OBJECT ORIENTED PROGRAMMING USING JAVA

By

Er. C Rout

Er.M Mohanty

Assistant Professor

Computer Science and Engineering

Introduction To Programming

What is a program?

A computer program is a collection of instructions that can be executed by a computer to perform a specific task. A computer program is usually written by a computer programmer in a programming language.

Programming language

A programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms. Most programming languages consist of instructions for computers.

Types of programming languages

There are two **types of programming languages**, which can be categorized into the following ways

1. Low level language

a) Machine language (1GL)

b) Assembly language (2GL)

2. High level language

a) Procedural-Oriented language (3GL)

b) Problem-Oriented language (4GL)

c) Natural language (5GL)

Low level language

a) Machine Language (1GL): Machine language consists of strings of binary numbers (i.e. 0s and 1s) which the processor directly understands. Machine language has the merits of very fast execution speed and efficient use of primary memory.

Merits

- It is directly understood by the processor. So it has faster execution time since the programs written in this language need not to be translated. It doesn't need larger memory.

Demerits

- It is a very difficult task writing program in machine level language since all the instructions are to be represented by 0s and 1s.
- Use of this machine level language makes programming time consuming.
- It is difficult to find an error and to debug them in machine level language.
- It can be used by experts only.

Low level language

b) Assembly Language: Assembly language is also known as low-level language because to design a program, the programmer requires detailed knowledge of hardware specification. This language uses mnemonics code such as (‘ADD’ for addition, ’STORE’ for keeping data etc) in place of 0s and 1s. The program is converted into machine code by assembler. The resulting program is referred to as an object code.

Merits

- It makes programming easier than machine level language since it uses mnemonics code for programming. Eg: ADD for addition, SUB for subtraction, DIV for division, etc.
- It makes the programming process faster.
- Error can be identified much easily as compared to machine level language.
- It is easier to debug than machine language.

Demerits

- Programs written in this language is not directly understood by the computer. So a translator is required which will translate from assembly language to object code.
- It is the hardware dependent language so programmers are forced to think in terms of computer’s architecture rather than the problem being solved.
- Since it is the machine dependent language, programs written in this language are very less or not portable.
- Programmers must know its mnemonics codes to perform any task.

High level language

a) Procedural-Oriented language (3GL): Procedural Programming is a methodology for modelling the problem being solved by determining the steps and the order of those steps that must be followed in order to reach a desired outcome or specific program state. These languages are designed to express the logic and the procedure of a problem to be solved. It includes languages such as Pascal, COBOL, C, FORTAN, etc.

Merits

- Because of its flexibility, procedural language is able to solve a variety of problems. Programmer does not need to think in term of computer architecture which makes them focused on the problem.
- Programs written in this language are portable.

Demerits

- It is easier but needs higher processor and larger memory. It needs to be translated. So its execution time is more.

High level language

b) Problem-Oriented language (4GL): It allows the users to specify what the output should be, without describing all the details of how the data should be manipulated to produce the result. This is one step ahead from 3GL. These are result oriented and include database query language, eg: Visual Basic, C#, PHP, etc.

The objectives of 4GL

- are to increase the speed of developing programs.
- Minimize user's effort to obtain information from computer.
- Reduce errors while writing programs.

Merits

- Programmer need not to think about the procedure of the program. So, programming is much easier.

Demerits

- It is easier but needs higher processor and larger memory.
- It needs to be translated. So its execution time is more.

High level language

c) Natural language (5GL): Natural language are still in developing stage where we could write statements that would look like normal sentences.

Merits

- It is easy to program. Since, the program uses normal sentences, they are easy to understand. The programs designed using 5GL will have artificial intelligence (AI).
- The programs would be much more interactive and interesting.

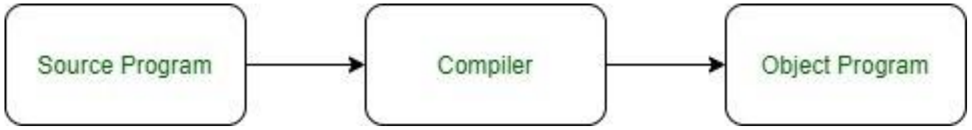
Demerits

- It is slower than previous generation language as it should be completely translated into binary code which is a tedious task. Highly advanced and expensive electronic devices are required to run programs developed in 5GL. Therefore, it is an expensive approach.

Compiler and Interpreter

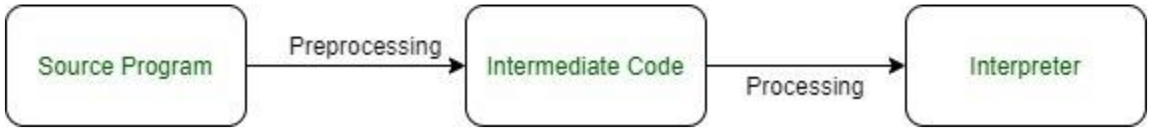
Compiler

- It is a translator which takes input from high Level Language and produces an output in machine level or assembly language.
- Compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But it’s program run time is more and occupies a larger part of memory. Its speed is slow because a compiler goes through the entire program and then translates the entire program into machine codes.



Interpreter

- An interpreter is a program which translates a programming language into a comprehensible language.
- It translates only one statement of the program at a time.
- Interpreters, more often are smaller than compilers.



Introduction to JAVA

- Java is first and foremost a programming language, has become inseparably linked with the online environment of the Internet
- Java is related to C++, which is a direct descendent of C. Much of the character of Java is inherited from these two languages.
- This language was initially called “Oak” but was renamed “Java” in 1995
- Java is not a replace for C++. Both will co-exist for many years to come.
- Motivation for creation of Java- Need for a platform independent language.
- Java is a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.
- Java is forefront language for- Creating efficient, portable programs for World Wide Web
- Designed to support applications on a network (Distributed computing).

Java and the Internet

- Java simplified web programming by using a new type of networked program called applet
- Java also addressed some of the issues associated with the Internet- *portability and security*.

Java Applets:

- An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser
- Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, that execute locally, rather than on the server.
- The applet allows some functionality to be moved from the server to the client.

Security

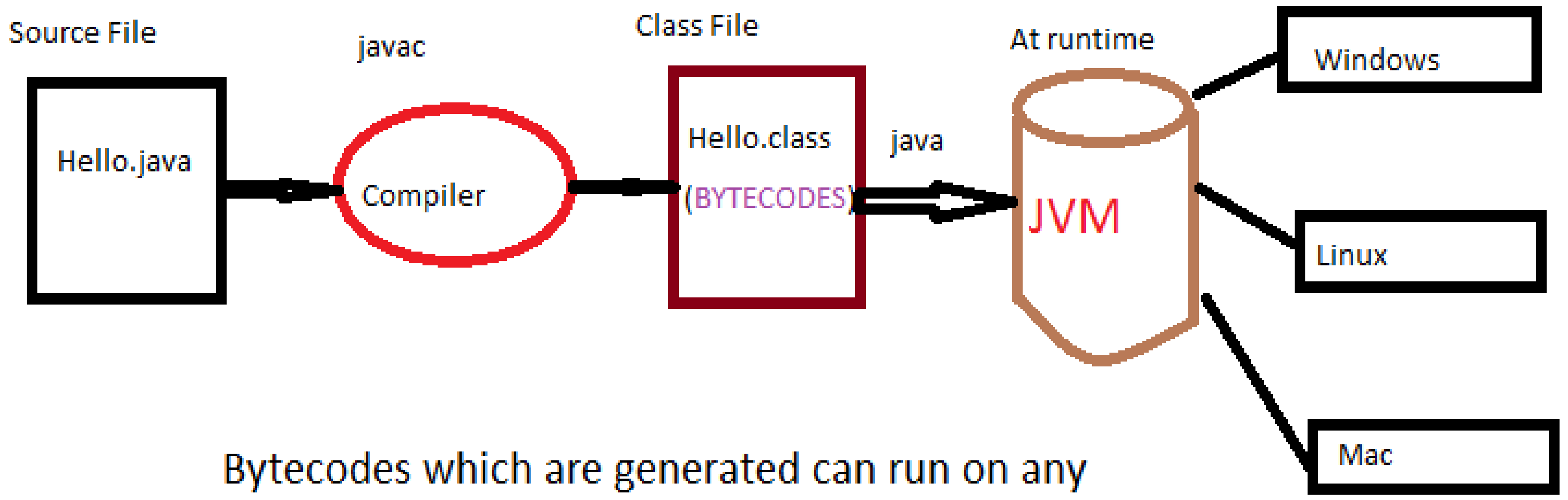
- When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent (applets downloaded and executed on the client computer safely)
- Java achieves this protection by confining an applet to the Java execution environment and not allowing access to other parts of the computer
- Security is not breached, hence Java is considered by many to be the single most innovative aspect of Java.

Portability

- Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- Java is portable across many types of computers and operating systems that are in use throughout the world

Java 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*).
- Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments.
- When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code
- Optimized bytecode enables the Java run-time system to execute programs much faster than we expect.
- Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release.
- When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis
- The JIT compiles code as it is needed, during execution



Bytecodes which are generated can run on any machine which has the JVM and are secure because JVM itself checks the code at runtime.

The Java Buzzwords (Features of Java)

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

- If you already understand the basic concepts of object-oriented programming, learning Java will be even easier
- Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java
- Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have *surprising* features

Object oriented

- The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects

Robust

- Ability to create robust programs was given a high priority in the design of Java
- To better understand how Java is robust, consider two of the main reasons for program failure: ***memory management mistakes*** and ***mishandled exceptional conditions*** (that is, run-time errors)
- Memory management can be a difficult, tedious task in traditional programming environments
- For example, in C/C++, the programmer must manually allocate and free all dynamic memory. Programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using
- Java virtually eliminates these problems by managing memory allocation and deallocation for you
- Java provides object-oriented exception handling.

Multithreaded

- Java supports multithreaded programming, which allows you to write programs that do many things simultaneously
- The Java run-time system comes with an elegant yet sophisticated solution for multi-process synchronization that enables you to construct smoothly running interactive systems

Architecture-Neutral

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction
- Java Virtual Machine in an attempt to alter this situation
- Their goal was "write once; run anywhere, any time, forever."

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode

- This code can be interpreted on any system that provides a Java Virtual Machine

- the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler

Distributed

- Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols
- *Remote Method Invocation* (*RMI*) feature of Java brings an unparalleled level of abstraction to client/server programming

Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.

