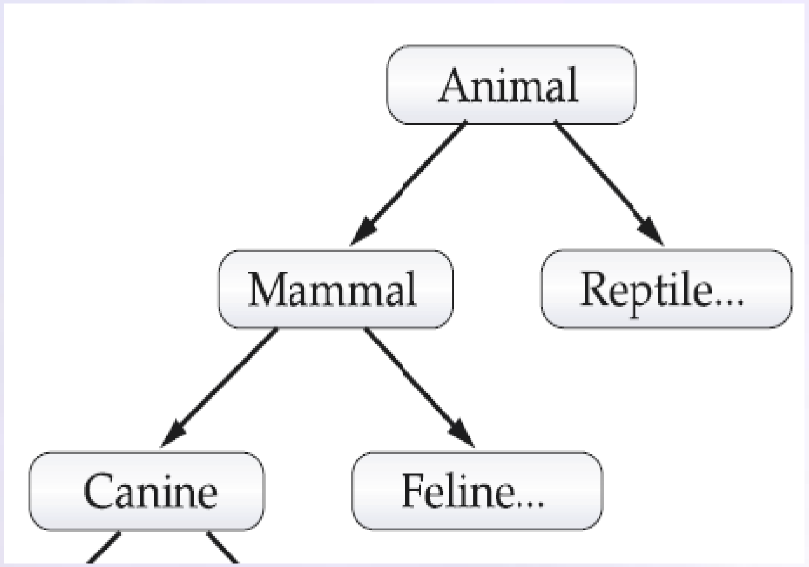
The Three OOP Principles

- **Encapsulation**
  - Mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
  - In Java the basis of encapsulation is the class.
  - Each method or variable in a class may be marked private or public.
  - The *public interface of a class represents* everything that external users of the class need to know, or may know.
  - The *private* methods and data can only be accessed by code that is a member of the class.


A Class
Public instance variables (not recommended)
Public methods
Private methods
Private instance variables

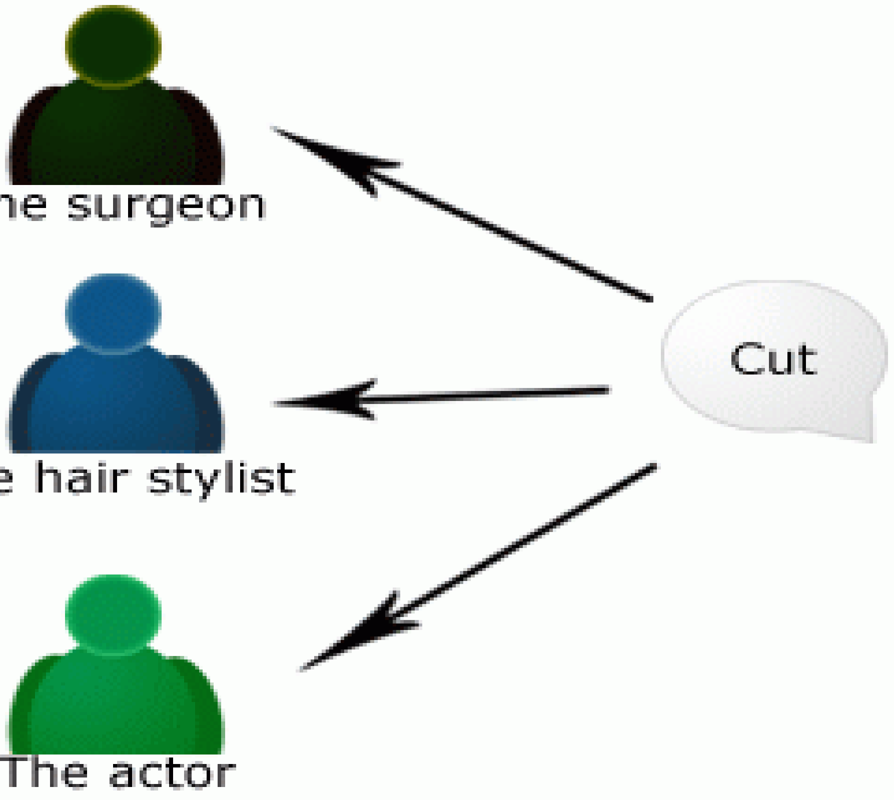
Encapsulation: public methods can be used to protect private data

- **Inheritance**
  - *Inheritance is the process by which one object acquires the properties of another object.*
  - *I*t supports the concept of hierarchical classification.
  - By use of inheritance, an object need only define those qualities that make it unique within its class.
  - It can inherit its general attributes from its parent.



- **Polymorphism**
  - It is a feature that allows one interface to be used for a general class of actions.
  - "one interface, multiple methods."
  - reduce complexity
  - same interface to be used to specify a *general class of action.*
  - It is the compiler's job to select the *specific action (that is, method) as it applies to each situation.*

iF ANY BODY SAYS "CUT" TO THESE PEOPLE



The surgeon would begin to make an incision.

The hair stylist would begin to cut someone's hair.

The actor would abruptly stop acting out the current scene, awaiting directorial guidance.

Introduction to Java Programming

First Simple Program

Entering the Program

- In Java, a source file is officially called a compilation unit.
- The java compiler requires that a source file use the .java filename extension.
- In Java, all code must reside inside a class and the name of that class should match the name of the file that holds the program.
- Java is case-sensitive

- Example.java

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
class Example {
   // Your program begins with a call to main().
   public static void main(String args[]) {
   System.out.println("This is a simple Java
   program.");
   }
}
```

- public static void main(String args[]){

➢ All Java applications begin execution by calling main( )

➢ The public keyword is an access specifier, which allows the programmer to control the visibility of class members.

➢ main( ) must be declared as public, since it must be called by code outside of its class when the program is started.

- The keyword static allows main( ) to be called without having to instantiate a particular instance of the class.
- This is necessary since main( ) is called by the Java interpreter before any objects are made.
- The keyword void tells the compiler that main( ) does not return a value.
- In main( ), there is only one parameter.
- String args[ ] declares a parameter named args, which is an array of instances of the class String.

- Objects of type String store character strings.
- args receives any command-line arguments present, when the program is executed.
- A complex program will have dozens of classes, anyone of which will need to have main( ) method to get things started.
- The next line of code is,
  - System.out.println("This is a simple Java program");
- Output is accomplished by the built-in println( ) method.

- println( ) displays the string which is passed to it.
- System is a predefined class that provides access to the system.
- out is the output stream that is connected to the console.
- All statements in Java end with a semicolon.
- The first } ends the main( ) and the last } ends the class definition.

- Compiling the Program

```
C:\>javac Example.java
```

- The **javac** compiler creates a file called Example.class that contains the bytecode version of the program.
- To actually run the program, you must use the Java interpreter, called **java.**

```
C:\>java Example
```

Output:

```
This is a simple Java program.
```

Example2.java.

```
/*
    Here is another short example.
    Call this file "Example2.java".
*/
class Example2 {
    public static void main(String args[]) {
            int num; // this declares a variable called num
            num = 100; // this assigns num the value 100
            System.out.println("This is num: " + num);
            num = num * 2;
            System.out.print("The value of num * 2 is ");
            System.out.println(num);
    }
}
```

Two Control Statements

- **The if Statement**

    if(*condition) statement;*

- If *condition is true, then the statement is* executed. If *condition is false, then the statement is bypassed.*

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

IfSample.java

```
class IfSample {
    public static void main(String args[]) {
            int x, y;
            x = 10;
            y = 20;
            if(x < y)
                        System.out.println("x is less than y");
            else
                        System.out.println(“y is less than x");

            x = x * 2;
            if(x == y)
            System.out.println("x now equal to y");
            x = x * 2;
            if(x > y)
            System.out.println("x now greater than y");
            // this won't display anything
            if(x == y)
            System.out.println("you won't see this");
    }
}
```

- **The for Loop**

  for(*initialization; condition; iteration)*

  statement;

```
class ForTest {
   public static void main(String args[]) {
        int x;
        for(x = 1; x<=10; x ++)
        System.out.println("This is x: " + x);
   }
}
```

- **<u>Using Blocks of Code</u>**
  - Java allows two or more statements to be grouped into *blocks of code, also called code blocks.*
  - This is done by enclosing the statements between opening and closing curly braces.

