Programming Languages

- Computers do not understand human languages, so programs must be written in a language a computer can use.
- Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps.
- Three general language types:
  - Machine Language
  - Assembly Language
  - High-Level Language

Machine Language

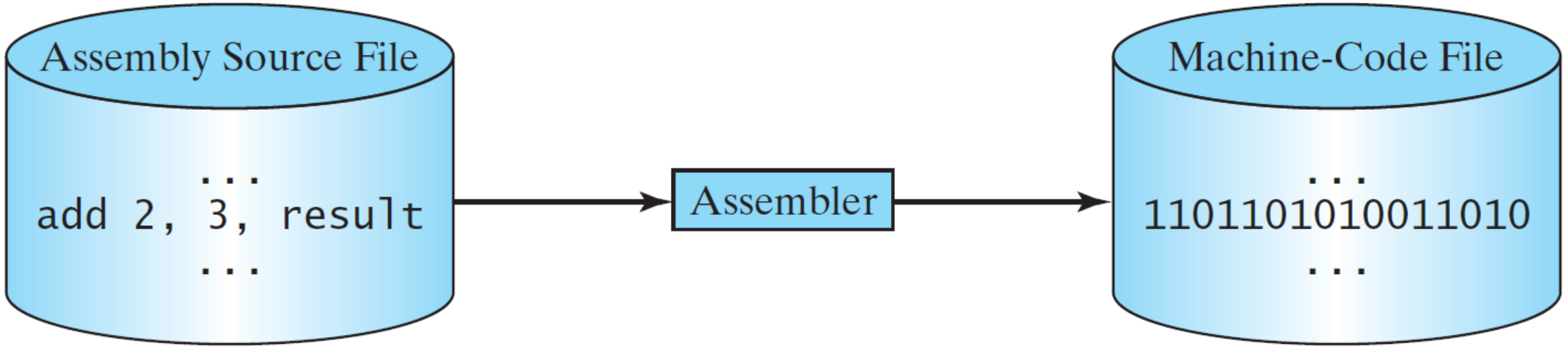
- Computer's native language—a set of built-in primitive instructions defined by its hardware design.
- Machine dependent—a particular machine language can be used on only one type of computer.
- Any computer can directly understand only its own machine language.
- These instructions are in the form of binary code.
- Example:
  - Add two numbers: **1101101010011010**

Assembly Language

- Machine Language issues:
  - Tedious Process
  - Very difficult to read and modify
- **Assembly Language** was created as an alternative to machine languages.
- **mnemonic**—a short descriptive word to represent each of the machine-language instructions.
- **Example:** Add two numbers → `add 2, 3, result`

Assembler

- Translator programs called assemblers convert assembly-language programs to machine language.



Assembly Language (Cont.)

- Although writing code in assembly language is easier than in machine language, it is still tedious to write code in assembly language.
- An instruction in assembly language corresponds to an instruction in machine code.
- *low-level language*—it is close in nature to machine language and is machine dependent

High-Level Language

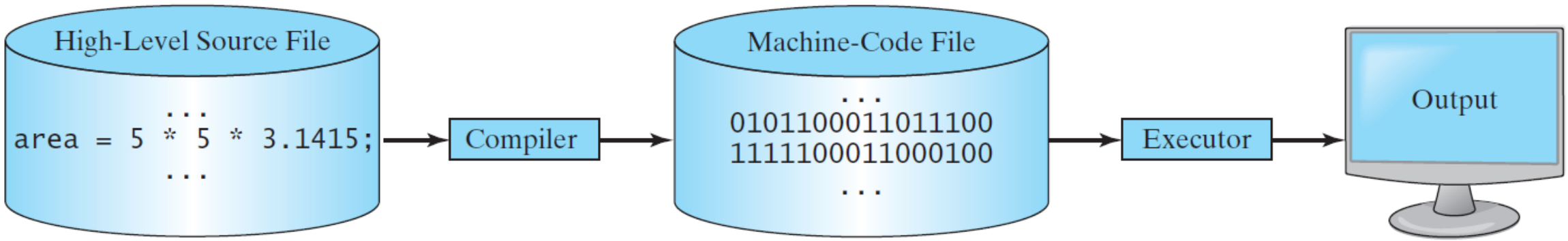
- Platform independent—you can write a program and run it in different types of machines.
- Allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations.
- *statements*—The instructions in a high-level programming language.
- Single statement can accomplish substantial tasks.
- A *statement* Example: computing area of a circle with a radius of **5**
  - **area = 5 * 5 * 3.14159;**

High-Level Language

- *source program* or *source code*—A program written in a high-level language
- A source program must be translated into machine code for execution.
  - **Compiler**
  - **Interpreter**
- Java uses a clever mixture of compilation and interpretation to run programs.
- C, C++, Microsoft's .NET languages (e.g., Visual Basic, Visual C++ and C#) are among the most widely used high-level programming languages; Java is by far one of the most widely used.
- Java evolved from C++, which evolved from C

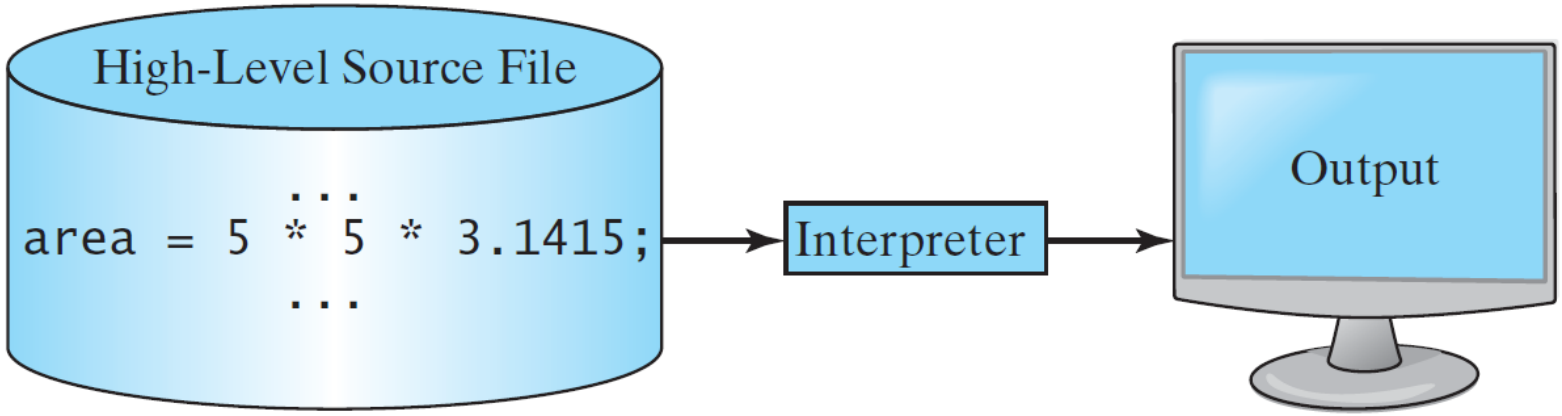
Compiler

- A compiler translates the entire source code into a machine-code **file**, and the machine-code file is then executed.
- It can take a considerable amount of computer time.