AN OVERVIEW OF JAVA

- Object-oriented programming (OOP) is the basic concept of Java programming.
- All computer programs consist of two elements: ***code*** and ***data.*** A program can be conceptually organized around its code or around its data

Two Paradigms

Process-oriented model:

The process-oriented model can be thought of as *code acting on data*

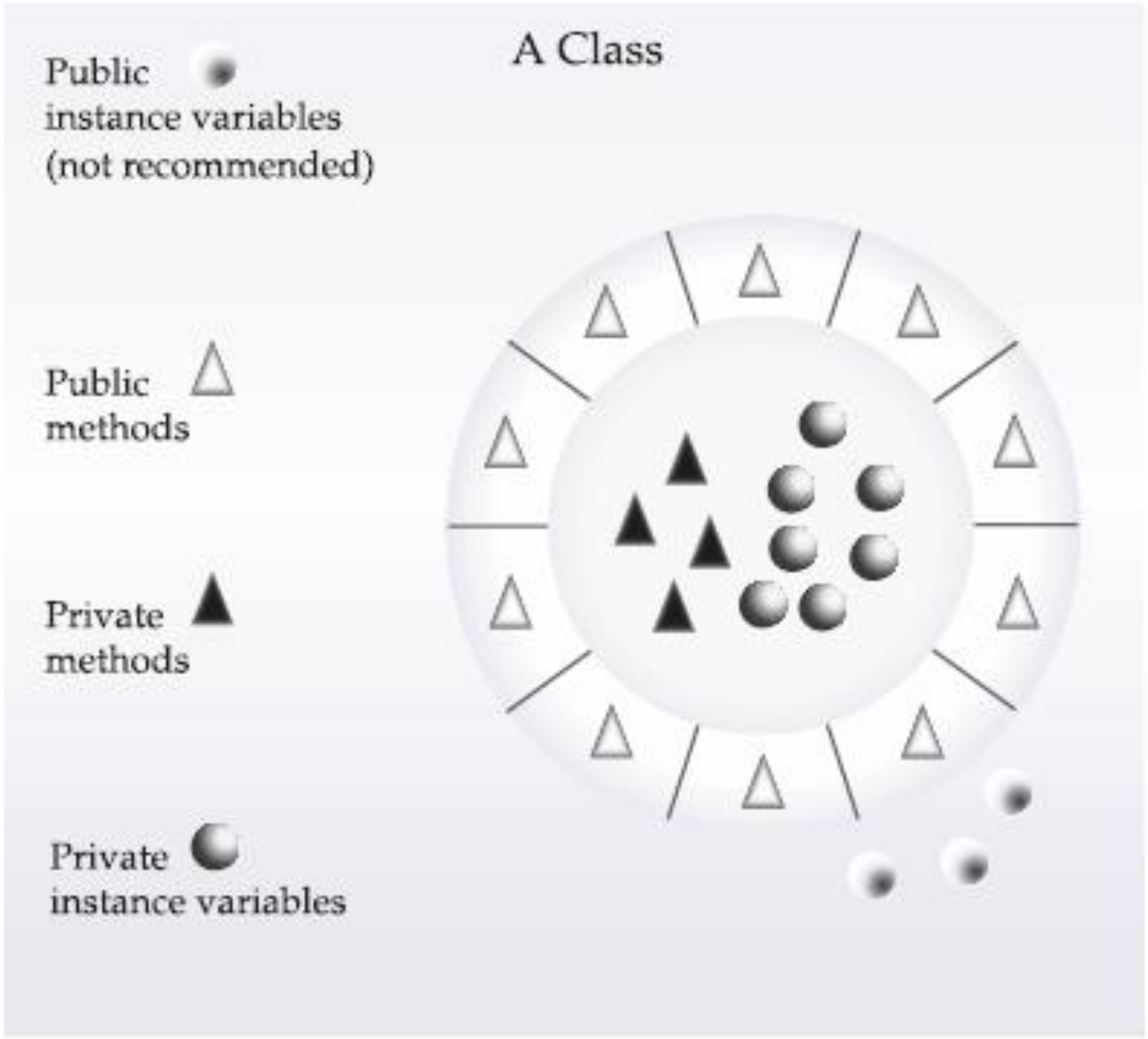
Object-oriented programming:

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data

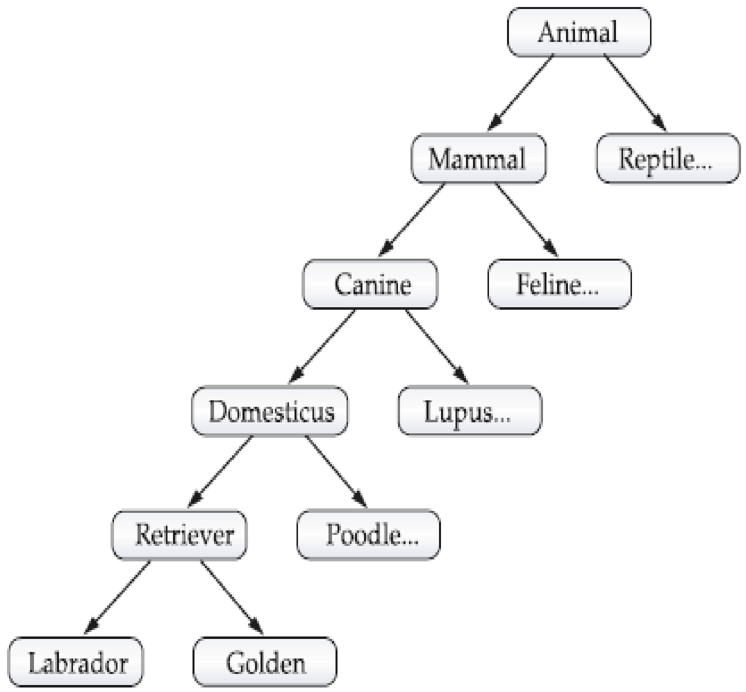
The Three OOP Principles: *encapsulation*, *inheritance*, *and polymorphism*.

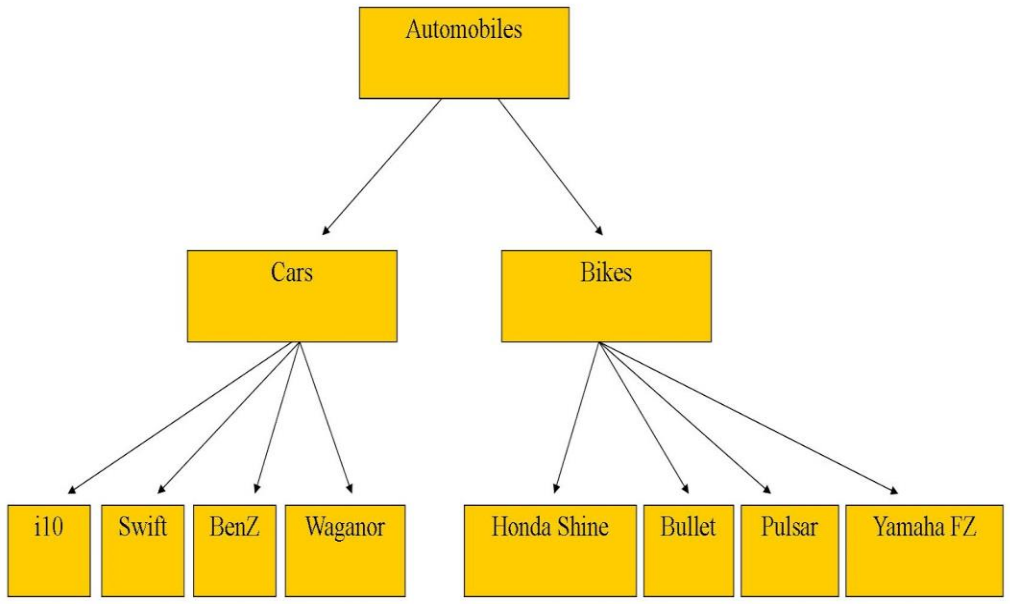
- ***Encapsulation*** - is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse
- In Java, the basis of encapsulation is the class. class defines the structure and behavior (data and code) that will be shared by a set of objects. objects are sometimes referred to as instances of a class.

Keywords: class, objects, member variables, member methods, private, public


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

- ***Inheritance*** - the process by which one object acquires the properties of another object. Important because it supports the concept of hierarchical classification (top down).





- ***Polymorphism*** – (meaning many forms) -- is a feature that allows one interface to be used for a general class of actions
- The specific action is determined by the exact nature of the situation.

Example- STACK operations

- the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.*

A First Simple Program

```
/* 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.");
}
}
```

When above program is run, the following output is displayed:

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

Compiling the Program

- ***C:\>javac Example.java***
- The javac compiler creates a file called ***Example.class*** that contains the bytecode version of the program
- Java bytecode is the intermediate representation of the program that contains instructions the Java Virtual Machine will execute.The output of **javac** is not the code that can be directly executed
- To actually run the program, you must use the Java application launcher, called **java (**interpreter).

- ***C:\>java Example***
- When a class member is preceded by **public**, then that member may be accessed by the code outside the class in which it is declared
- The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class
- The keyword **void** simply tells the compiler that **main( )** does not return a value

A Second Program

```
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 =“ + num);
}
}
```

When above program is run, the following output is displayed:

```
The value of num * 2 is = 200
```

Lexical Issues

- ***Whitespace:***
  - Java is a free-form language
  - In Java, whitespace is a space, tab, or newline
- ***Identifiers:***

  - Identifiers are symbols used to uniquely identify a program element in the code.

  - Identifiers are used for class names, method names, and variable names

  - Java is case-sensitive

Ex: sum, sum10, student etc.

Literals

- a **literal** is a notation for representing a fixed value in source code.
- A constant value in Java is created by using a *literal* representation of it
- A literal may be of any type such as: integer, floating point, character, Boolean etc.

Integer Literals

- Decimal, Hexadecimal and Octal
- When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type.
- An integer literal can always be assigned to a long variable. To specify a long, we need to explicitly tell the compiler that the literal value is of type long, by appending an upper- or lowercase L to the literal.
- Example: 10, 205, 100 etc.

Floating-Point Literals

- For example, 2.0, 3.14159, and 0.6667 represent valid standard -notation floating-point numbers
- *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied
- Example: 6.022E23, 314159E–05, and 2e+100
- Floating-point literals in Java default to *double* precision. To specify a float literal, you must append an *F or f* to the constant. You can also explicitly specify a double literal by appending a *D or d.*

Boolean Literals

- Used for truth value i.e true (1) or false (0)

Character Literals

- Characters in Java are indices into the Unicode character set
- They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators
- A literal character is represented inside a pair of single quotes
- Ex: ‘a’, ‘b’ , ‘T’ etc.

String Literals: String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.

Example:

“Hello World”

“two\nlines”

“\”This is in quotes\”“

Comments

- There are three types of comments defined by Java.

- Single-line
- multiline

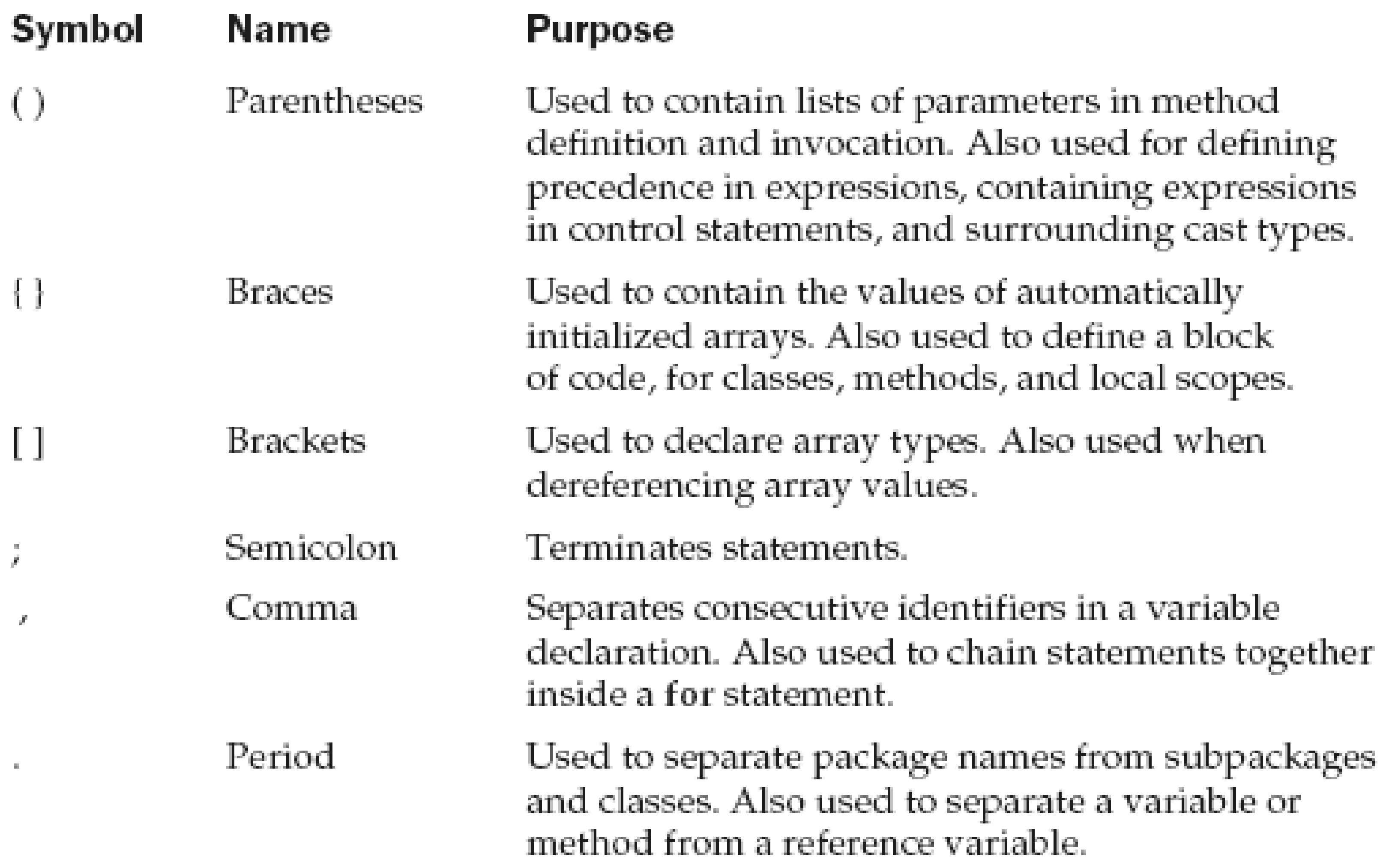
- The third type is called a *documentation comment*

- This type of comment is used to produce an HTML file that documents your program

- The documentation comment begins with **/**** and ends with ***/.**

- The **JDK javadoc** tool uses *doc comments* when preparing automatically generated documentation.