```
if(x < y) { // begin a block
        x = y;
        y = 0;
} // end of block
```

- **BlockTest.java**

```
Class BlockTest {
    public static void main(String args[]) {
            int x, y;
            y = 20;
            // the target of this loop is a block
            for(x = 0; x<10; x++) {
                        System.out.println("This is x: " + x);
                        System.out.println("This is y: " + y);
                        y = y - 2;
            }
    }
}
```

Data Types, Variables, and Arrays

- **Data Types**
  - Java is a strongly typed language.
  - Every variable has a type, every expression has a type, and every type is strictly defined.
  - The Java compiler checks all expressions and parameters to ensure that the types are compatible.
  - Any type mismatches are errors.

(*in C/C++ you can assign a floating-point value to an integer. In Java, you cannot.)*

The Simple Types

- Java defines eight simple (or elemental) types of data:
  - **byte, short, int, long, char, float, double, and boolean.**
- 4 Groups
  - Integers → includes **byte, short, int, and long**
  - Floating-point numbers → includes **float and double**
  - Characters → includes **char**
  - Boolean → includes **boolean**

- <u>Integers</u>
  - byte – signed 8-bit type
  - short – signed 16-bit type
  - int – signed 32-bit type
  - long – signed 64-bit type
- <u>Floating-Point types</u>
  - float – single-precision value, 32 bits of storage
  - double – double precision, 64 bits of storage
- <u>Characters</u>
  - 16-bit type (In C/C++ → 8 bits)
  - Java uses Unicode to represent characters.
- <u>Boolean</u>
  - logical values; true or false

Variables

- The variable is the basic unit of storage in a Java program.
- All variables have a scope, which defines their visibility, and a lifetime.
- **Declaring a Variable**

  type identifier [= value][, identifier [= value] ...] ;

  The *type is one of Java's atomic types, or the name of a class or interface.*

  The *identifier is the name of the* variable.

- **Dynamic Initialization**
  - Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

    ```
    double a = 3.0, b = 4.0;
    double c = Math.sqrt(a * a + b * b);
    ```
- **The Scope and Lifetime of Variables**
  - A block defines a *scope. Thus, each time you start a new block, you are creating* a new scope.
  - It also determines the lifetime of those objects.

- In Java, the two major scopes are those defined
  - by a class and
  - by a method
- Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- Scopes can be nested.
- Objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true.

```
class Scope {
    public static void main(String args[]) {
        int x; // known to all code within main
        x = 10;
        if(x == 10) { // start new scope
            int y = 20; // known only to this block
            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y not known here
        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

Type Conversion and Casting

- If the two types are compatible, then Java will perform the conversion automatically.
  - it is always possible to assign an **int value to a long variable.**
- Not all types are compatible, and thus, not all type conversions are implicitly allowed.
  - there is no conversion defined from **double to byte.**
- To obtain a conversion between incompatible types, you must use a *cast, which* performs an explicit conversion between incompatible types.

Java's Automatic Conversions(Widening Conversion)

- An *automatic type conversion will take place if the following two conditions are met:*
  - The two types are compatible.
  - The destination type is larger than the source type.
- For widening conversions,
  - the numeric types, including integer and floating-point types, are compatible with each other.
  - the numeric types are not compatible with **char or boolean.**
  - **char and boolean are not compatible with each other.**

Casting Incompatible Types (Narrowing Conversion)

- If an int value has to be assigned to a byte variable, the type conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a ***narrowing conversion**.*
- A ***cast*** is simply an explicit type conversion.
- It has this general form:

  (*target-type) value*

- *Here the target-type specifies the desired type to convert the specified value to.*

Eg)
```
int a;
byte b;
// ...
b = (byte) a;
```

- If the integer’s value is larger, than the range of a **byte, it will be reduced modulo byte’s range(256).**
- **Truncation**
  - when a floating-point value is assigned to an integer type, the fractional component is lost.
  - if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type’s range.

```
class Conversion {
    public static void main(String args[]) {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);
        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

Automatic Type Promotion in Expressions

- In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.
- Eg)

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

- a * b easily exceeds the range of either of its byte operands.
- Java automatically promotes each **byte or short operand to int when evaluating an expression.**

- They can cause confusing compile-time errors also.
- Eg) byte b = 20;

  ```
  b = b * 10; // Error! Cannot assign an int to a byte!
  ```

- In this case, the operands were automatically promoted to int and the result is promoted to int, which cannot be assigned to a byte without the use of a cast.
- In cases of such overflow, an explicit cast has to be used.
- Eg) byte b=50;

  ```
  b = (byte)(b * 2);
  ```

The Type Promotion Rules

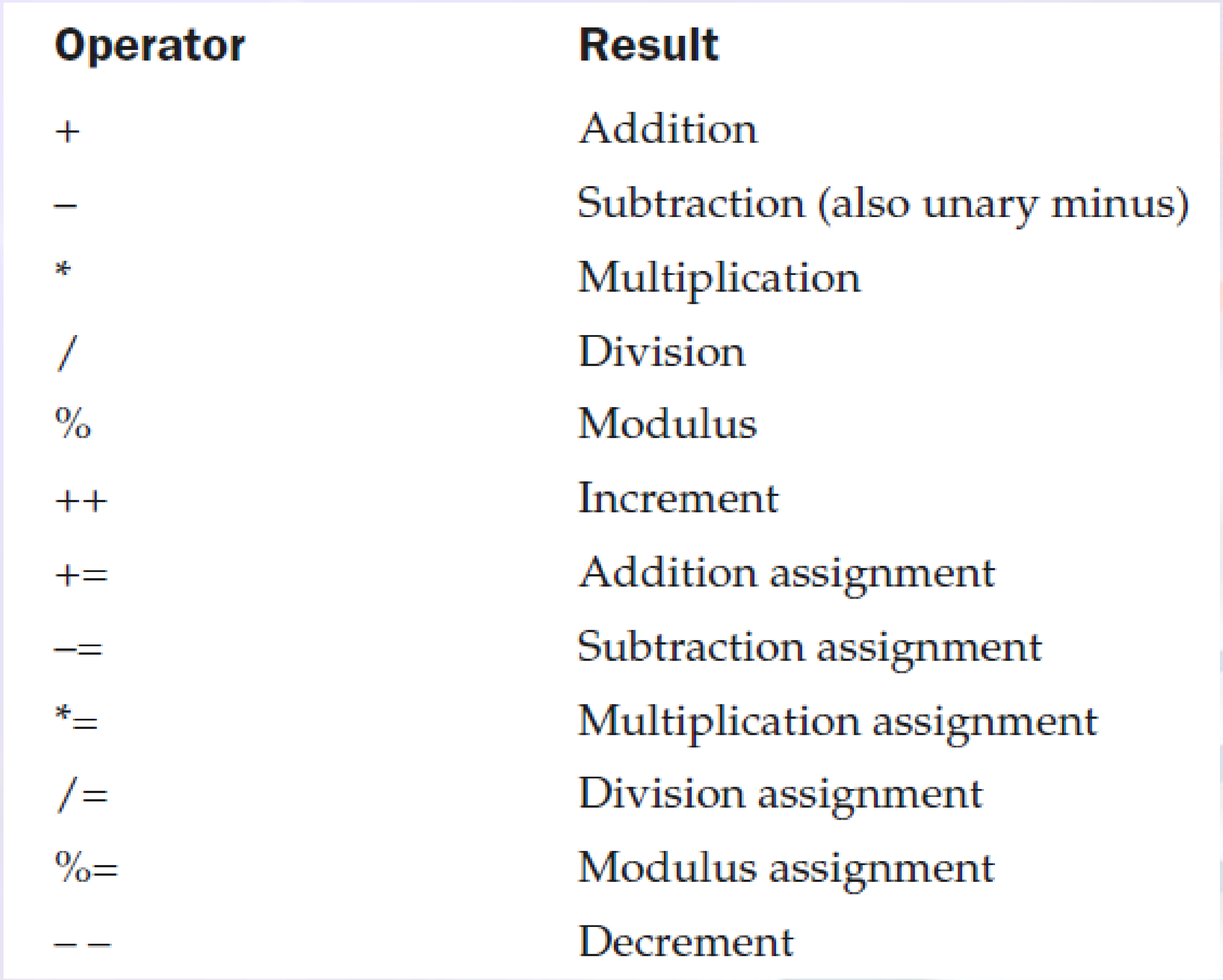
- Java defines several *type promotion rules that apply to expressions.*
  - All **byte and short values are** promoted to **int.**
  - if one operand is a **long, the whole expression** is promoted to **long.**
  - If one operand is a **float, the entire expression is promoted to float.**
  - If any of the operands is **double, the result is double.**