Interpreter

- An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away.
- It is slower than **compiled** programs run.



History of Java

- Sun Microsystems funded an internal corporate research project, which resulted in a C++-based language named Java.
- Originally called *Oak*, Java was designed for use in embedded chips in consumer electronic appliances.
- The web exploded in popularity, Sun saw the potential of using Java to add dynamic content to web pages.
- In 1995, Java was redesigned for developing Web applications.

Java Language Specification and API

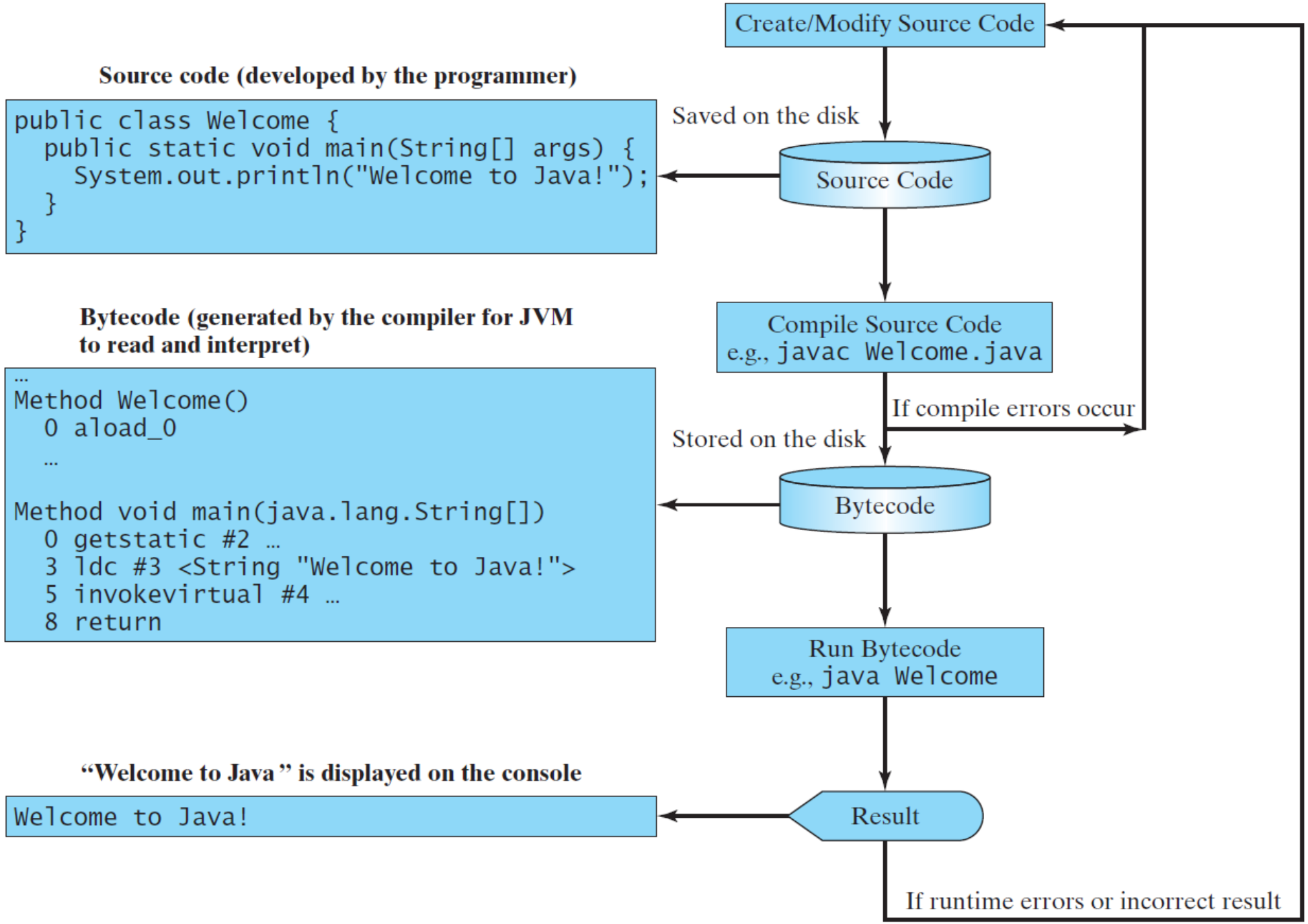
- The **Java language specification** is a technical definition of the Java programming language's syntax and semantics.
  - http://docs.oracle.com/javase/specs/
- Java programs consist of pieces called classes.
- Classes include methods that perform tasks and return information when the tasks complete.
- **Application Program Interface (API)**, also known as **library**, contains predefined classes and interfaces for developing Java programs.
  - https://docs.oracle.com/javase/8/docs/api/

Java Editions

- Java **Standard Edition (Java SE)** to develop client-side applications. The applications can run standalone or as applets running from a Web browser.
- Java **Enterprise Edition (Java EE)** to develop server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF).
- Java **Micro Edition (Java ME)** to develop applications for mobile devices, such as cell phones.
- Java SE is a foundation that all other Java technology is based on it.