Separators



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

The Java Keywords

- There are 50 reserved keywords currently defined in the Java language

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	

DATA TYPES, VARIABLES, AND ARRAYS

Java Is a Strongly Typed Language

- Three of Java’s fundamental elements: data types, variables, arrays
- Every variable has a type, every expression has a type, and every type is strictly defined
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility
- The Java compiler checks all expressions and parameters to ensure that the types are compatible

The Primitive Types

Java defines eight primitive (or elemental) types of data:

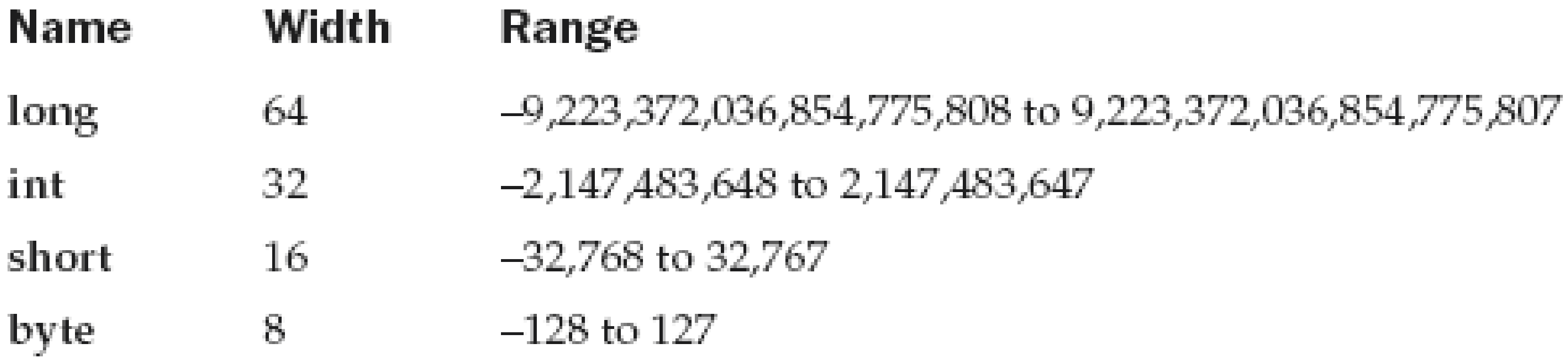
byte, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**

These can be put in four groups:

- **Integers:** This group includes **byte, short, int, and long,** which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes **float and double,** which represent numbers with fractional precision.
- **Characters :** This group includes **char,** which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes **boolean,** which is a special type for representing true/false values.
- The primitive types are defined to have an explicit range and mathematical behavior.
- Because of Java’s portability requirement, all data types have a strictly defined range.

Integers

- Java defines four integer types: **byte, short, int, and long.**
- All of these are signed, positive and negative values.
- Java does not support unsigned, positive-only integers.



Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

```
class Light {
    public static void main(String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;
        // approximate speed of light in miles per second
        lightspeed = 186000;
        days = 1000; // specify number of days here
        seconds = days * 24 * 60 * 60; // convert to seconds
        distance = lightspeed * seconds; // compute distance
        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

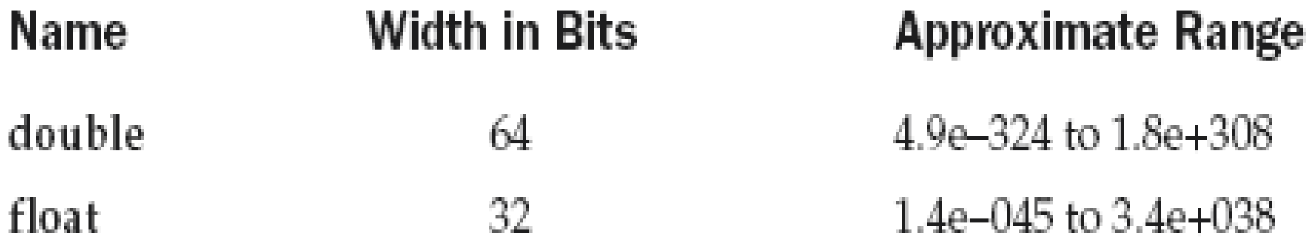
Clearly, the result could not have been held in an **int variable.**

Floating-Point Types

Floating-point numbers, also known as *real numbers, are used when evaluating expressions* that require fractional precision.

Java implements the standard (IEEE–754) set of floating-point types and operators.

There are two kinds of floating-point types, **float and double,** which represent single- and double-precision numbers, respectively.



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Example:

```
class Area {
    public static void main(String args[ ]) {
    double pi, r, a;
    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area
    System.out.println("Area of circle is " + a);
    }
}
```

Characters

- **char** in Java is not the same as **char** in C or C++.
- In C/C++, **char** is an integer type that is 8 bits wide
- Instead, Java uses Unicode to represent characters
- *Unicode* defines a fully international character set that can represent all of the characters found in all human languages
- In Java **char** is a 16-bit type, The range of a **char** is 0 to 65,536
- There are no negative **char**s
- Since Java is designed to allow programs to be written for worldwide use, it uses Unicode to represent characters.

Example 1:

```
class CharDemo1 {
  public static void main(String args[ ]) {
  char ch1, ch2;
  ch1 = 88; // code for X
  ch2 = 'Y';
  System.out.print("ch1 and ch2: ");
  System.out.println(ch1 + " " + ch2);
  }
}
```

Above program displays the following output:

```
ch1 and ch2: X Y
```

Example 2:

```
class CharDemo2 {
   public static void main(String args[ ]) {
   char ch1;
   ch1 = 'X';
   System.out.println("ch1 contains " + ch1);
   ch1++; // increment ch1
   System.out.println("ch1 is now " + ch1);
   }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

Booleans

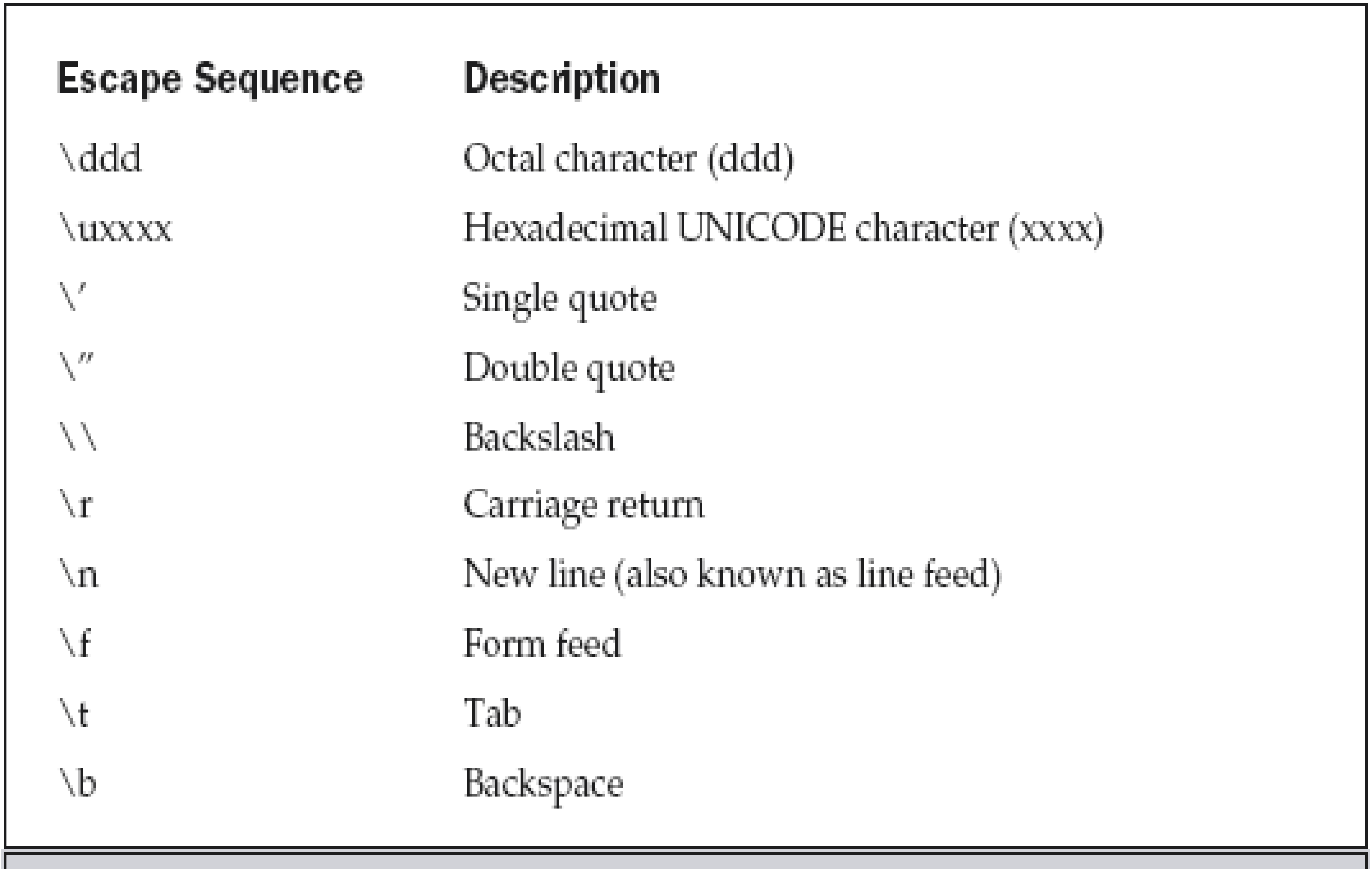
- Java has a primitive type, called **boolean,** for logical values. Can have only one of two possible values, **true** or **false**
- This is the type returned by all relational operators, as in the case of **a < b.**
- **boolean** is also the type required by the conditional expressions that govern the control statements such as if and for.

```
class BoolTest {
    public static void main(String args[]) {
            boolean b;
            b = false;
            System.out.println("b is " + b);
            b = true;
            System.out.println("b is " + b);
    // a boolean value can control the if statement
            if(b) System.out.println("This is executed.");
            b = false;
            if(b) System.out.println("This is not executed.");
    // outcome of a relational operator is a boolean value
            System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Character escape sequences



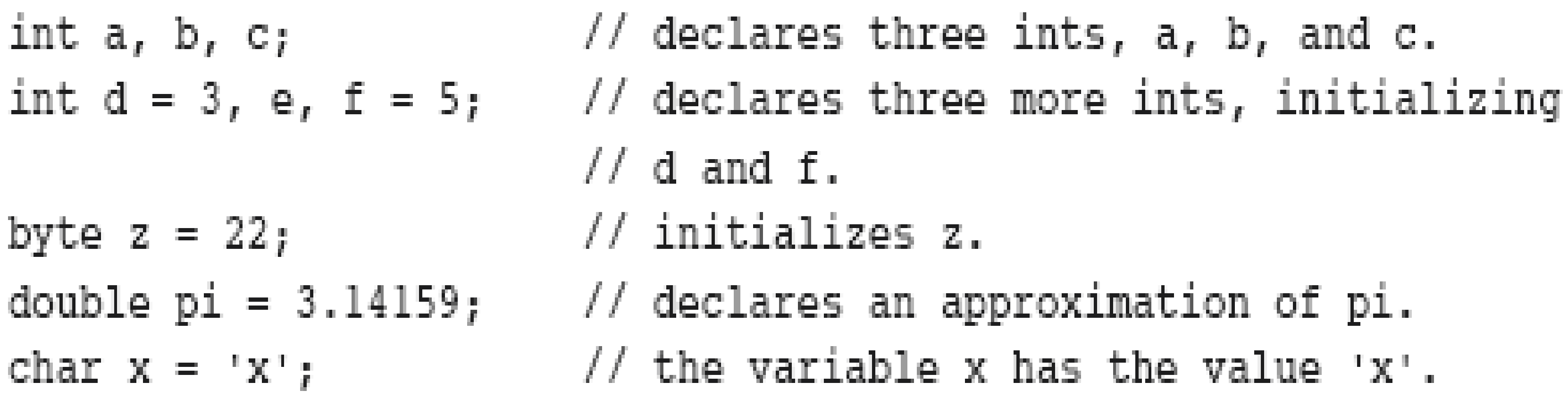
Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Declaring a Variable

- The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable: In Java, all variables must be declared before they can be used.

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