```
class Promote {
    public static void main(String args[]) {
            byte b = 42;
            char c = 'a';
            short s = 1024;
            int i = 50000;
            float f = 5.67f;
            double d = .1234;
            double result = (f * b) + (i / c) - (d * s);
            System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
            System.out.println("result = " + result);
    }
}
```

Operators

- Java provides a rich operator environment.
- Most of its operators can be divided into the following four groups:
  - arithmetic,
  - bitwise,
  - relational, and
  - logical.

1. Arithmetic Operators



Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- **The Basic Arithmetic Operators**
  - addition, subtraction, multiplication, and division
  - **+**, **-**, *****, **/**
- **The Modulus Operator**
  - The modulus operator, **%, returns the remainder of a division operation.**
- **Arithmetic Assignment Operators**

  a = a + 4;

  - In Java, the above stmt can be written as,

  a += 4;

  - Any statement of the form

  var = var op expression;

  Can be written as,

  var op= expression;

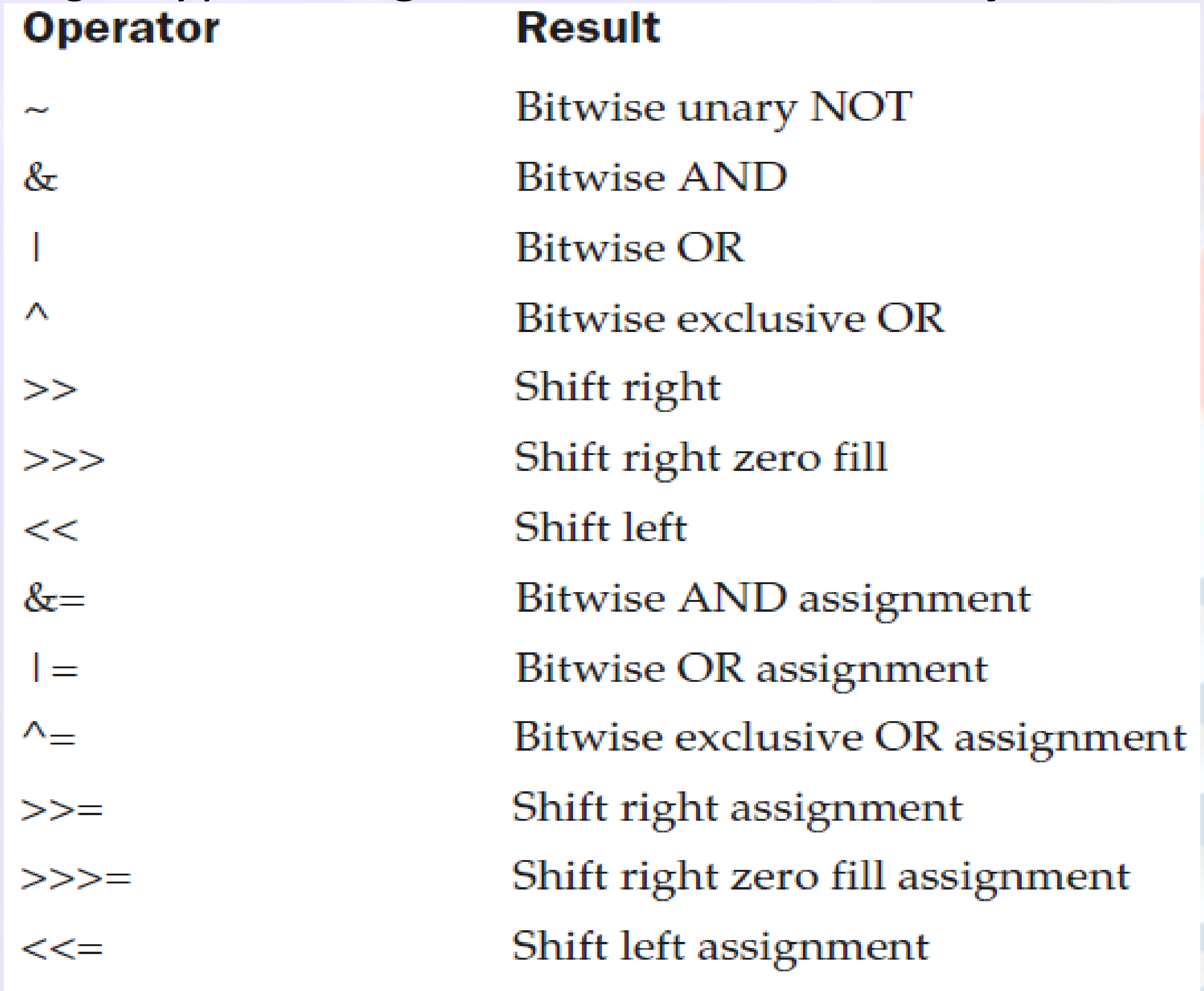
- **<u>Increment and Decrement Operators</u>**
  - **++, --**
  - The increment operator increases its operand by one.
  - The decrement operator decreases its operand by one.
  - <u>Prefix form</u>
    - the operand is incremented or decremented before the value is obtained for use in the expression.

      y=++x; → x=x+1; y=x;
  - <u>Postfix form</u>
    - the previous value is obtained for use in the expression, and then the operand is modified.

      y=x++; → y=x; x=x++;

2. The Bitwise Operators

- Java defines several *bitwise operators which can be applied to the integer types,* ***long,*** **int, short, char, and byte.**



Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

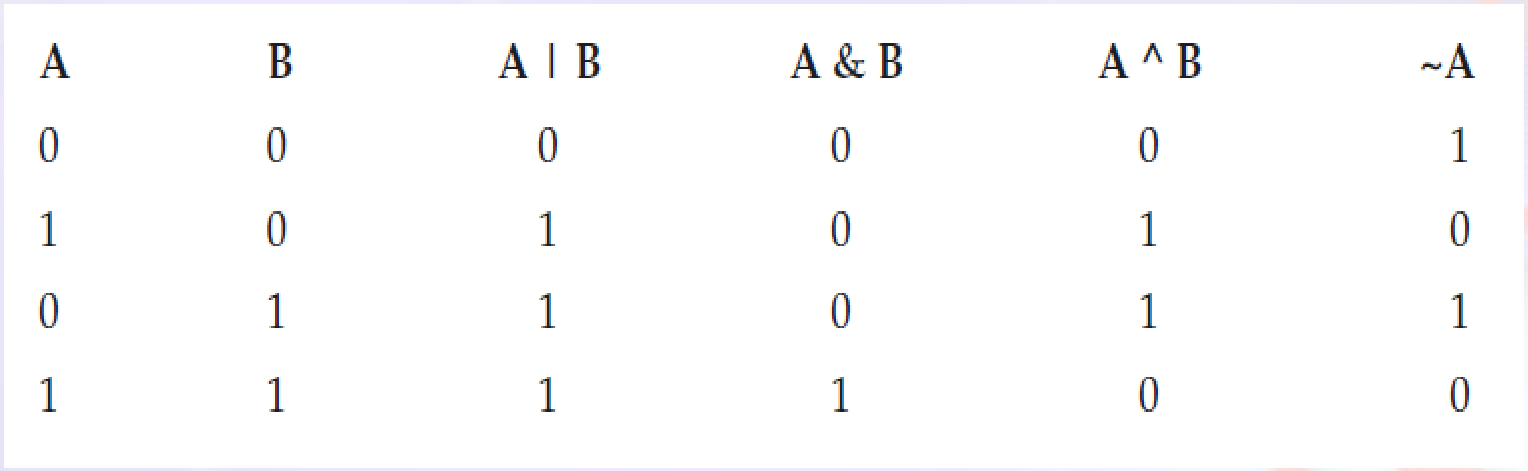
- Bitwise operators manipulate the bits within an integer.
- All of the integer types are represented by binary numbers of varying bit widths.
  - eg) the **byte value for 42 in binary is 00101010**
- All of the integer types (except **char) are signed integers.**
- This means that they can represent negative values as well as positive ones.
- Java uses an encoding known as *two's complement, which means that negative numbers are represented by inverting* (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.

    42 → 00101010

    -42 → 11010101 + 1 → 11010110

- The reason Java uses two’s complement is because of considering the issue of ZERO CROSSING
    - Assuming a **byte value, zero is** represented by 00000000. In one’s complement, simply inverting all of the bits creates11111111, which creates negative zero. The trouble is that negative zero is invalid in integer math. This problem is solved by using two’s complement to represent negative values. When using two’s complement, 1 is added to the complement, producing 100000000.
    - This produces a 1 bit too far to the left to fit back into the **byte value, resulting** in the desired behavior, where –0 is the same as 0, and 11111111 is the encoding for –1.

- **The Bitwise Logical Operators**
  - The bitwise logical operators are **&, |, ^, and ~**



A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

  - **Bitwise OR → |**
  - **Bitwise AND → &**
  - **Bitwise XOR → ^**
  - **Bitwise NOT → ~**

- **<u>The Left Shift (<<)</u>**
  - shifts all of the bits in a value to the left a specified number of times.

    value << num
  - For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
  - <u>Left shifting a byte value</u>