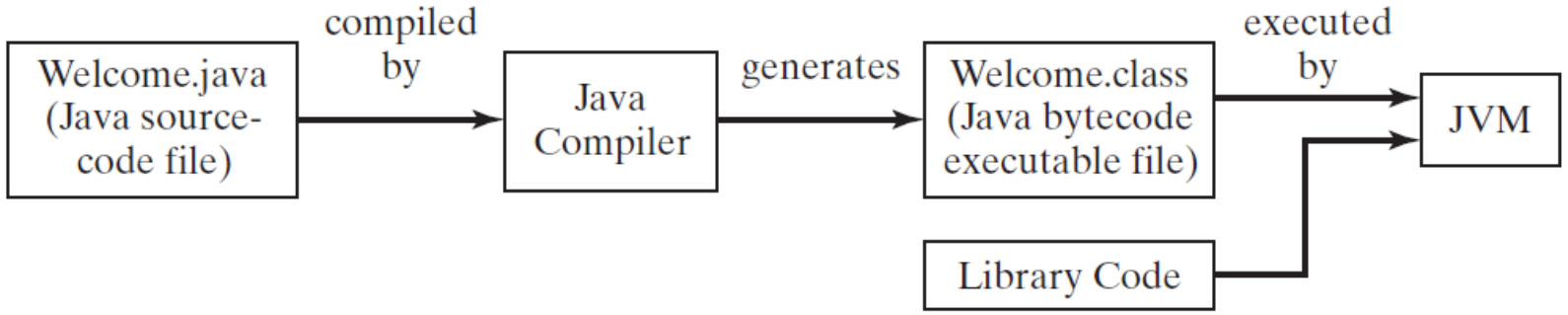
Java **JDK**

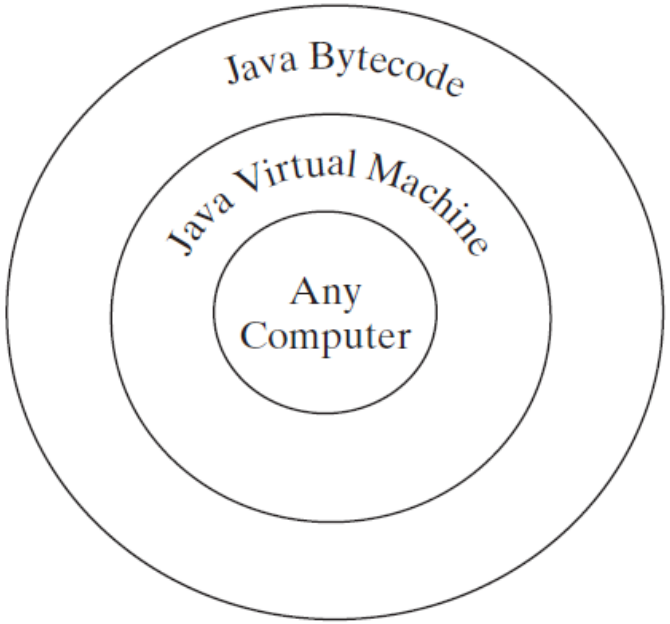
- **Java Development Toolkit (JDK)** consists of a set of separate programs, each invoked from a command line, for developing and testing Java programs.
- Oracle releases each version of Java SE with a JDK.
- The latest **JDK** version of Java SE is Oracle JDK 1.8.0_112
  - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html


Create/Modify Source Code
Source code (developed by the programmer)
public class Welcome {
public static void main(String[] args) {
System.out.println("Welcome to Java!");
}
}
Saved on the disk
Source Code
Compile Source Code
e.g., javac Welcome.java
If compile errors occur
Bytecode (generated by the compiler for JVM to read and interpret)
...
Method Welcome()
0 aload_0
...
Method void main(java.lang.String[])
0 getstatic #2 ...
3 ldc #3 <String "Welcome to Java!">
5 invokevirtual #4 ...
8 return
Stored on the disk
Bytecode
Run Bytecode
e.g., java Welcome
"Welcome to Java" is displayed on the console
Welcome to Java!
Result
If runtime errors or incorrect result

Java Virtual Machine (JVM)

- The **bytecode** is similar to machine instructions but is architecture neutral and can run on any platform that has a **Java Virtual Machine (JVM)**.





Editor

- To create or edit a Java file, you can use editor program (normally known simply as an editor).
- The source file must end with the **.java** extension.
- Save the program
  - A file name ending with the .java extension indicates that the file contains Java source code.
  - It must have the same exact name as the public class name.
- Linux editors: vi, emacs, and jEdit.
- Windows editors: Notepad++, EditPlus (www.editplus.com), TextPad (www.textpad.com) and jEdit (www.jedit.org).

Java **IDE**

- Java development tool—software that provides an integrated development environment (IDE) for developing Java programs quickly.
  - **Eclipse**
  - NetBeans
  - TextPad
- Editing, compiling, building, debugging, and online help are integrated in one graphical user interface.

Compiling

- Use the command javac (the Java compiler) to compile a program. For example, to compile a program called `HelloWorld.java`, you'd type

```
javac HelloWorld.java
```

- If the program compiles, the compiler produces a .class file called `HelloWorld.class` that contains the compiled version of the program.

Compiling (Cont.)

- Java compiler translates Java source code into bytecodes that represent the tasks to execute.
- The **bytecode** is similar to machine instructions but is architecture neutral and can run on any platform that has a **Java Virtual Machine (JVM)**.
- Bytecodes are executed by the Java Virtual Machine (JVM)—a part of the JDK and the foundation of the Java platform.
- Virtual machine (VM)—a software application that simulates a computer
  - Hides the underlying operating system and hardware from the programs that interact with it.
- One of Java's primary advantages: **Java bytecode can run on a variety of hardware platforms and operating systems.**

Executing

- Bytecodes are platform independent
  - They do not depend on a particular hardware platform.
- Bytecodes are portable
  - The same bytecodes can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled.
- The JVM is invoked by the java command. For example, to execute a Java application called `Helloworld`, you'd type the command

```
java HelloWorld
```

Comment Style

- Line comments
  - Beginning with //
- Block Comments
  - Begin with /* and end with */

Indentation and Spacing

- A consistent indentation style makes programs clear and easy to read, debug, and maintain.
- **Indentation** is used to illustrate the structural relationships between a program's components or statements.
- Example:
  - `System.out.println(3+4*4);`
  - `System.out.println(3 + 4 * 4);`

First Java Program!!!

- Let’s create our first Java program called HelloWolrd.java that just prints “Hello World!!!” into the output screen.