```
int a, b, c;              // declares three ints, a, b, and c.
int d = 3, e, f = 5;      // declares three more ints, initializing
                          // d and f.
byte z = 22;              // initializes z.
double pi = 3.14159;      // declares an approximation of pi.
char x = 'x';             // the variable x has the value 'x'.
```

Compile time Initialization

- Java allows variables to be initialized using any valid expression at the time the variable is declared.

Example:

```
class Initialize {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

The Scope and Lifetime of Variables

- Java allows variables to be declared within any block.
- A block is begun with an opening curly brace and ended by a closing curly brace
- A block defines a *scope.* A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

- In Java, the two major scopes are –
    - ➢ those defined by a class and
    - ➢ those defined by a method

- As a general rule, 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 ; but reverse not true

Example 1:

```
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);
    }
}
```

Example 2:

```
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1; // y is initialized each time block is entered
System.out.println("y is: " + y); // this always prints -1
y = 100;
System.out.println("y is now: " + y);
}
}
}
```

Example 3:

```
class ScopeErr {
public static void main(String args[]) {
int bar = 1;
{ // creates a new scope
int bar = 2; // Compile-time error – bar already defined!
}
}
}
```

Type Conversion and Casting

- When one type of data is assigned to another type of variable, 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.

  When these two conditions are met, a *widening conversion* takes place.

Ex: *byte* to *int*

Note:

1. Integer and floating-point types are compatible
2. Numeric types are not compatible with *char* or *boolean*
3. *char* or *boolean* are not compatible with each other

Casting Incompatible Types

- What if you want to assign an **int** value to a **byte** variable?

  This conversion will not be performed automatically, because a **byte** is smaller than an **int (called** *Narrowing conversion*: Ex: *int* to *byte)*

- Use a *cast* to create a conversion between two incompatible types

  cast is an explicit type conversion

  Syntax: *(target-type) value*

  (target-type specifies the desired type to convert the specified value to)

  If integer value larger than the *byte*'s range, reduce it modulo *byte*'s range

Ex:
```
int a;
byte b;
b = (byte) a;
```

- *Truncation*: A different type of conversion will occur when a floating-point value is assigned to an integer type. The fractional part is lost after conversion

  Ex: floating-point to integer (float to int)

Ex:
```
int a;
float b;
a = (int) b;
```

```
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);
    }
}
```

Output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Automatic Type Promotion in Expressions

Example:

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

- Java automatically promotes each **byte** or **short** operand to **int** when evaluating an expression
- Subexpression **a * b** is performed using integer

```
 byte b = 50
 b = b * 2
// error: Cannot assign an int to a byte
```

In this case we need to explicitly specify:

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

The Type Promotion Rules

- 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);
    }
}
```

- In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**
- Next, in the subexpression **i / c**, **c** is promoted to **int**, and the result is of type **int**
- In **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**
- The outcome of **float** plus an **int** is a **float**
- Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression

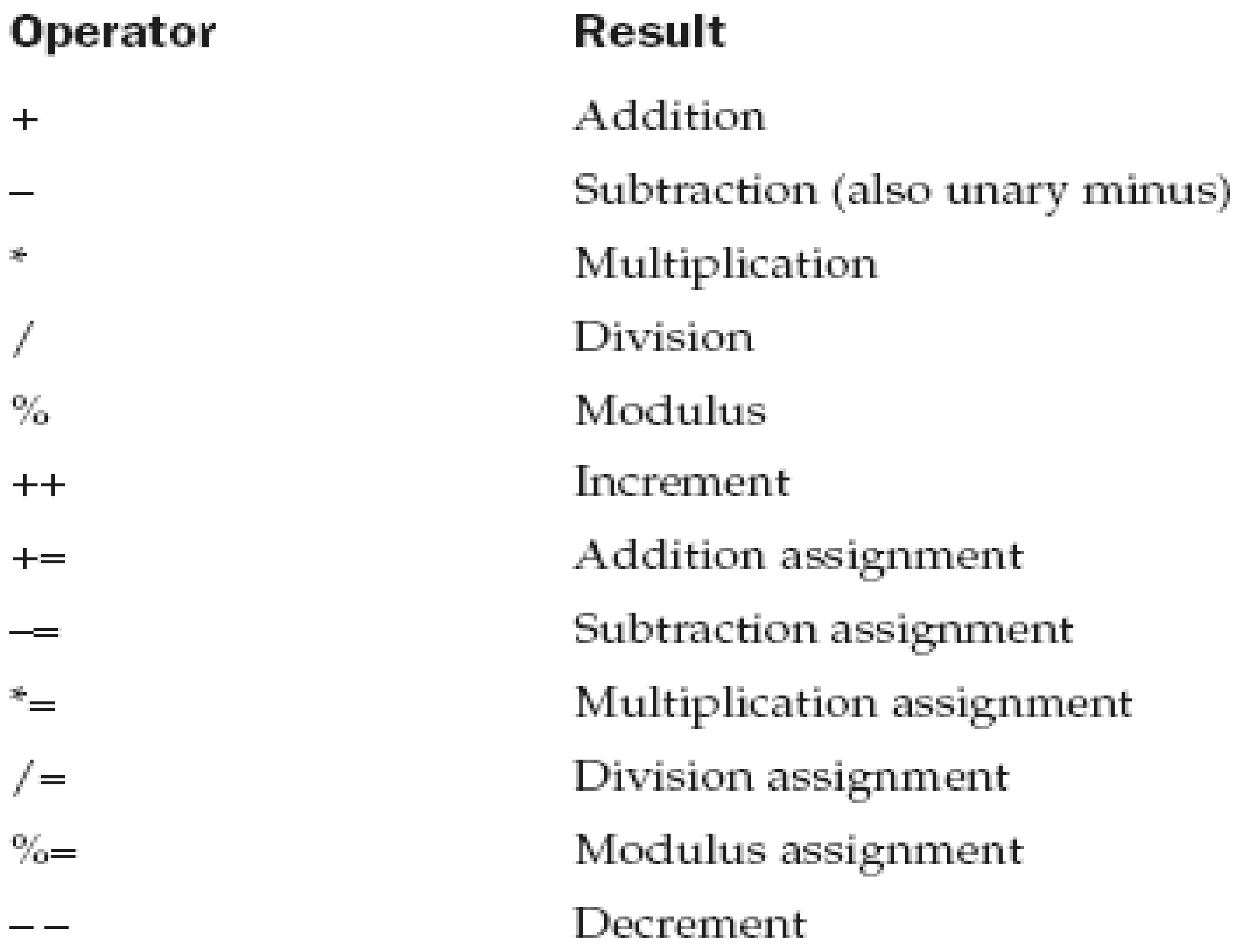
OPERATORS

Operators

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

Arithmetic Operators

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types



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

Basic arithmetic operators

- The basic arithmetic operations—addition, subtraction, multiplication, and division
- The minus operator also has a unary form that negates its single operand.
- when the division operator is applied to an integer type, there will be no fractional component attached to the result.

```
class BasicMath {
public static void main(String args[]) {
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
```

```
// arithmetic using doubles
System.out.println("\nFloating Point
Arithmetic");

double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

The Modulus Operator

- The modulus operator, **%**, returns the remainder of a division operation

- It can be applied to floating-point types as well as integer types

- This differs from C/C++, in which the **%** can only be applied to integer types

```
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

When you run this program you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

a = a + 4; this can be written as a += 4;

Any statement of the form

***var* = *var op expression*;**

can be rewritten as

***var op*= *expression*;**

Operator	Example	Equivalent To
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Increment and Decrement

- The ++ and the – – are Java’s increment and decrement operators.
- The increment operator increases its operand by one. The decrement operator decreases its operand by one.
- they can appear both in *postfix form,* and *prefix form,*

$$\left.\begin{array}{l} x = 42; \\ y = ++x; \end{array}\right\} \quad \begin{array}{l} x = x + 1; \\ y = x; \end{array}$$

$$\left.\begin{array}{l} x = 42; \\ y = x++; \end{array}\right\} \quad \begin{array}{l} y = x; \\ x = x + 1; \end{array}$$

The Bitwise Operators

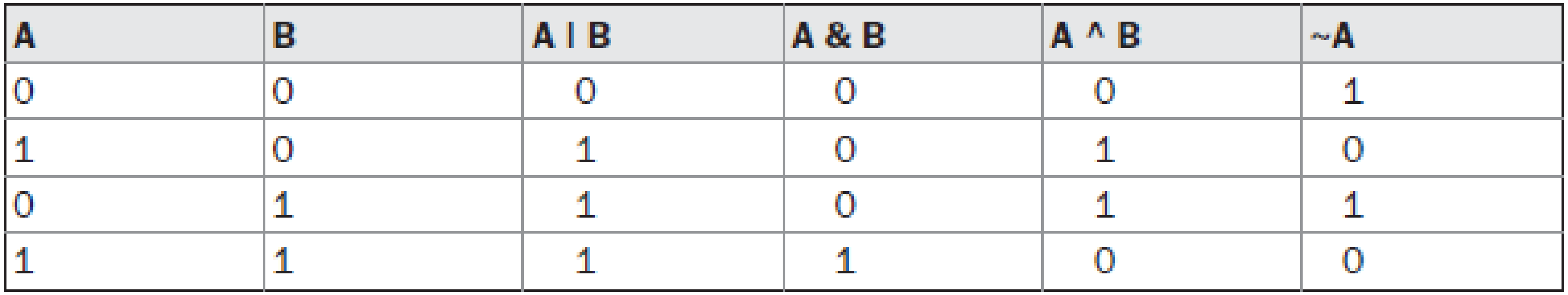
Java defines several *bitwise operators* that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands.

~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Shift Right
>>>	Shift Right zero fill
<<	Shift left
& =	Bitwise AND Assignment
|=	Bitwise OR Assignment
^=	Bitwise XOR Assignment
>>=	Shift Right Assignment
>>>=	Shift Right zero fill Assignment
<<=	Shift Left Assignment

The Bitwise Operators

The Bitwise Logical Operators

~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR



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

The Left Shift

It has this general form:

value << num

- If you left-shift a **byte** value, that value will first be promoted to **int** and then shifted
- This means that you must discard the top three bytes of the result if what you want is the result of a shifted **byte** value
- The easiest way to do this is to simply cast the result back into a **byte**