```
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
```

- Output

```
Original value of a: 64
i and b: 256   0
```

a= 64 (0100 0000) → promoted to int

→ and left shifted twice

result, i = 256 (…1 0000 0000) -> int value

→ casted to byte

b = 0 (0000 0000) -> higher order bits discarded

- **The Right Shift (>>)**
  - shifts all of the bits in a value to the right a specified number of times.

    value >> num
  - *eg)* int a = 32;

    a = a >> 2; // a now contains 8

    binary form

    0010 0011 (35)

    >>2

    0000 1000 (8)

- When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit.
- This is called *sign extension and serves* to preserve the sign of negative numbers when you shift them right.

  11111000 (–8)

  >>1

  11111100 (–4)
- Sometimes it is not desirable to sign-extend values when you are shifting them to the right.

- **The Unsigned Right Shift**
  - In case of working with pixel-based values and graphics, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift.*
  - To accomplish this, Java’s unsigned, shift-right operator, **>>>** is used, which always shifts zeros into the high-order bit.

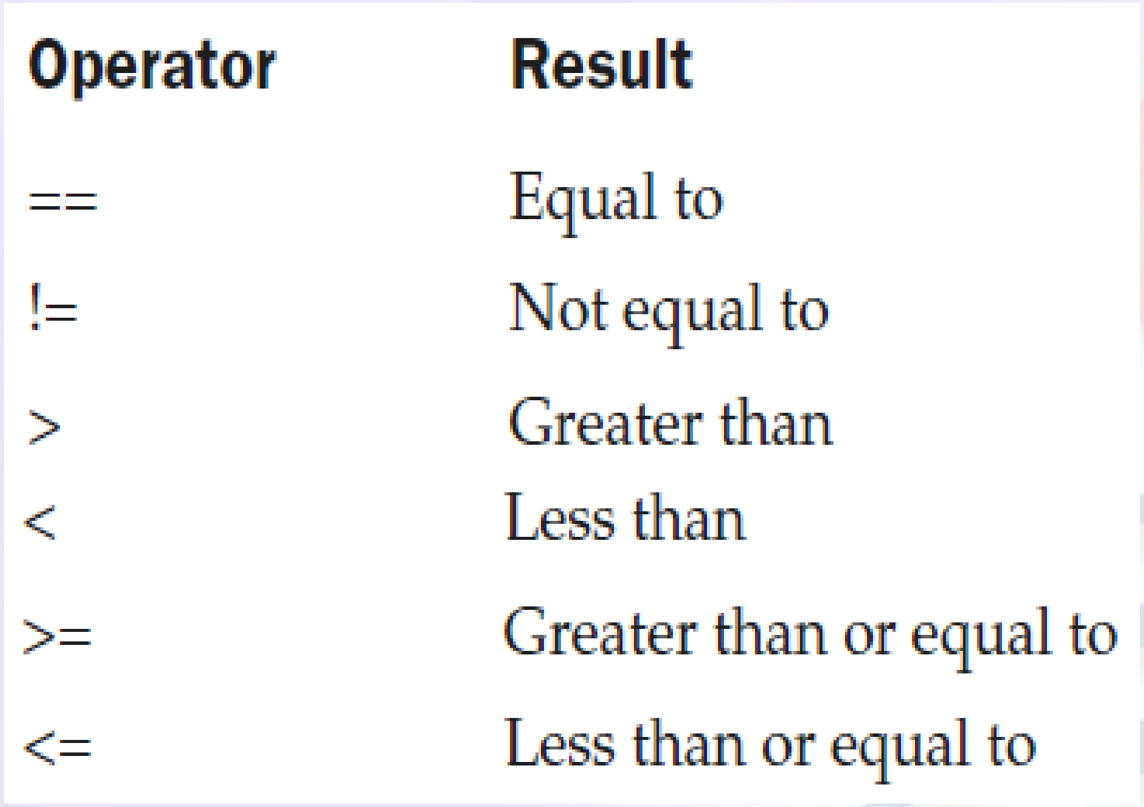
```
    eg) int a = -1;
             a = a >>> 24;
11111111 11111111 11111111 11111111 (–1)
              >>>24
00000000 00000000 00000000 11111111 (255)
```

- **Bitwise Operator Assignments**
  - All of the binary bitwise operators have a shorthand form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.
  - eg) a = a >> 4; → a >>= 4;

    a = a | b; → a |= b;

3.Relational Operators

- The *relational operators determine the relationship that one operand has to the other.*



Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- The outcome of these operations is a **boolean value.**