Programming Errors

- Programming errors can be categorized into three types:
  - Syntax errors
  - Runtime errors
  - Logic errors

Syntax Errors

- Errors that are detected by the compiler are called **syntax errors** or compile errors.
- Syntax errors result from errors in code construction:
  - mistyping a keyword
  - omitting some necessary punctuation
  - using an opening brace without a corresponding closing brace
- A single error will often display many lines of compile errors
  - Fix errors from the top line and work downward.
  - Fixing errors that occur earlier in the program may also fix additional errors that occur later

Example of Syntax Errors

```
1  public class ShowSyntaxErrors {
2    public static main(String[] args) {
3      System.out.println("Welcome to Java);
4    }
5  }
```

Runtime Errors

- **Runtime errors** are errors that cause a program to terminate abnormally.
- They occur while a program is running if the environment detects an operation that is impossible to carry out.
- Input mistakes typically cause runtime errors that are called input errors.
- Examples:
  - data-type error: if the program expects to read in a number, but instead the user enters a string.
  - division by zero: this happens when the divisor is zero for integer divisions.

Example of Runtime Error

```
1  public class ShowRuntimeErrors {
2    public static void main(String[] args) {
3      System.out.println(1 / 0);
4    }
5  }
```

Logic Errors

- **Logic errors** occur when a program does not perform the way it was intended to.

```
1  public class ShowLogicErrors {
2    public static void main(String[] args) {
3      System.out.println("Celsius 35 is Fahrenheit degree ");
4      System.out.println((9 / 5) * 35 + 32);
5    }
6  }
```

```
Celsius 35 is Fahrenheit degree
67
```

Common Errors

- Missing a closing brace
  - Each opening brace must be matched by a closing brace.
  - To avoid this error, type a closing brace whenever an opening brace is typed.
  - Most of the IDEs automatically inserts a closing brace for each opening brace typed.
- Missing a semicolon
  - Each statement ends with a statement terminator (;)

Common Errors (Cont.)

- Missing quotation marks for strings
  - To avoid this error, type a closing quotation whenever an opening quotation is typed.
  - Most of IDEs automatically insert a closing quotation mark for each opening quotation mark typed
- Misspelling names

```
1  public class Test {
2    public static void Main(string[] args) {
3      System.out.println((10.5 + 2 * 3) / (45 - 3.5));
4    }
5  }
```

Identifiers

- An identifier is a sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
  - An identifier cannot be a keyword/reserved word.
    - **keywords** are reserved for use by Java and are always spelled with all **lowercase** letters.
  - An identifier **cannot be** true, false, or null.
  - An identifier can be of any length.

Variables

```
1. // Compute the first area
2. radius = 1.0;
3. area = radius * radius * 3.14159;
4. System.out.println("The area is " +
     area + " for radius " + radius);

1. // Compute the second area
2. radius = 2.0;
3. area = radius * radius * 3.14159;
4. System.out.println("The area is " +
     area + " for radius " + radius);
```

Declaring Variables

```
int x;              // Declare x to be an
                    // integer variable;
double radius;      // Declare radius to
                    // be a double variable;
char a;             // Declare a to be a
                    // character variable;
```

Assignment Statements

```
x = 1;              // Assign 1 to x;

radius = 1.0;   // Assign 1.0 to radius;

a = 'A';            // Assign 'A' to a;
```

Declaration and Assignment in One Step

```
int x = 1;

double d = 1.4;
```

Constants

```
final datatype CONSTANTNAME = VALUE;


final double PI = 3.14159;
final int SIZE = 3;
```

Naming Conventions

- Choose meaningful and descriptive names.
- **Variable** and **method** names use lowercase.
  - If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name. For example, the variables radius and area, and the method computeArea.
- Constants:
  - Capitalize all letters in constants, and use underscores to connect words. For example, the constant PI and MAX_VALUE

Numerical Data Types

Name	*Range*	*Storage Size*	
`byte`	-2^7 to $2^7 - 1$ (−128 to 127)	8-bit signed	`byte` type
`short`	-2^{15} to $2^{15} - 1$ (−32768 to 32767)	16-bit signed	`short` type
`int`	-2^{31} to $2^{31} - 1$ (−2147483648 to 2147483647)	32-bit signed	`int` type
`long`	-2^{63} to $2^{63} - 1$	64-bit signed	`long` type
	(i.e., −9223372036854775808 to 9223372036854775807)		
`float`	Negative range: −3.4028235E + 38 to −1.4E − 45	32-bit IEEE 754	`float` type
	Positive range: 1.4E − 45 to 3.4028235E + 38		
`double`	Negative range: −1.7976931348623157E + 308 to −4.9E − 324	64-bit IEEE 754	`double` type
	Positive range: 4.9E − 324 to 1.7976931348623157E + 308		

Numeric Operators

Name	*Meaning*	*Example*	*Result*
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

Integer Division

+, -, *, /, and %

5 / 2 yields an integer 2.

5.0 / 2 yields a double value 2.5

5 % 2 yields 1 (the remainder of the division)

The String Type

- The char type only represents one character. To represent a string of characters, use the data type called String. For example,

```
String message = "Welcome to Java";
```

- String is actually a predefined class in the Java. The String type is not a primitive type.

String Concatenation

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes
SupplementB
```

```
// HelloWorld.java

public class HelloWorld {

      // method main executes the Java application
      public static void main (String[] args) {

            // this line prints a single line of text to default output
            System.out.println("Hello World !!!");

   } // end method main

} // end class HelloWorld
```

Class declaration

```
class HelloWorld
```

- Every Java program consists of at least one class that you define it.
- `Class keyword` introduces a class declaration and is immediately followed by the **ClassName**.

Class Names

- By convention, begin with a capital letter and capitalize the first letter of each word they include

  `(e.g., SampleClassName)`

- A class name is an identifier—a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces.
- Java is case sensitive—uppercase and lowercase letters are distinct—so `a1` and `A1` are different (but both valid) identifiers.

Braces

- A left brace, {, begins the body of every class declaration.
- A corresponding right brace, }, must end each class declaration.
- Code between braces should be indented.
- This indentation is one of the spacing conventions mentioned earlier.

Declaring the `main` Method