```
class ByteShift {
public static void main(String args[]) {
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
```

The Right Shift

- The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times

value >> num

Example:

```
int a = 32;
a = a >> 2; // a now contains 8
```

Relational Operators

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

>	**greater than**
>=	**greater than or equal to**
<	**less than**
<=	**less than or equal to**
= =	**equal to**
!=	**not equal to**

The outcome of these operations is a boolean value.

The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

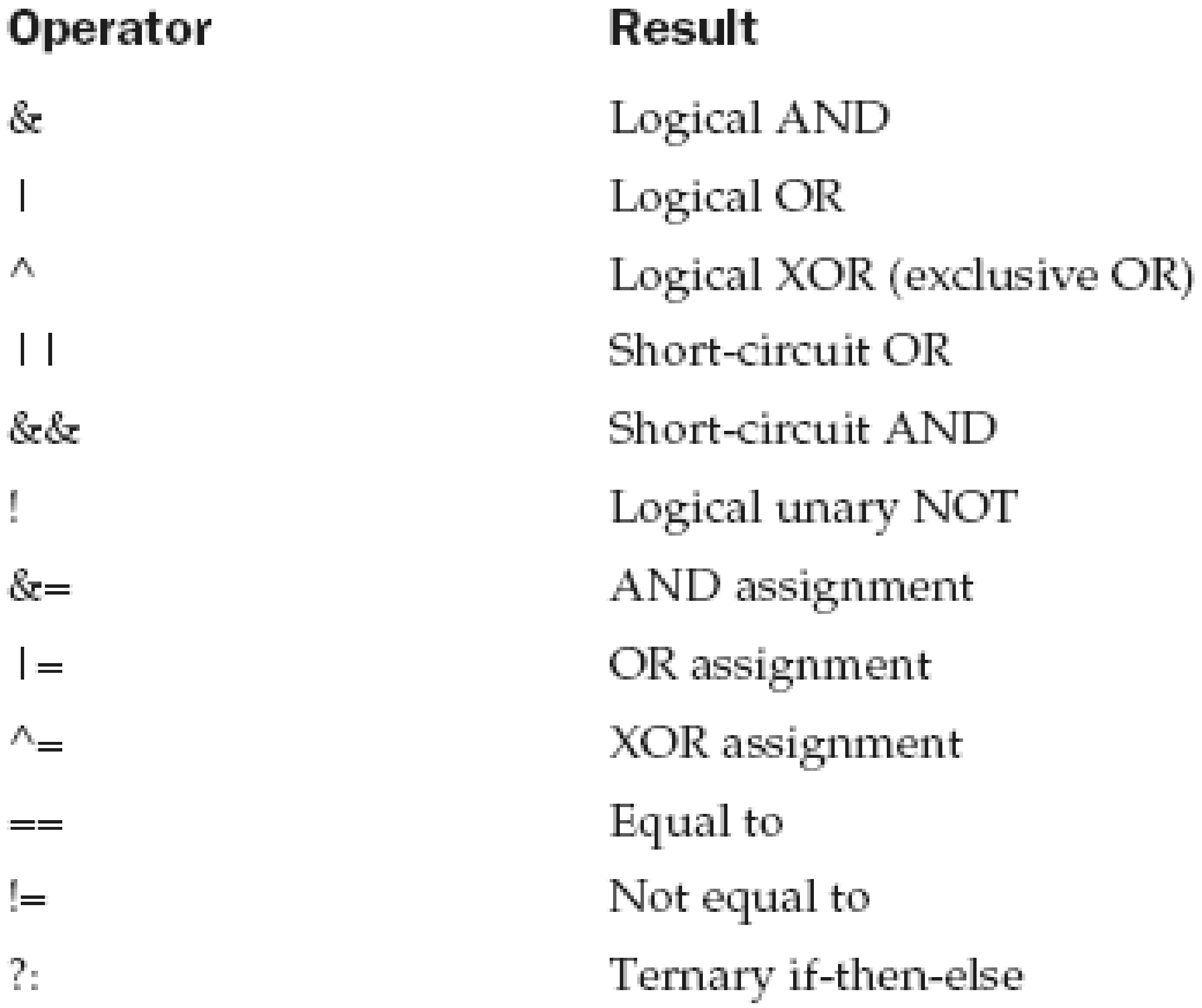
```
                  int a = 4;
                  int b = 1;
                  boolean c = a < b;

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

In Java, these statements must be written like this:

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

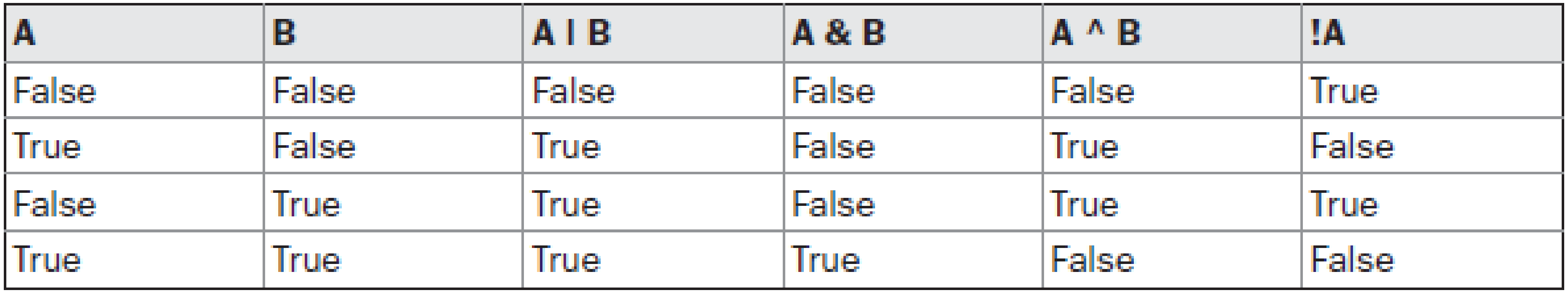
*In Java, **true** and **false** are nonnumeric values which do not relate to zero or nonzero*

Boolean Logical Operators



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

Boolean Logical Operators



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

The Assignment Operator

var = *expression*;

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

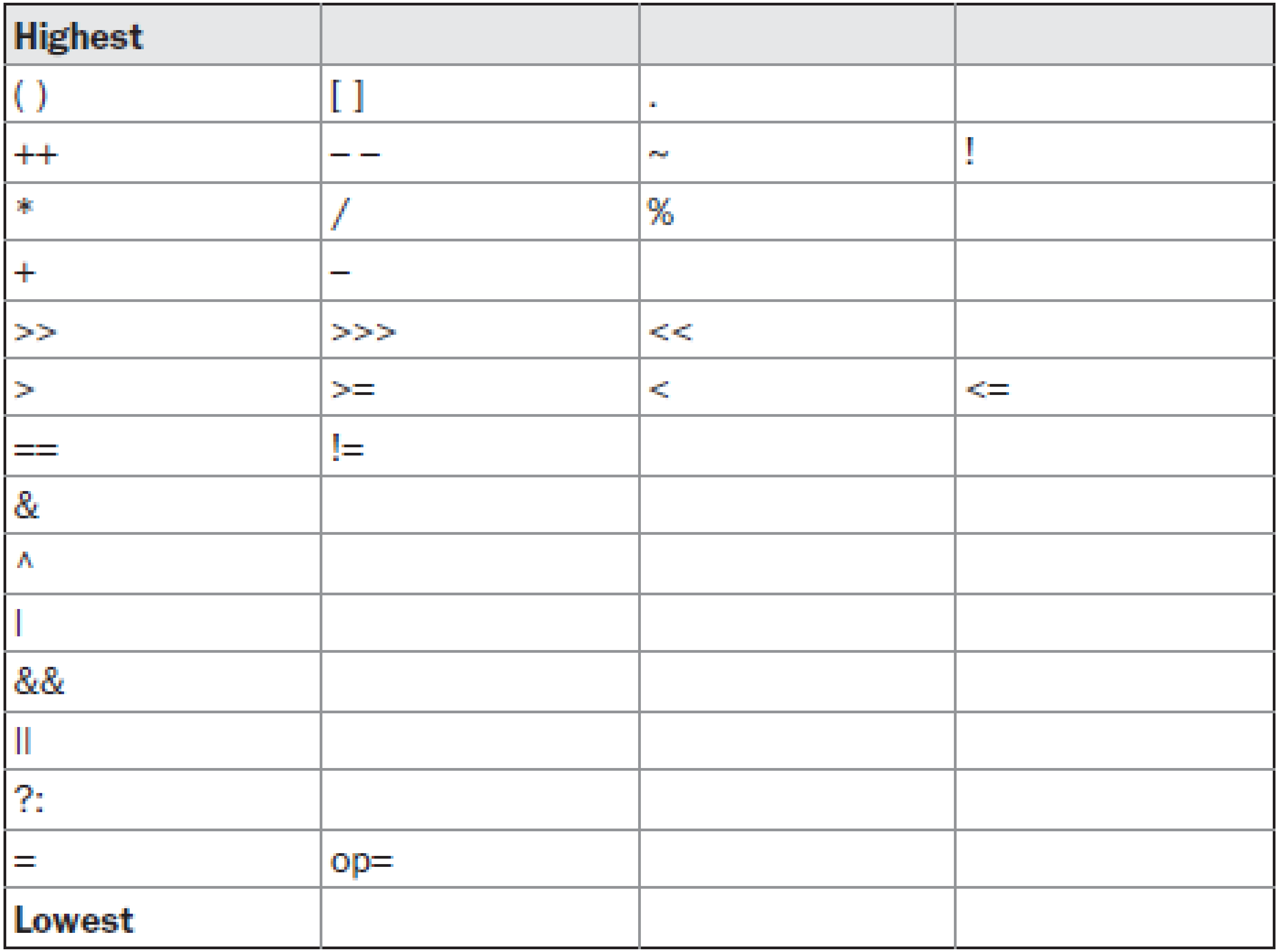
The ?: Operator (ternary)

General form: *expression1* **?** *expression2* **:** *expression3*

Examples:

```
if (a > b) { max = a; } else { max = b; }
max = (a > b) ? a : b;

ratio = (denom == 0) ? 0 : num / denom;

i = 10;
k = i < 0 ? -i : i; // get absolute value of i

i = -10;
k = i < 0 ? -i : i; // get absolute value of i
```

Operator Precedence



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

Control Statements

- Control statements cause the flow of execution to advance and branch based on changes to the state of a program.
- In Java, control statements are categorised as: selection, iteration, and jump.
- Selection statements allow the 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 (that is, iteration statements form loops).
- Jump statements allow your program to execute in a nonlinear fashion.

Java’s Selection Statements

- Java supports two selection statements: **if and switch.** These statements allow you to control the flow of your program’s execution based upon conditions known only during run time.

If : The **if statement** is Java’s conditional branch statement.