```
int a = 4;
int b = 1;
boolean c = a < b;
```

- In this case, the result of a<b (which is false) is stored in c.
- In C/C++, these types of statements are very common:

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ... // but not valid in Java.
```

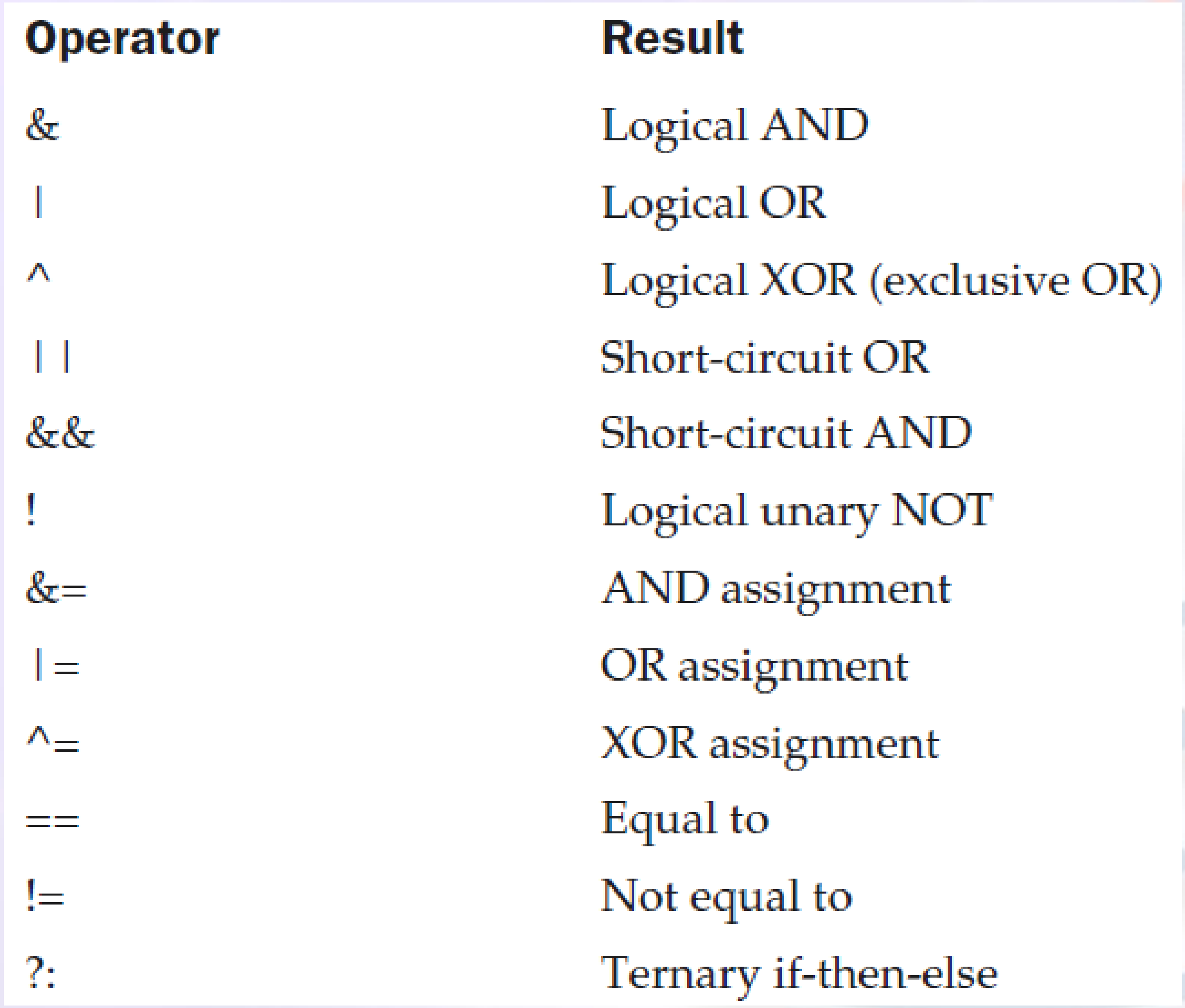
- In Java, these statements must be written like this:

```
if(done == 0)) ... // This is Java-style.
if(done != 0) ...
```

- In Java, **true and false are nonnumeric values** which do not relate to zero or nonzero.

4.Boolean Logical Operators

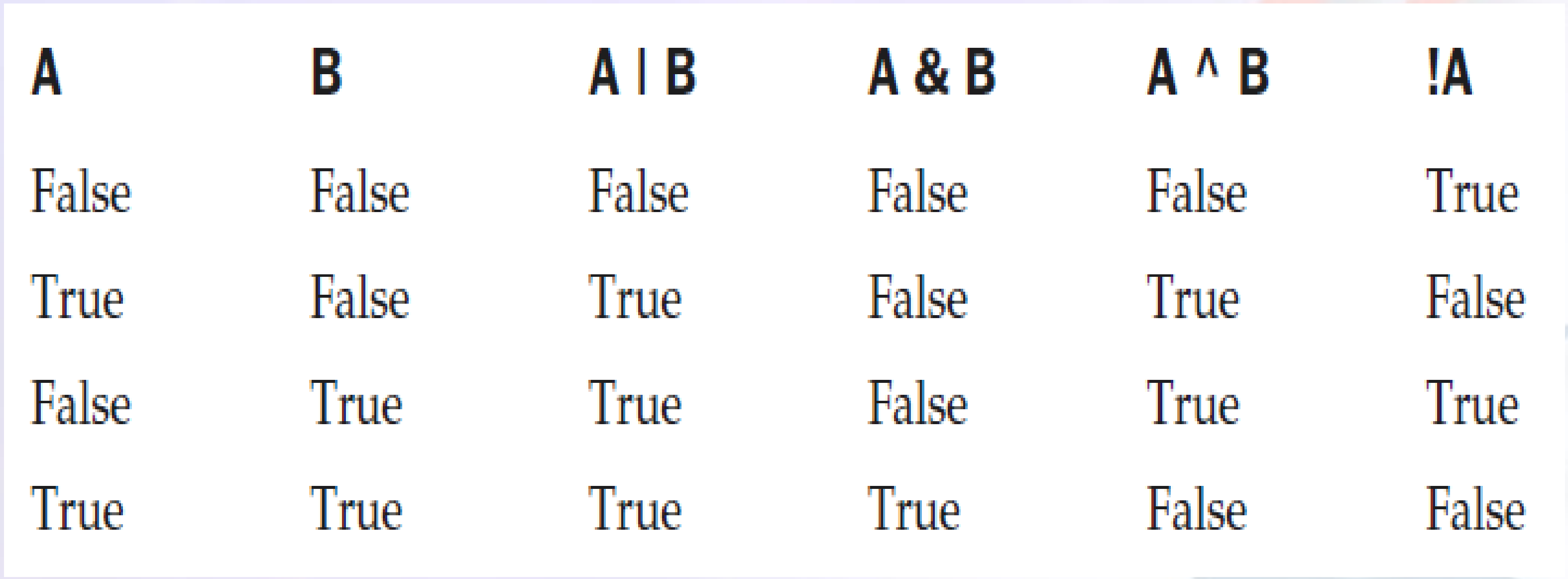
- The Boolean logical operators operate only on **boolean operands.**



Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

- The logical Boolean operators, **&, |, and ^,** operate on boolean values in the same way that they operate on the bits of an integer.
- The logical ! operator inverts the Boolean state:

  !true == false and !false == true.