```
public static void main(String[] args)
```

- Starting point of every Java application.
- Parentheses after the identifier main indicate that it's a program building block called a method.
- Java class declarations normally contain one or more methods.
- JVM will not execute the application, if it does not have the `main` method.
- Methods perform tasks and can return information when they complete their tasks.
- Keyword void indicates that this method will not return any information.

Body of the Method Declaration

- Enclosed in left and right braces.
- Statements

  System.out.println("Hello World !!!");

  - Instructs the computer to perform an action
    - Print the string of characters contained between the double quotation marks.
  - A string is sometimes called a character string or a string literal.
  - White-space characters in strings are **not** ignored by the compiler.

Body of the Method Declaration (Cont.)

- System.out.println method
  - Displays (or prints) a line of text in the command window.
  - The string in the parentheses the argument to the method.
  - Positions the output cursor at the beginning of the next line in the command window.

Modifying Our First Java Program (Cont.)

- Newline characters indicate to System.out's print and println methods when to position the output cursor at the beginning of the next line in the command window.
- Newline characters are white-space characters.
- The backslash (\) is called an escape character.
  - Indicates a "special character"
- Backslash is combined with the next character to form an escape sequence.
- The escape sequence \n represents the newline character.

```
// HelloWorld2.java

public class HelloWorld2 {

      // method main executes the Java application
      public static void main (String[] args) {

            System.out.print("Hello\nWorld\n!!!\n");

      } // end method main

} // end class HelloWorld2
```

Displaying Text with printf

- System.out.printf method
  - f means “formatted”
  - displays formatted data
- Multiple method arguments are placed in a comma-separated list.
- Java allows large statements to be split over many lines.
  - Cannot split a statement in the middle of an identifier or string.
- Method printf’s first argument is a format string
  - May consist of fixed text and format specifiers.
  - Fixed text is output as it would be by print or println.
  - Each format specifier is a placeholder for a value and specifies the type of data to output.
- Format specifiers begin with a percent sign (%) and are followed by a character that represents the data type.
- Format specifier %s is a placeholder for a string.

```
// HelloWorld3.java

public class HelloWorld3 {

      // method main executes the Java application
      public static void main (String[] args) {

            System.out.printf("%s\n%s\n%s\n", "Hello ", "World ", "!!!");

      } // end method main

} // end class HelloWorld2
```

Escape Sequence

Escape Sequence	*Name*
`\t`	Tab
`\n`	Linefeed
`\\`	Backslash
`\"`	Double Quote

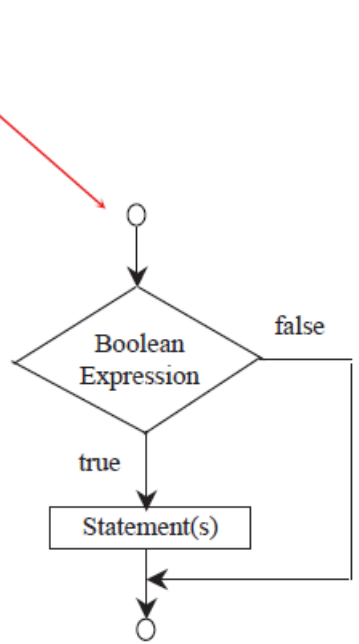
Relational Operators

Java Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	less than	`radius < 0`	`false`
`<=`	≤	less than or equal to	`radius <= 0`	`false`
`>`	>	greater than	`radius > 0`	`true`
`>=`	≥	greater than or equal to	`radius >= 0`	`true`
`==`	=	equal to	`radius == 0`	`false`
`!=`	≠	not equal to	`radius != 0`	`true`

Radius = 5

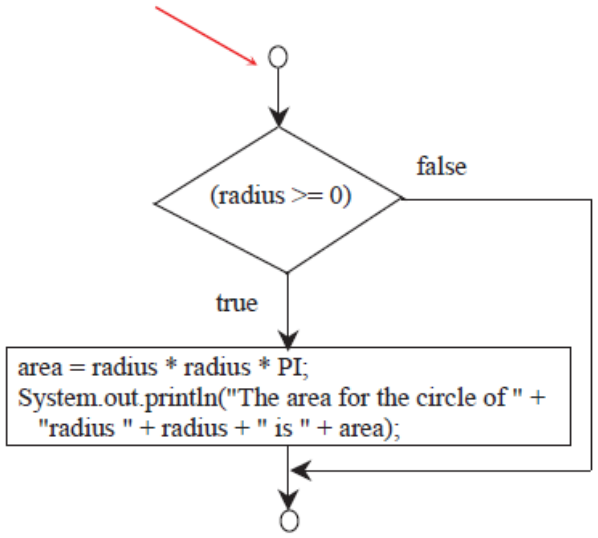
One-way `if` Statements

```
if (boolean-expression) {
  statement(s);
}
```



(A)

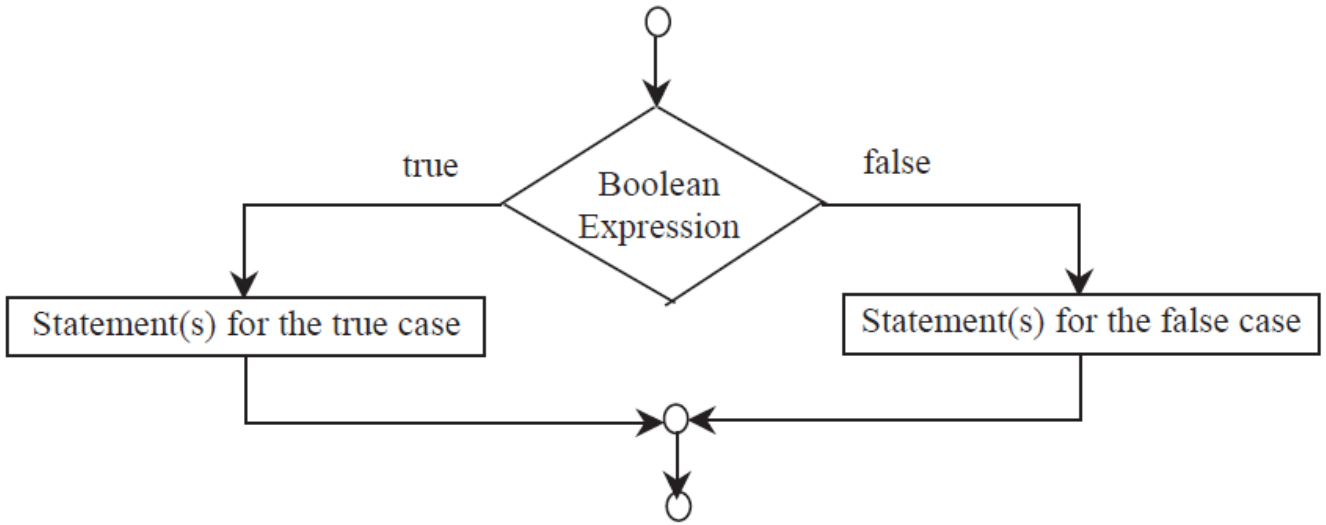
```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area"
    + " for the circle of radius "
    + radius + " is " + area);
}
```



(B)

The Two-way `if` Statement

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```



Conditional Operator