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

Nested If:

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

The if-else-if Ladder

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

The if-else-if Ladder:

```
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
        int month = 4; // April
        String season;
        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
}
```

Here is the output produced by the program:

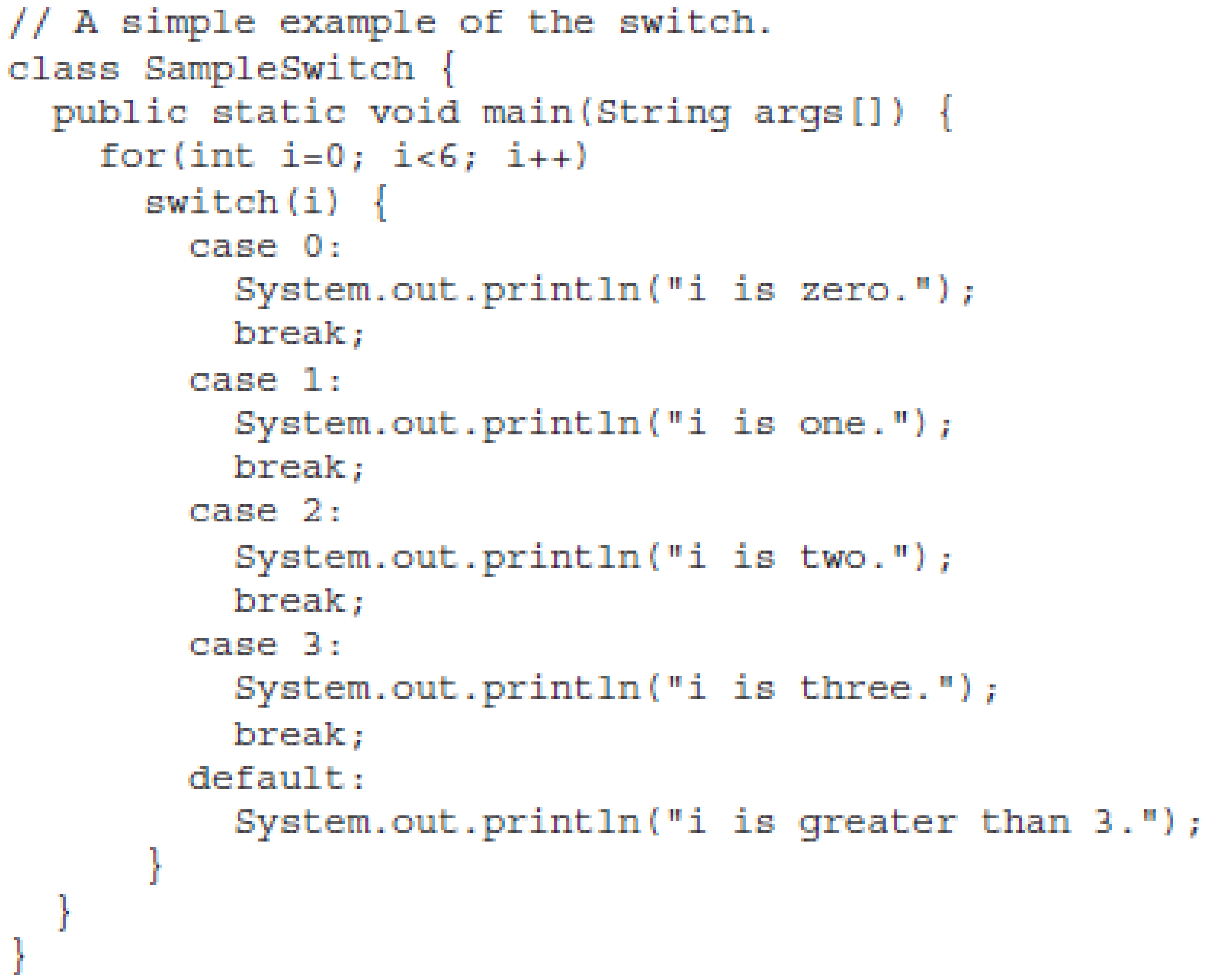
```
April is in the Spring.
```

switch

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

- *Note: The expression must be of type byte, short, int, or char;* each of the *values specified in the* case statements must be of a type compatible with the expression.

switch



```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

switch

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

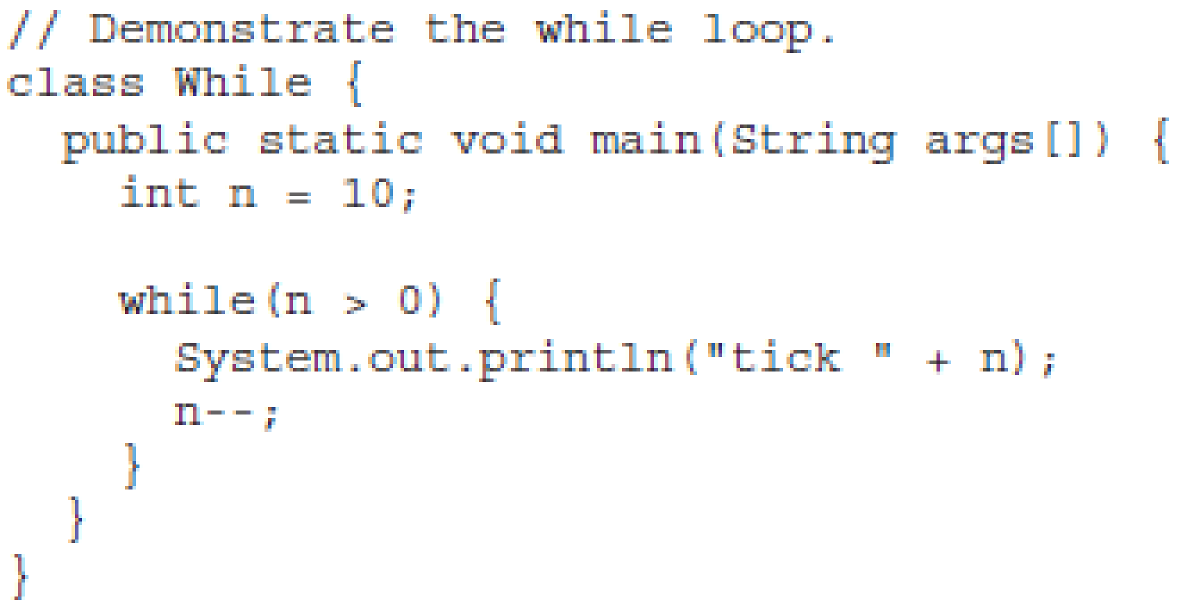
Iteration Statements

Java's iteration statements are: for, while, and do-while. These statements are called loops.
A loop repeatedly executes the same set of instructions until a termination condition is met.

While: while(*condition*) {

    // body of loop

}



```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

```
class  NoBody {
public static void main(String args[]) {
    int i, j;
    i = 100;
    j = 200;
          // find midpoint between i and j
    while(++i < --j) ;    // no body in this loop
    System.out.println("Midpoint is " + i);
    }
}
```

This program finds the midpoint between i and j. It generates the following output:

```
Midpoint is 150
```

do-while

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

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String args[]) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

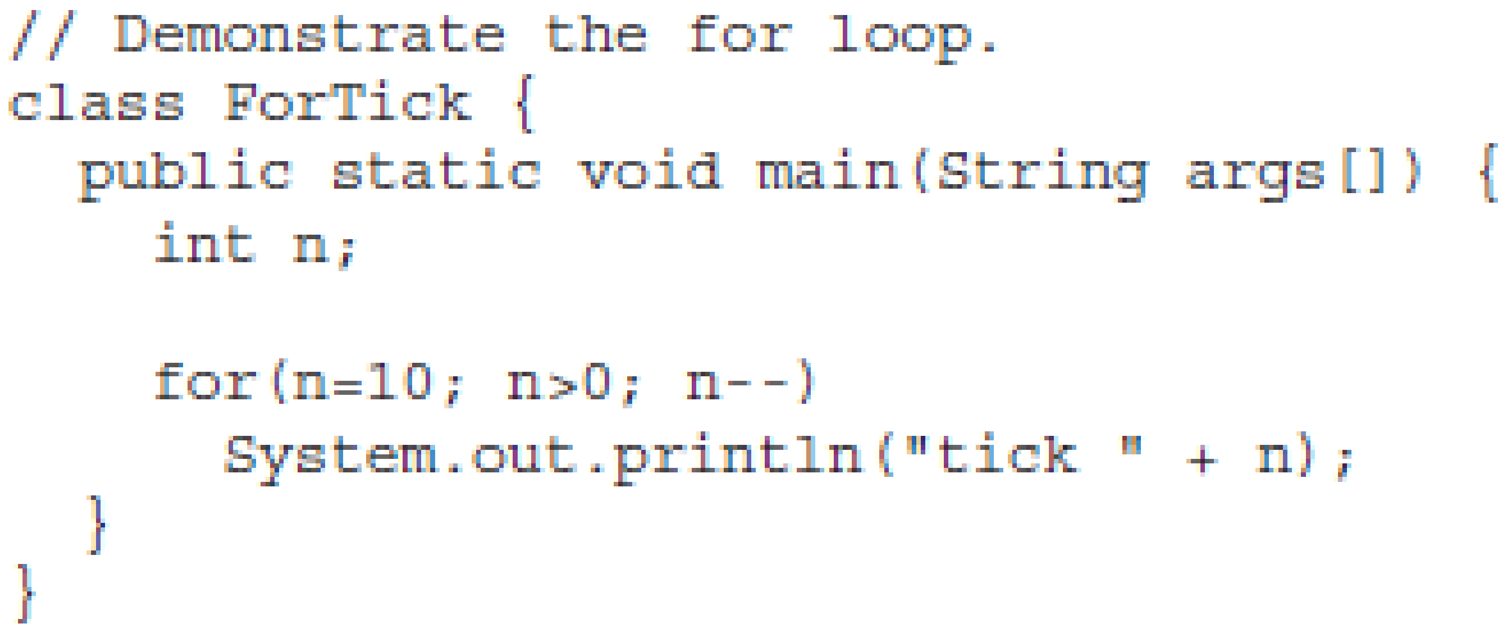
```
// Using a do-while to process a menu selection
class Menu {
public static void main(String args[])
throws java.io.IOException {
char choice;
do {
System.out.println("Help on:");
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.println(" 3. while");
System.out.println(" 4. do-while");
System.out.println(" 5. for\n");
System.out.println("Choose one:");
choice = (char) System.in.read();
} while( choice < '1' || choice > '5');
System.out.println("\n");
switch(choice) {
case '1':
System.out.println("The if:\n");
System.out.println("if(condition) statement;");
System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\n");
System.out.println("switch(expression) {");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" // ...");
System.out.println("}");
break;
case '3':
System.out.println("The while:\n");
System.out.println("while(condition) statement;");
break;
case '4':
System.out.println("The do-while:\n");
System.out.println("do {");
System.out.println(" statement;");
System.out.println("} while (condition);");
break;
case '5':
System.out.println("The for:\n");
System.out.print("for(init; condition; iteration)");
System.out.println(" statement;");
break;
}
}
}
```