A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

- **Short-Circuit Logical Operators**
  - Java provides two interesting Boolean operators, &&, ||, that are secondary versions of the Boolean AND and OR operators and are knows as *short-circuit logical operators.*
  - The OR operator results in **true when A is true, no matter what B is.**
  - Similarly, the AND operator results in **false when A is false, no matter what B is.**
  - If the short-circuit operators || and &&, rather than | and &, then Java will not evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

- Eg) if (denom != 0 && num / denom > 10)
  - Since the short-circuit form of AND (**&&) is used, there is no risk of causing a** run-time exception when **denom is zero.**
  - when using &&, the right hand operand is not evaluated.
  - If & is used, then the right hand operand will also be evaluated and that causes a run-time exception when denom==0.
- It is standard practice to use the short-circuit AND(&&) and OR(||) in cases involving boolean logic and the logical AND(&) and OR(|) exclusively for bitwise operations.

- **The Assignment Operator**
  - The *assignment operator is the single equal sign, =*

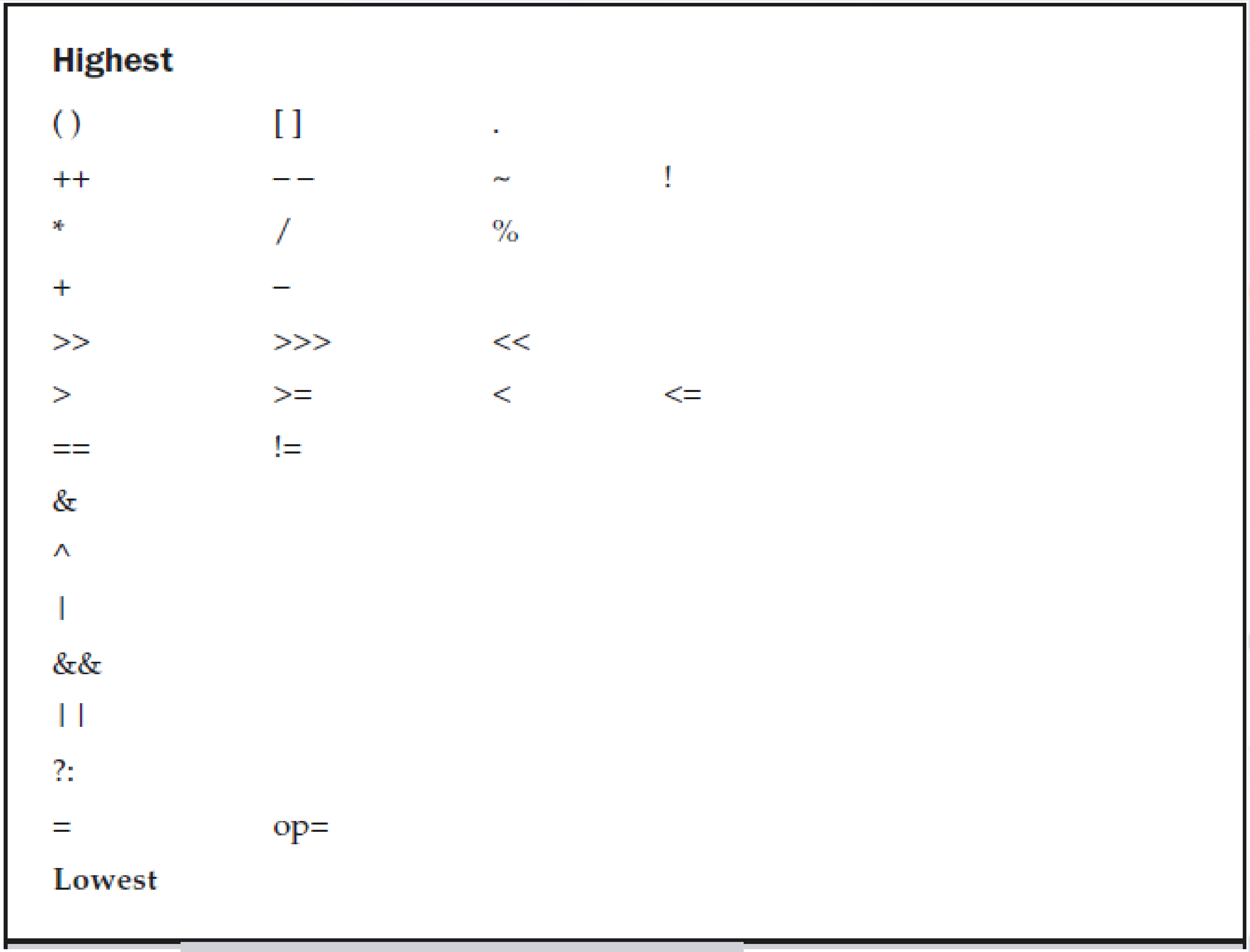
    var = expression;

  - The = is an operator that yields the value of the right-hand expression.
- **The ? Operator**
  - Java includes a special *ternary (three-way) operator that can replace certain types of* if-then-else statements.
  - General form

    expression1 ? expression2 : expression3

- Here, *expression1 can be any expression that evaluates to a boolean value***.**
- If ***expression1*** **is true, then** ***expression2 is evaluated; otherwise, expression3 is evaluated.***
- *Eg)* ratio = denom == 0 ? 0 : num / denom;
  - If denom equals zero, then the expression between the question mark (**?**) and colon (**:**) is evaluated and used as the value of the entire ? expression. (then ratio = 0)
  - If denom does not equal zero, then the expression after the colon is evaluated and ratio is assigned the resulting value.

- **Operator Precedence**



Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
| |			
?:			
=	op=		
Lowest			

The Precedence of the Java Operators

- Paranthesis are used to alter the precedence of an operation.

  eg) a >> (b+3)

  → first adds 3 to b and then shifts right by that result.

  (a >> b) + 3

  → first shift right by b positions and then add 3 to that result.

- The square brackets provide array indexing.
- The dot operator is used to dereference objects.

Control Statements

- Java's program control statements can be put into the following categories:
  - Selection
  - Iteration
  - Jump
- *Selection statements allow your program to choose different paths of execution* based upon the outcome of an expression or the state of a variable.
- *Iteration statements* enable program execution to repeat one or more statements.
- *Jump statements allow your program to execute in a nonlinear fashion.*

1. Java's Selection Statements

- Java supports two selection statements: **if and switch.**
- **if statement**

```
if (condition) statement1;
else statement2;
```

- Most often, the expression used to control the if will involve the relational operators.
- However, It is possible to control the if using a single boolean variable.

```
boolean dataAvailable;
// ...
if (dataAvailable)
    ProcessData();
else
     waitForMoreData();
```

- **Nested ifs**
  - A *nested if is an if statement that is the target of another if or else.*
  - When you nest *ifs*, an *else* statement always refers to the nearest *if* statement that is within the same block as the *else.*

```
if(i == 10)
 {
        if(j < 20)
             a = b;
        if(k > 100) // this if is
             c = d;
        else a = c; // associated with this else
 }
 else a = d; // this else refers to if(i == 10)
```

- **The if-else-if Ladder**
  - based upon a sequence of nested **ifs**

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
...
else
        statement;
```

  - The if statements are executed from the top down.
  - As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
  - If none of the conditions is true, then the final else statement will be executed.

- **switch statement**

```
switch (expression)
{
  case value1:
              // statement sequence
  break;
  case value2:
              // statement sequence
  break;
  ...
case valueN:
              // statement sequence
break;
default:
              // default statement sequence
}
```

- **Nested switch Statements**