```
if (num % 2 == 0){
  System.out.println(num + "is even ");
}
else {
  System.out.println(num + "is odd");
}
```

if...else Example

```
if (radius >= 0) {
     area = radius * radius * 3.14159;

     System.out.println("The area for the “
       + “circle of radius " + radius +
       " is " + area);
}
else {
     System.out.println("Negative input");
}
```

Conditional Operator (Cont.)

```
if (x > 0)
   y = 1
else
   y = -1;
is equivalent to
y = (x > 0) ? 1 : -1;
(boolean-expression) ? expression1 : expression2


System.out.println((num % 2 == 0)?
   num + "is even" : num + "is odd");
```

Note

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

Multiple Alternative `if` Statements

```
if (score >= 90.0)
  grade = 'A';
else
  if (score >= 80.0)
    grade = 'B';
  else
    if (score >= 70.0)
      grade = 'C';
    else
      if (score >= 60.0)
        grade = 'D';
      else
        grade = 'F';
```

Equivalent

```
if (score >= 90.0)
  grade = 'A';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 60.0)
  grade = 'D';
else
  grade = 'F';
```

Note

- The else clause matches the most recent if clause in the same block.

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

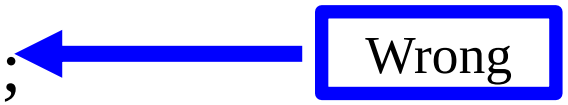
```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(b)

Common Errors

- Adding a semicolon at the end of an if clause is a common mistake.

```
if (radius >= 0);
{
        area = radius*radius*PI;
        System.out.println(
                "The area for the circle of radius " +
                radius + " is " + area);
}
```



- This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.

TIP

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

Equivalent

```
boolean even
  = number % 2 == 0;
```

(b)

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

Logical Operators

Operator	*Name*	*Description*
!	not	logical negation
&&	and	logical conjunction
||	or	logical disjunction
^	exclusive or	logical exclusion

Truth Table for Operator !

p	!p	*Example (assume* `age = 24, weight = 140`*)*
`true`	`false`	`!(age > 18)` is `false`, because `(age > 18)` is `true`.
`false`	`true`	`!(weight == 150)` is `true`, because `(weight == 150)` is `false`.

Truth Table for Operator &&

p_1	p_2	p_1 && p_2	*Example (assume* `age = 24, weight = 140`*)*
`false`	`false`	`false`	
`false`	`true`	`false`	`(age > 28) && (weight <= 140)` is `true`, because `(age > 28)` is `false`.
`true`	`false`	`false`	
`true`	`true`	`true`	`(age > 18) && (weight >= 140)` is `true`, because `(age > 18)` and `(weight >= 140)` are both `true`.

Truth Table for Operator ||

p_1	p_2	p_1 || p_2	*Example (assume* `age = 24, weight = 140`*)*
`false`	`false`	`false`	`(age > 34) || (weight >= 150)` is `false`, because `(age > 34)` and `(weight >= 150)` are both `false`.
`false`	`true`	`true`	
`true`	`false`	`true`	`(age > 18) || (weight < 140)` is `true`, because `(age > 18)` is `true`.
`true`	`true`	`true`	

Shortcut Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Increment and Decrement Operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

Increment and Decrement Operators (Cont.)

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

Increment and Decrement Operators (Cont.)

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables, or the same variable for multiple times such as this:

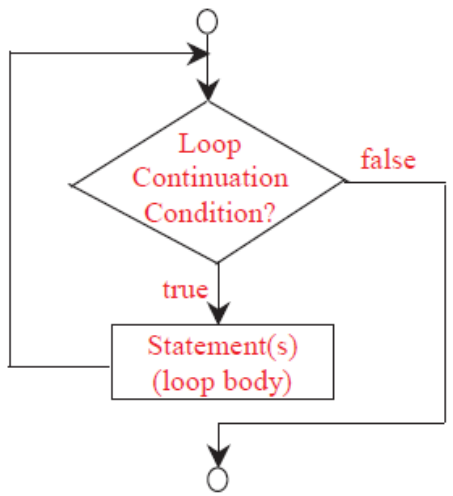
```
int k = ++i + i;
```

Introducing while Loops

```
int count = 0;
while (count < 100) {
        System.out.println("Welcome to Java");
        count++;
}
```

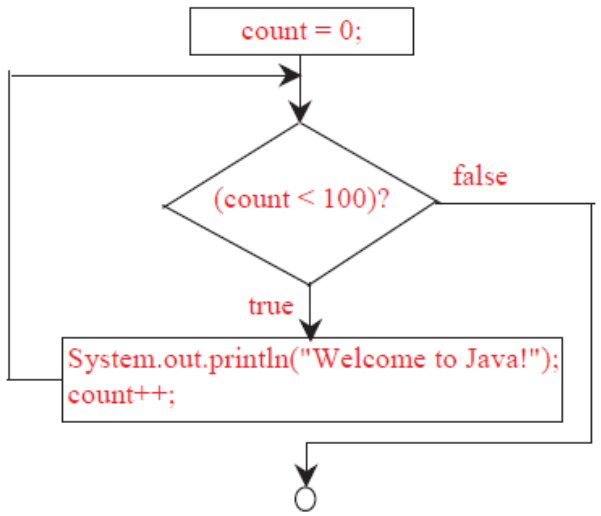
while Loop Flow Chart

```
while (loop-continuation-condition) {
  // loop-body;
  Statement(s);
}
```