for

for(*initialization*; *condition*; *iteration*) {

    // body

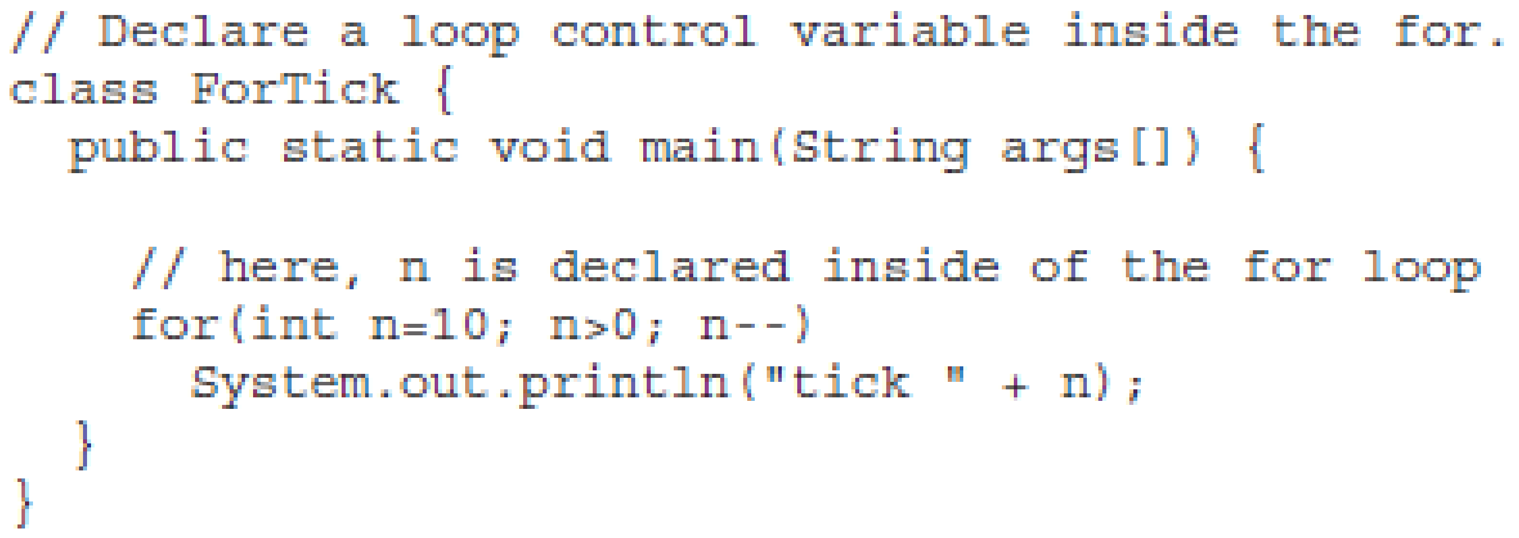
}



```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

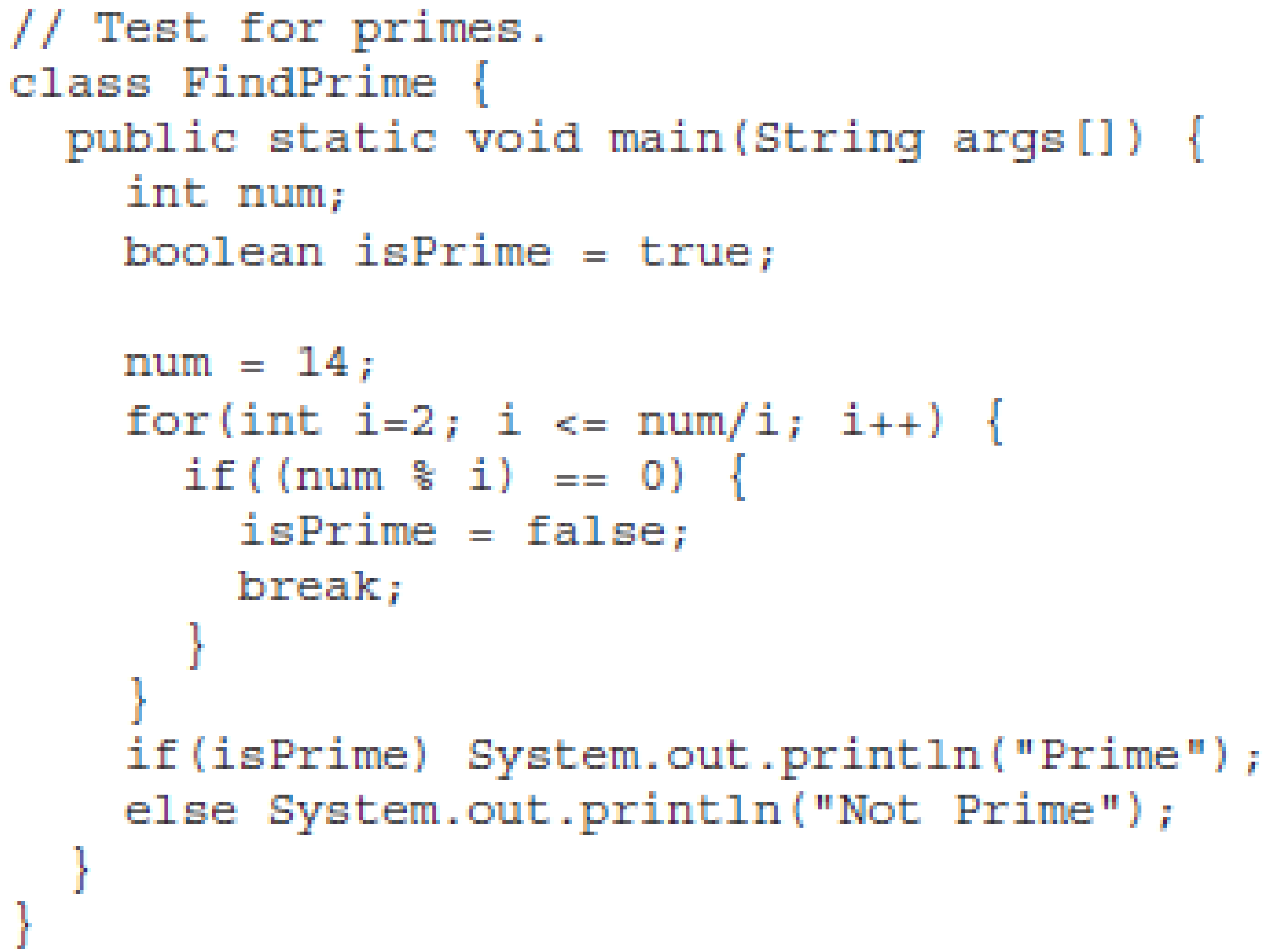
for

- Declaring Loop Control Variables Inside the for Loop



```
// Declare a loop control variable inside the for.
class ForTick {
  public static void main(String args[]) {

    // here, n is declared inside of the for loop
    for(int n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

for



```
// Test for primes.
class FindPrime {
  public static void main(String args[]) {
    int num;
    boolean isPrime = true;

    num = 14;
    for(int i=2; i <= num/i; i++) {
      if((num % i) == 0) {
        isPrime = false;
        break;
      }
    }
    if(isPrime) System.out.println("Prime");
    else System.out.println("Not Prime");
  }
}
```

Using the Comma (separator)

```
class Comma {
    public static void main(String args[]) {
            int a, b;
            for(a=1, b=4; a<b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
    }
  }
}
```

The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

Some for Loop Variations

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

```
// Parts of the for loop can be empty.

class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;
        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Another for loop variation, to create an infinite loop (a loop that never terminates)

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

- This loop will run forever, because there is no condition under which it will terminate

Nested Loops

Java allows loops to be nested. That is, one loop may be inside another.

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

The output produced

```
. . . . . . . . . .
. . . . . . . . .
. . . . . . . .
. . . . . . .
. . . . . .
. . . . .
. . . .
. . .
. .
.
```

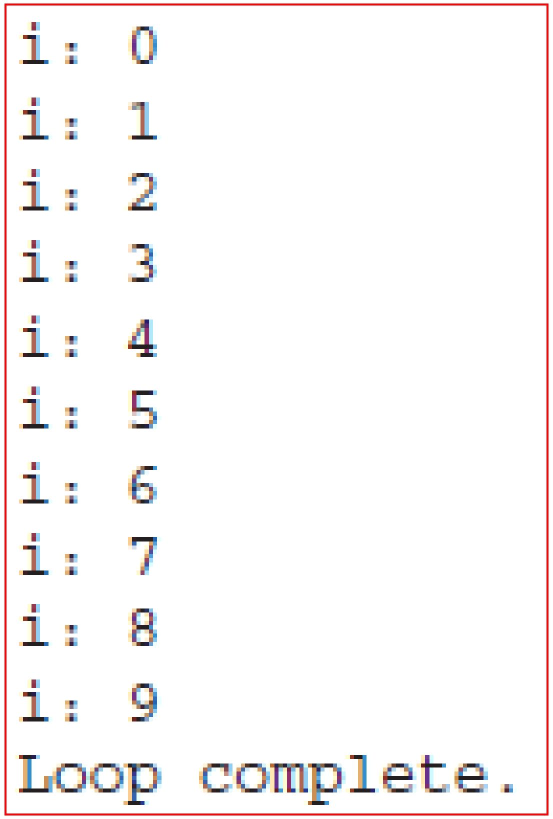
Jump Statements

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

Using break to Exit a Loop:

```
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

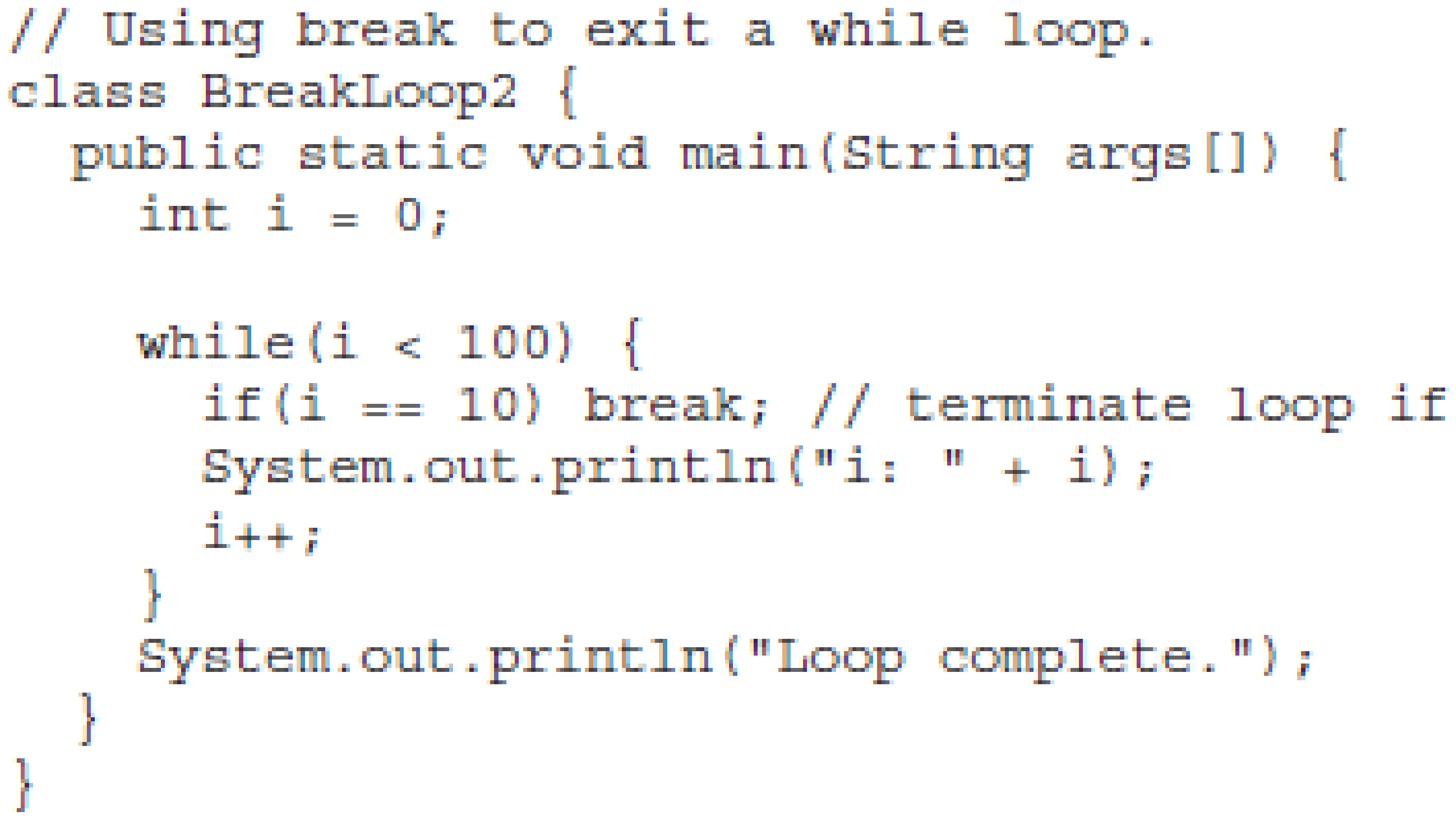
The output



```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

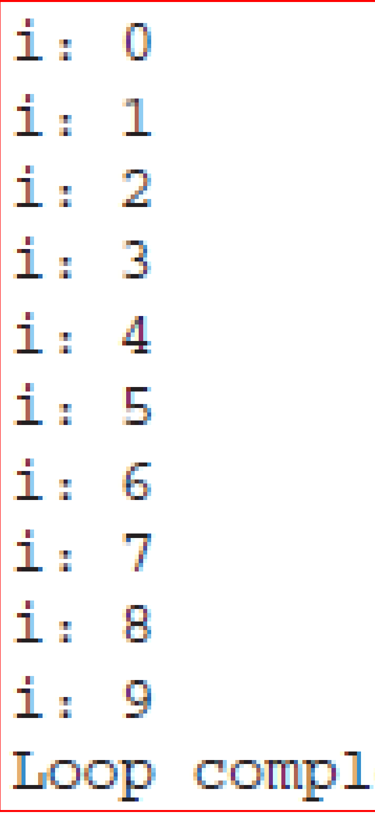
Jump Statements

Using break to Exit a Loop in while loop