```
switch(count)
{
        case 1:
        switch(target) // nested switch
        {
                case 0:
                System.out.println("target is zero");
                break;
                case 1: // no conflicts with outer switch
                System.out.println("target is one");
                break;
        }
        break;
        case 2: // ...
```

- 3 important features of **switch** statement
  - The switch differs from the ***if*** in that ***switch*** can only test for equality, whereas ***if*** can evaluate any type of Boolean expression.
  - No two case constants in the same ***switch*** can have identical values.
  - A ***switch*** statement is usually more efficient than a set of nested ***if***s.

2. Java's Iteration Statements

- **while statement**

```
while(condition)
{
// body of loop
}
```

  - The body of the loop will be executed as long as the conditional expression is true.
  - When *condition becomes false, control passes* to the next line of code immediately following the loop.

- **do-while statement**

```
do
{
// body of loop
} while (condition);
```

  - Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
  - If this expression is true, the loop will repeat.
  - Otherwise, the loop terminates.

- <u>**for statement**</u>

```
for(initialization; condition; iteration)
{
// body
}
```

  - When the loop first starts, the *initialization portion* of the loop is executed.
  - Next, *condition is evaluated. This* must be a Boolean expression. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
  - Next, the *iteration portion of the loop is executed.* The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass.
  - This process repeats until the controlling expression is false.

- <u>Declaring Loop Control Variables Inside the for Loop</u>
  - it is possible to declare the variable inside the initialization portion of the **for.**

    ```
    for(int n=10; n>0; n--)
    ```

  - When you declare a variable inside a **for loop,** the scope of that variable ends when the **for statement does.**
  - When the loop control variable will not be needed elsewhere, it can be declared inside the for.

- Using the Comma
  - if you want to include more than one statement in the initialization and iteration portions of the **for loop,** then the comma separator is used.

    for(a=1**, b=4; a<b; a++, b--)**

  - Each statement is separated from the next by a comma.

- **for Loop Variations**
  - The for loop supports a number of variations that increase its power and applicability.
  - One of the most common variations involves the conditional expression.
  - the condition controlling the **for can be any Boolean expression.**

```
boolean done = false;
for(int i=1; !done; i++) {
        // ...
        if(interrupted()) done = true;
}
```

  - In this example, the for loop continues to run until the boolean variable done is set to true.

- Another for loop variation is that, in a for loop, either the initialization or the iteration expression or both may be absent.

```
int i;
boolean done = false;
i = 0;
for( ; !done; ) {
        System.out.println("i is " + i);
        if(i == 10) done = true;
        i++;
}
```

- one more **for loop variation** is that, we can intentionally create an infinite loop by leaving all 3 parts of the for empty.

```
for( ; ; ) {
// ...
}
```

- This loop will run forever, and are called infinite loops with special termination requirements.

3.Java's Jump Statements

- Java supports three jump statements:
  - **break,**
  - **continue, and**
  - **return.**
- These statements transfer control to another part of your program.

  (another way in Java → Exception Handling)

- **Using break**
  - In Java, the **break statement has three uses.**
    - It terminates a statement sequence in a **switch statement.**
    - It can be used to exit a loop.
    - It can be used as a “civilized” form of goto.
  - Using break to Exit a Loop
    - By using **break, you can force immediate termination of a loop, bypassing the** conditional expression and any remaining code in the body of the loop.

    eg) if(i == 10) break; // terminate loop if i is 10

– <u>Using break as a Form of Goto</u>

- Java does not have a goto statement, because it provides a way to branch in an arbitrary and unstructured manner.
- The goto can be useful when you are exiting from a deeply nested set of loops.
- To handle such situations, Java defines an expanded form of the break statement , called labeled break statement by which you can break out of one or more blocks of code.
- General form

  break *label;*
- *label is the name of a label that identifies a block of code.*
- A *label is any valid Java identifier* followed by a colon.

- When this form of **break** executes, control is transferred out of the named block of code.
- The labeled block of code must enclose the break statement, but it does not need to be the immediately enclosing block.

```
class Break
{
    public static void main(String args[])
    {
            boolean t = true;
            first:
            {
                        second:
                        {
```

```
                    third:
                    {
                    System.out.println("Before the break.");
                    if(t) break second; // break out of second
                                        block
                    System.out.println("This won't execute");
                    }
               System.out.println("This won't execute");
               }
          System.out.println("This is after second block.");
          }
     }
}
```

- **Using continue**
  - To force an early iteration of a loop.
  - In **while and do-while** loops, a **continue statement causes control to be transferred directly to the conditional** expression that controls the loop.
  - As with the **break statement, continue may specify a label to describe which** enclosing loop to continue.

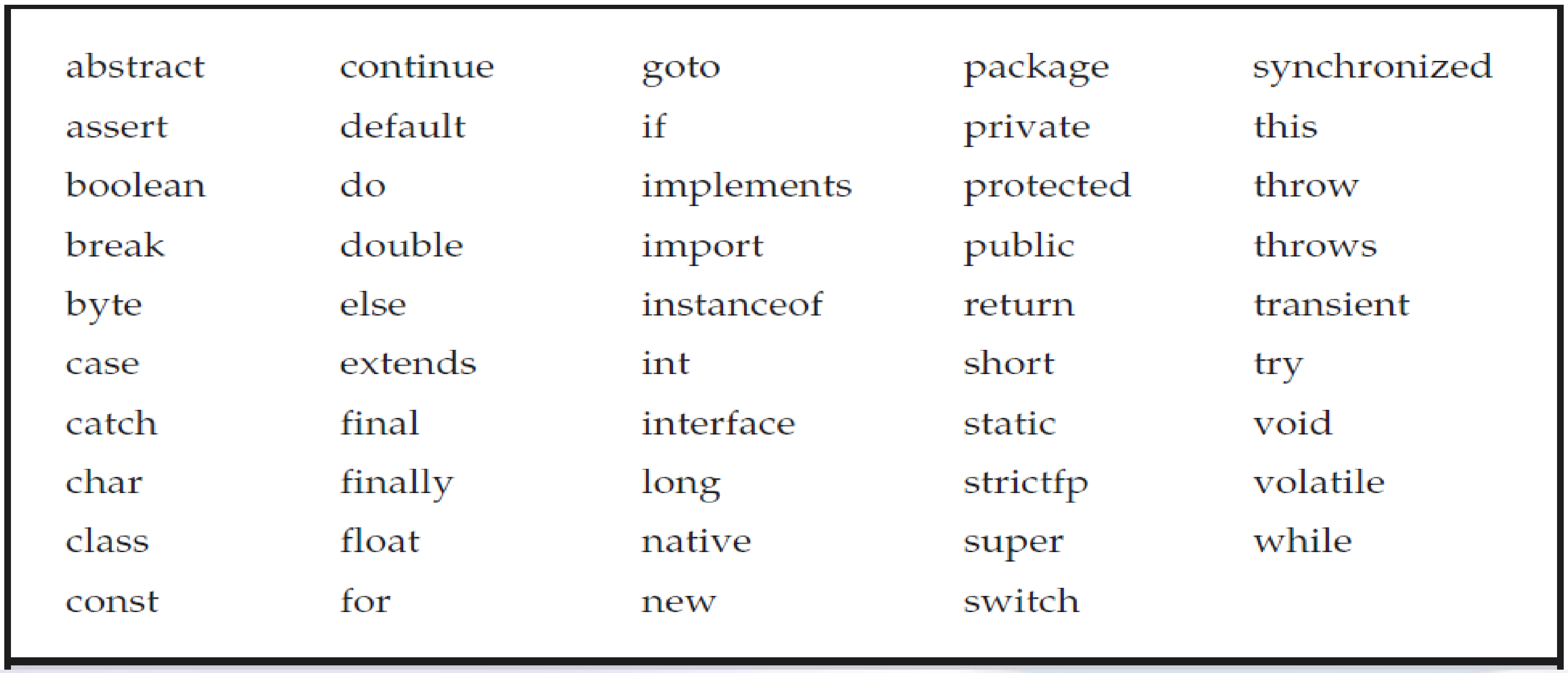
```
class ContinueLabel {
    public static void main(String args[]) {
    outer: for (int i=0; i<10; i++)
    {
            for(int j=0; j<10; j++)
            {
                    if(j > i)
                    {
                            System.out.println();
                            continue outer;
                    }
                    System.out.print(" " + (i * j));
            }
    }
    System.out.println();
    }
}
```

- <u>**return statement**</u>
  - The **return statement is used to explicitly return** from a method.
  - It causes program control to transfer back to the caller of the method.
  - The **return statement immediately terminates** the method in which it is executed.