(A)

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```



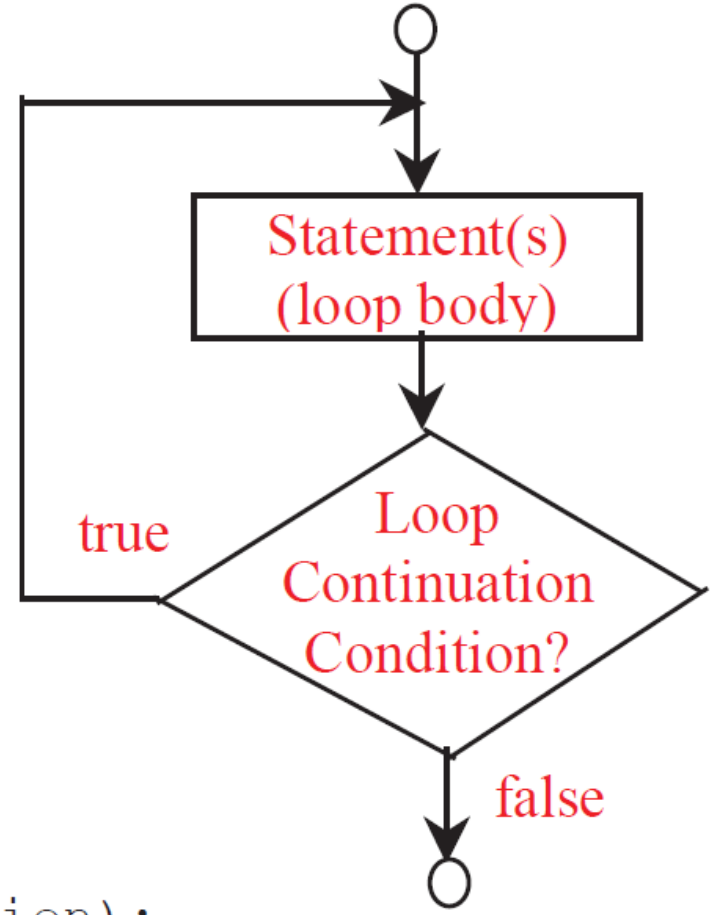
(B)

Ending a Loop with a Sentinel Value

Often the number of times a loop is executed is not predetermined. You may use an input value to signify the end of the loop. Such a value is known as a **sentinel** *value*.

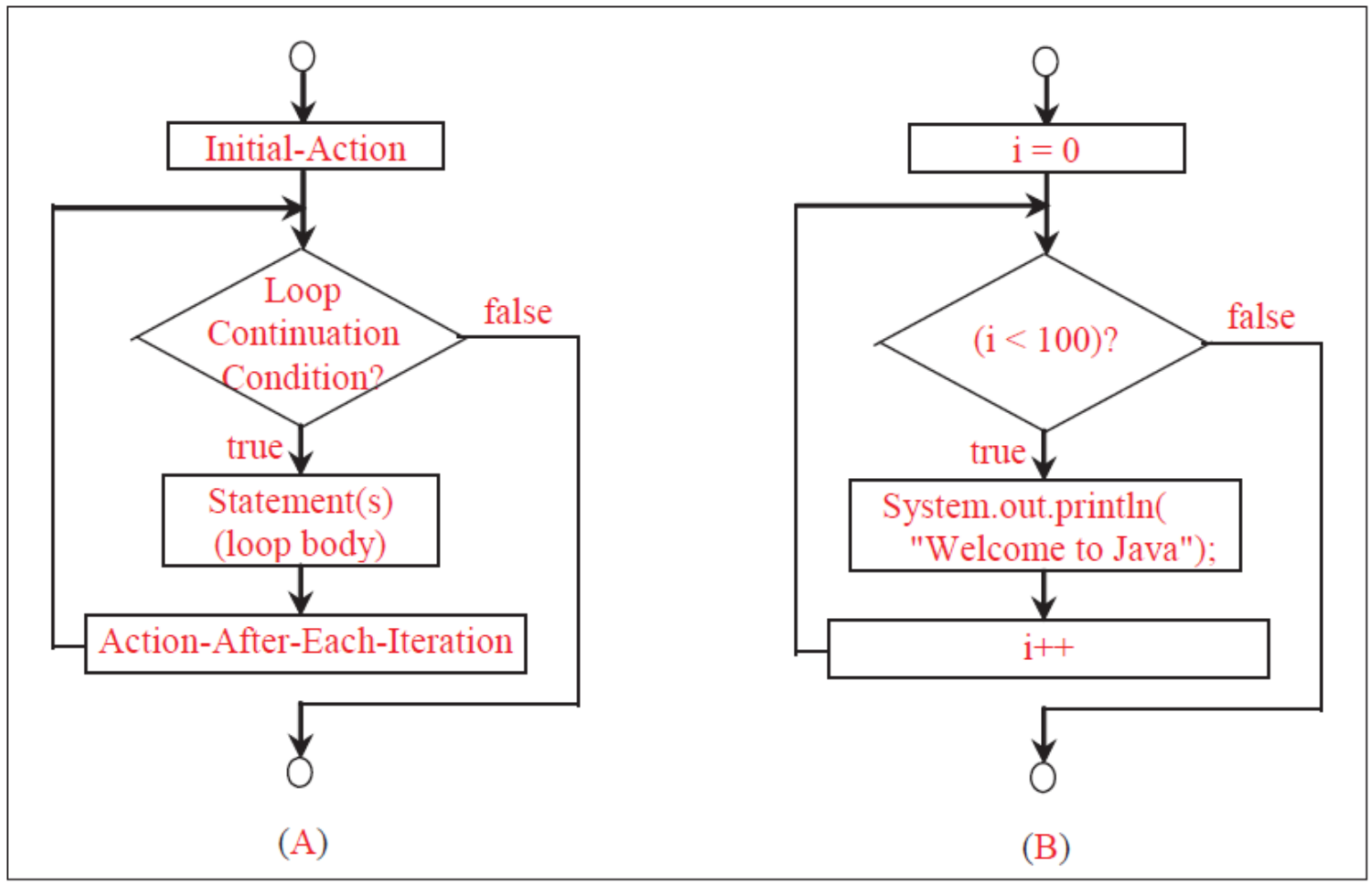
do-while Loop

```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```