```
// Using break to exit a while loop.
class BreakLoop2 {
  public static void main(String args[]) {
    int i = 0;

    while(i < 100) {
      if(i == 10) break; // terminate loop if
      System.out.println("i: " + i);
      i++;
    }
    System.out.println("Loop complete.");
  }
}
```

The output



```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop compl
```

Jump Statements

When used inside a set of nested loops, the break statement will only break out of the innermost loop.

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String args[]) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

- More than one **break** statement may appear in a loop

- Too many **break** statements have the tendency to destructure your code

- The **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops

Using break as a Form of Goto

- Java defines an expanded form of the **break** statement
- By using this form of **break**, we can break out of one or more blocks of code
- The general form of the labeled **break** statement is :

  break *label*;

  A *label* is any valid Java identifier followed by a colon
- We can use a labeled break statement to exit from a set of nested blocks

Example

```
// Using break as a civilized form of goto.
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.");
    }
  }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

One of the most common uses for a labeled **break statement is to exit from nested loops.** For example, in the following program, the outer loop executes only once:

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

We cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile

```
// This program contains an error.
class BreakErr {
    public static void main(String args[]) {
            one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
    }
    for(int j=0; j<100; j++) {
    if(j == 10) break one; // WRONG
    System.out.print(j + " ");
    }
  }
}
```

Using continue

```
class Continue {
   public static void main(String args[]) {
      for(int i=0; i<10; i++) {
          System.out.print(i + " ");
          if (i%2 == 0) continue;
          System.out.println("");
        }
     }
}
```

- 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();
    }
}
```

Output:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return

- Used to explicitly return from a 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==true) return; // return to caller
        System.out.println("This won't execute.");
    }
}
```

return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )

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

- To create an array, you first must create an array variable of the desired type

  type var-name[ ];

  Example: int month_days[];   // No array exists=>array with no value

We must allocate memory and assign it to array variable

array-var = new type[size];

Example:

```
 int month_days[];
 month_days[] = new int[12];
```

Also possible to combine the declaration of array with the allocation of the array,

```
      int month_days[];
      month_days=new int [12];
```

- Arrays can be initialized when they are declared
- An *array initializer* is a list of comma-separated expressions surrounded by curly braces
- There is no need to use **new**

Example:

```
class AutoArray {
public static void main(String args[ ]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}
```

Example: program to find the average of a set of 5 numbers stored in an array

```
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;
        for(i=0; i<5; i++)
        result = result + nums[i];
        System.out.println("Average is " + result / 5);
    }
}
```

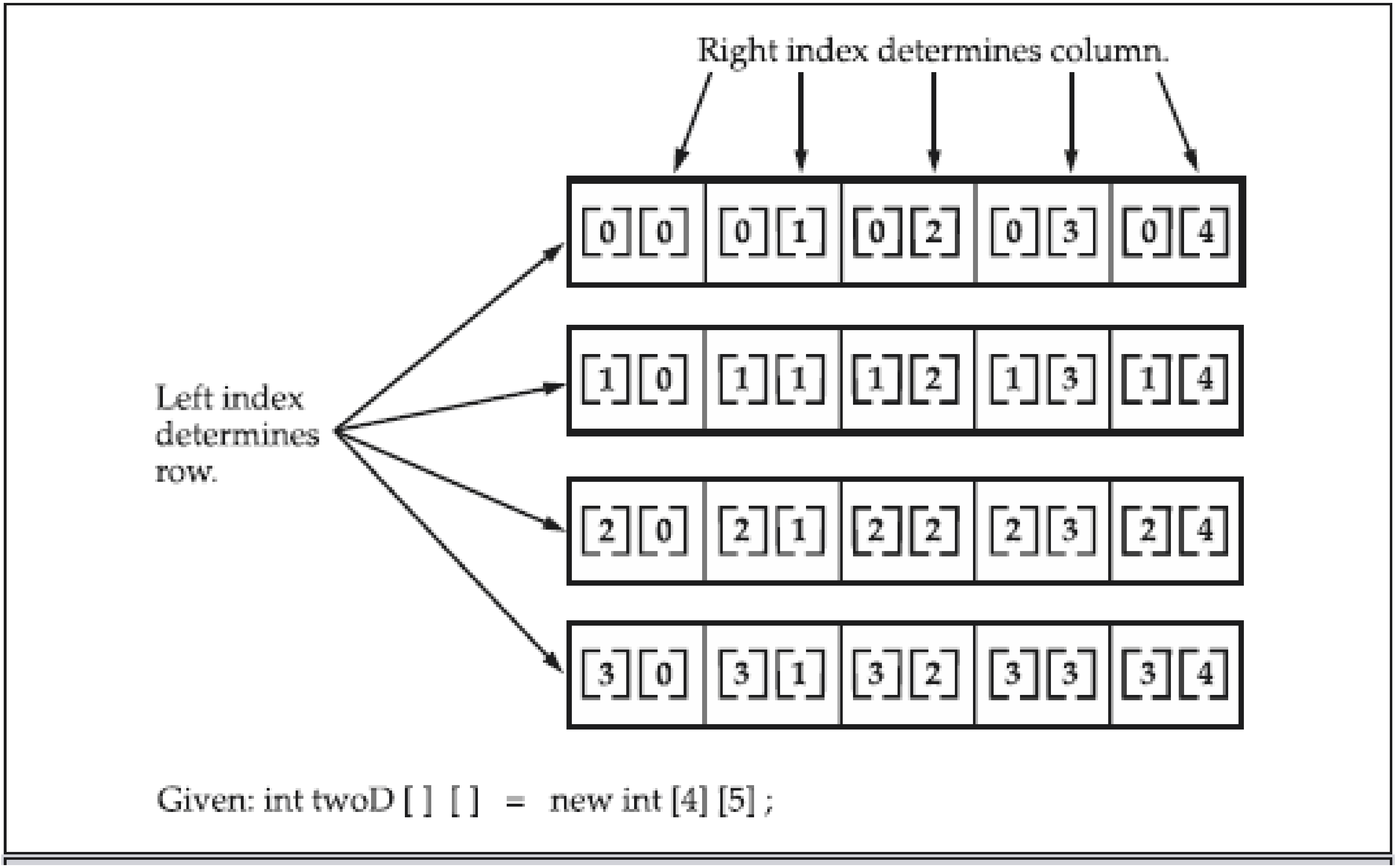
Multidimensional Arrays

- 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

Example:

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

This allocates a 4 by 5 array and assigns it to **twoD**.


Right index determines column.
Left index determines row.
[0][0]
[0][1]
[0][2]
[0][3]
[0][4]
[1][0]
[1][1]
[1][2]
[1][3]
[1][4]
[2][0]
[2][1]
[2][2]
[2][3]
[2][4]
[3][0]
[3][1]
[3][2]
[3][3]
[3][4]
Given: int twoD [ ] [ ] = new int [4] [5] ;

```
class TwoDArray {
    public static void main(String args[]) {
            int twoD[][]= new int[4][5];
            int i, j, k = 0;
            for(i=0; i<4; i++)
                for(j=0; j<5; j++) {
                        twoD[i][j] = k;
                        k++;
                  }
            for(i=0; i<4; i++) {
                for(j=0; j<5; j++)
                        System.out.print(twoD[i][j] + " ");
                        System.out.println();
                  }
        }
}
```

This program generates the following output:

```
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension

- You can allocate the remaining dimensions separately

Example:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

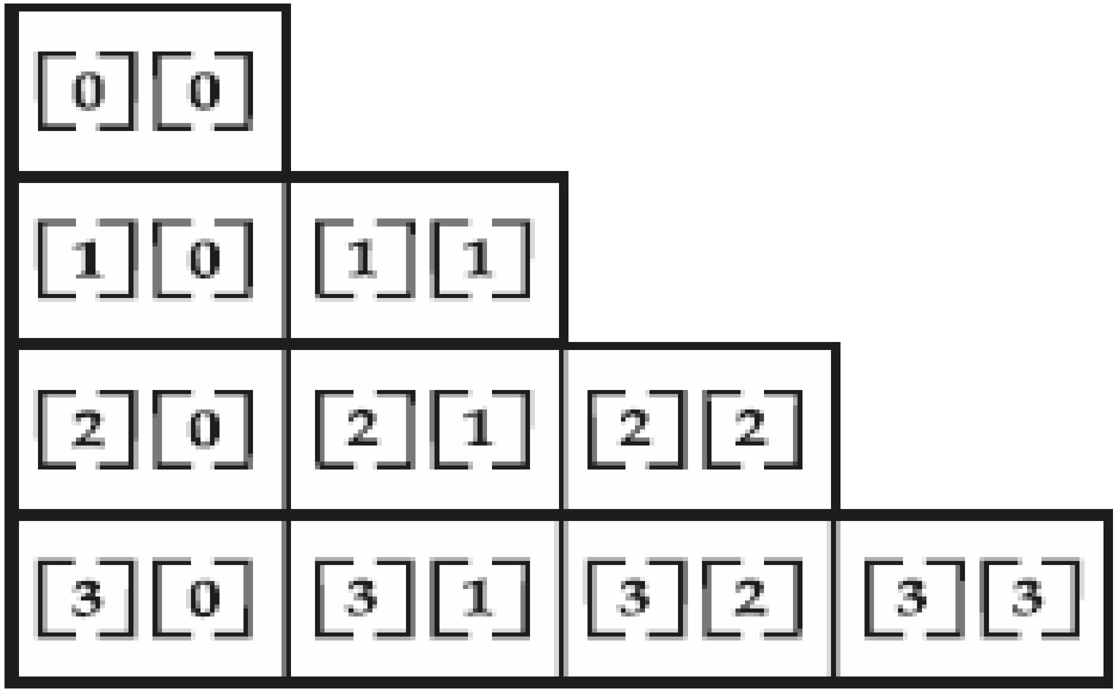
- When you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension
- Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control

```
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
} } }
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



- It is possible to initialize multidimensional arrays
- You can use expressions as well as literal values inside of array initializers

```
class Matrix {
public static void main(String args[]) {
double m[ ][ ] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();
}
}
}
```

Output:

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Alternative Array Declaration Syntax

- There is a second form that may be used to declare an array:

  type*[ ] *var-name;

Example: These two are equivalent

```
int al[ ] = new int[3];
int[ ] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[ ][ ] = new char[3][4];
char[ ][ ] twod2 = new char[3][4];
```

Note:

- Java does not support or allow pointers
- Java cannot allow pointers, because doing so would allow Java applets to breach the firewall between the Java execution environment and the host computer
- Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one

Reading data from keyboard

There are many ways to read data from the keyboard. For example:

- InputStreamReader and BufferedReader class
- Scanner

a) InputStreamReader class and BufferedReader class

- o BufferedReader class can be used to read data line by line by readLine() method.

Example 1: Reading data from keyboard by InputStreamReader and BufferdReader class

```
import java.io.*;
class readfromkeyboard  {
public static void main(String args[])throws Exception{
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
 }
}
```

Output:

```
Enter your name
Chinmayee
Welcome Chinmayee
```

Reading data from keyboard

b) Scanner Class in Java

- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.
- It is the easiest way to read input in a Java program.
- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.
- The Java Scanner class is widely used to parse text for strings and primitive types.
- The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Scanner Class Methods

boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	It scans the next token of the input as a byte.
double	nextDouble()	It scans the next token of the input as a double.
float	nextFloat()	It scans the next token of the input as a float.
int	nextInt()	It scans the next token of the input as an Int.
String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
long	nextLong()	It scans the next token of the input as a long.

Example 1: Use of Java Scanner class to read a single input (a string through in.nextLine() method) from the user.

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
        in.close();
        }
}
```

Output:

```
Enter your name: Ananya Tripathy
Name is: Ananya Tripathy
```