```
class Return
{
    public static void main(String args[])
    {
            boolean t = true;
            System.out.println("Before the return.");
            if(t) return; // return to caller
            System.out.println("This won't execute.");
    }
}
```

The Java Keywords

- There are 49 reserved keywords currently defined in the Java language.
- These keywords cannot be used as names for a variable, class, or method.



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

The Java Class Libraries

- The java built-in methods println() and print() are members of the System class, which is a class predefined by java that is automatically included in your programs.
- The Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics.
- Java as a totality is a combination of the Java language itself, plus its standard classes.

Arrays

- An array is a group of like-typed variables that are referred to by a common name.
- A specific element in an array is accessed by its index.
- **One-Dimensional Arrays**
  - A one-dimensional array is, essentially, a list of like-typed variables.
  - General form

    type var-name[ ];
  - type declares the base type of the array.
  - The base type determines the data type of each element that comprises the array.
  - Eg) int month_days[];

- The value of month_days is set to null, which represents an array with no value.
- To link month_days with an actual, physical array of integers**,** you must allocate one using **new** and assign it to month_days.
- **new is a special operator** that allocates memory.
- General form

  array-var = new type[size];
- To use **new** to allocate an array, you must specify the type and number of elements to allocate.
- Eg) month_days=new int[12];

- Obtaining an array is a two-step process.
  - Declare a variable of the desired array type.
  - allocate the memory that will hold the array, using **new, and assign it to the array variable.**
- In Java all arrays are dynamically allocated.
- Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets.

```
month_days[1] = 28;
```

- All array indexes start at zero.

```
class Array {
    public static void main(String args[]) {
            String colors[];
            colors=new String[7];
            colors[0]="violet";
            colors[1]="Indigo";
            colors[2]="Blue";
            colors[3]="Green";
            colors[4]="Yellow";
            colors[5]="Orange";
            colors[6]="Red";
            for(int i=0;i<7;i++){
                        System.out.println("Rainbow Colours:"+colors[i]);
            }
    }
}
```

- Arrays can be initialized when they are declared.
- An *array initializer is a list of comma-separated* expressions surrounded by curly braces.
- The commas separate the values of the array elements.
- Eg)

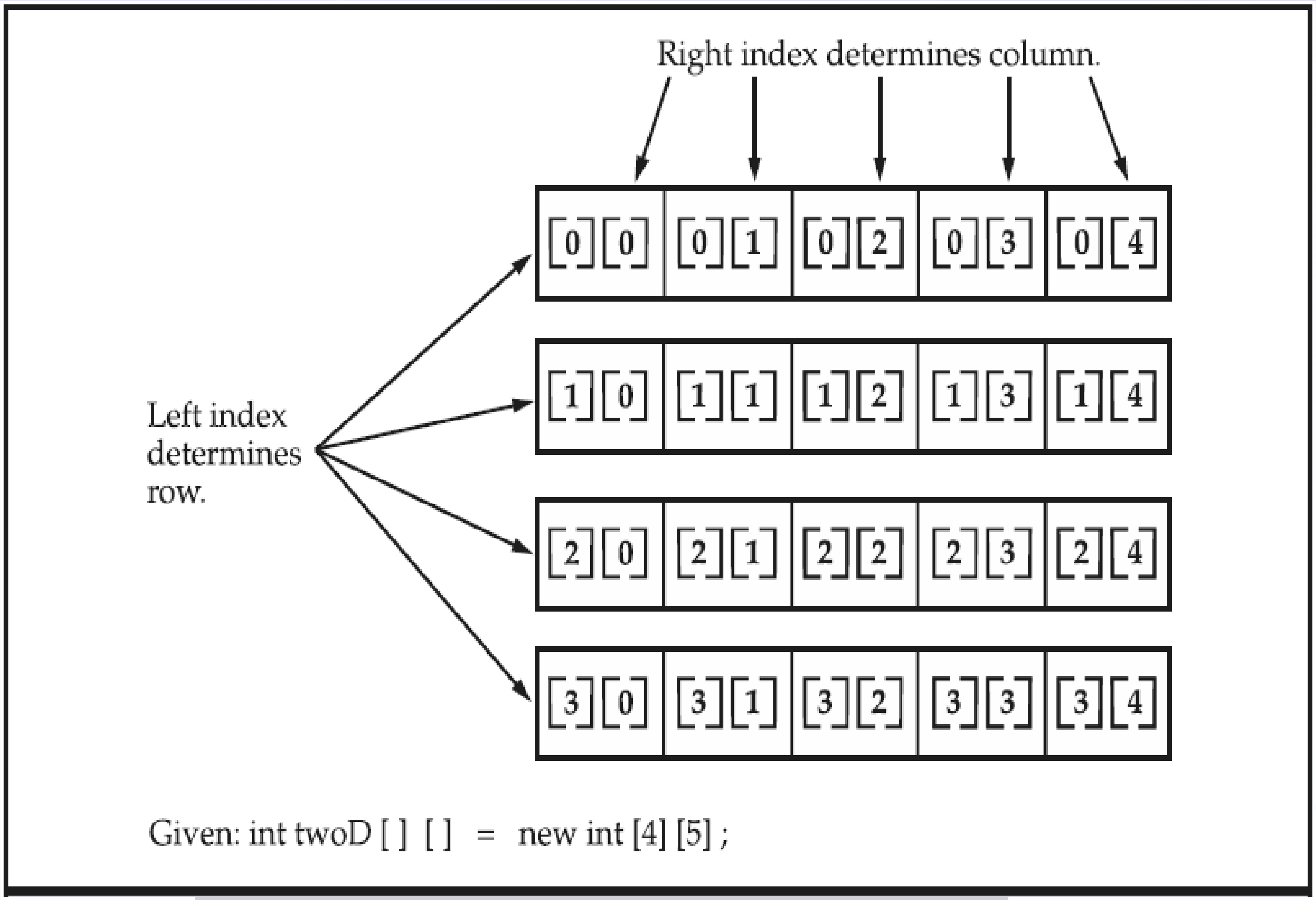
```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

- The Java run-time system will check to be sure that all array indexes are in the correct range.
- If elements outside the range of an array were tried to be accessed, then a run-time error will be caused.
- C/C++ provide no run-time boundary checks.

Multidimensional Arrays

- In Java, *multidimensional arrays are actually arrays of arrays.*
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.

```
int twoD[][] = new int[4][5];
```



A conceptual view of a 4 by 5, two-dimensional array