```
for (initial-action; loop-
    continuation-condition;
    action-after-each-iteration) {
  // loop body;
  Statement(s);
}
```

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println(
    "Welcome to Java!");
}
```



Note

- The initial-action in a for loop can be a list of zero or more comma-separated expressions. The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. Therefore, the following two for loops are correct. They are rarely used in practice, however.

```
for (int i = 1; i < 100; System.out.println(i++));

for (int i = 0, j = 0; (i + j < 10); i++, j++) {
    // Do something
}
```

Note

- If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:

```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

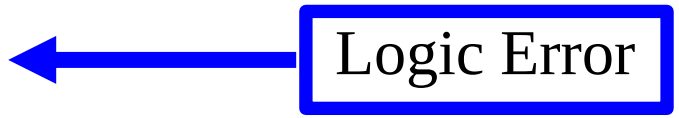
```
while (true) {
  // Do something
}
```

(b)

Caution

- Adding a semicolon at the end of the for clause before the loop body is a common mistake, as shown below:

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



Caution (Cont.)

Similarly, the following loop is also wrong:

```
int i=0;
while (i < 10);
{
     System.out.println("i is " + i);
     i++;
}
```

Logic Error

In the case of the do loop, the following semicolon is needed to end the loop.

```
int i=0;
do {
     System.out.println("i is " + i);
     i++;
} while (i<10);
```

Correct

Which Loop to Use?

- The three forms of loop statements, while, do-while, and for, are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a while loop in (a) in the following figure can always be converted into the following for loop in (b):

```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; )
  // Loop body
}
```

(b)

- A for loop in (a) in the following figure can generally be converted into the following while loop in (b) except in certain special cases.

```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration) {
  // Loop body;
}
```

(a)

Equivalent

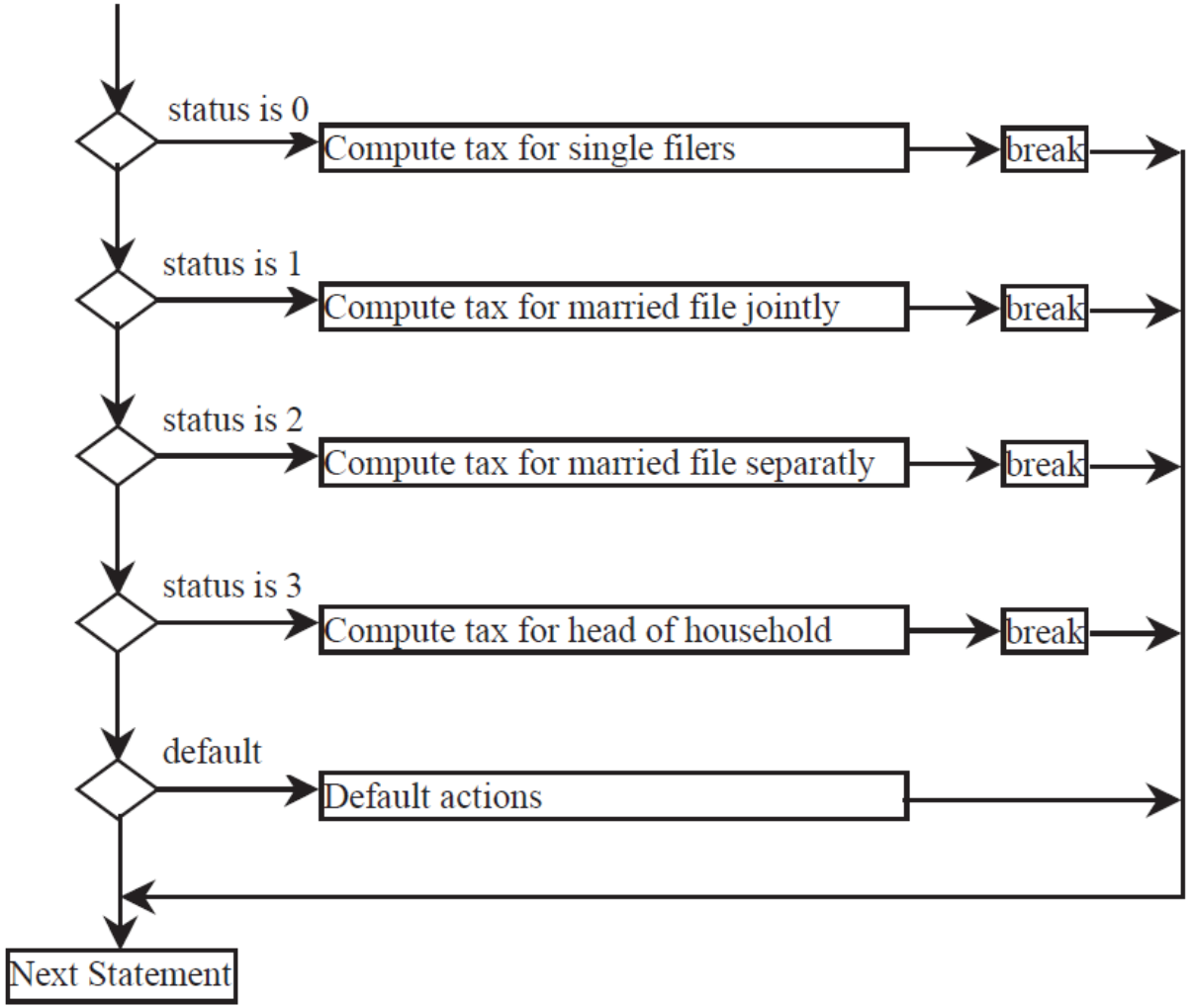
```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

switch Statements

```
switch (status) {
   case 0: do something here;
           break;
   case 1: do something here;
           break;
   case 2: do something here;
           break;
   case 3: do something here;
           break;
   default: System.out.println("Errors: invalid status");
           System.exit(0);
}
```

switch Statement Flow Chart



Type Casting

```
Implicit casting
    double d = 3; (type widening)
Explicit casting
    int i = (int)3.0; (type narrowing)
    int i = (int)3.9; (Fraction part is truncated)
```

What is wrong? `int x = 5 / 2.0;`