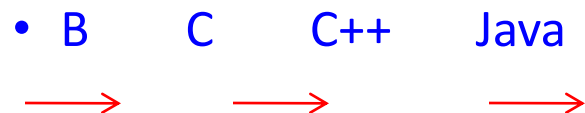


Java Origins

- Computer language innovation and development occurs for two fundamental reasons:
 - 1) to adapt to changing environments and uses
 - 2) to implement improvements in the art of programming
- The development of Java was driven by both in equal measures.

Many Java features are inherited from the earlier languages:



Before Java: C

- Designed by **Dennis Ritchie** in **1970s**.
- Before C, there was no language to reconcile: ease-of-use versus power,safety versus efficiency, rigidity versus extensibility.
- BASIC, COBOL, FORTRAN, PASCAL optimized one set of traits, but not the other.
- C- structured, efficient, high-level language that could replace assembly code when creating systems programs.
- Designed, implemented and tested by programmers, not scientists.

Before Java: C++

- Designed by **Bjarne Stroustrup** in **1979**.
- Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:
 - 1) assembler languages
 - 2) high-level languages
 - 3) structured programming
 - 4) object-oriented programming (OOP)
- OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.
- C++ extends C by adding object-oriented features.

Java History

- Originally Java was designed for Interactive television, but this technology was very much advanced for the industry of digital cable television at that time.
- Java history was started with the Green Team.
- The Green Team started a project to develop a language for digital devices such as television.
- But it works best for internet programming. After some time Java technology was joined by Netscape.
- Initially, Java was called "Greentalk" by James Gosling and at that time the file extension was .gt.

Java History

- Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at [Sun Microsystems in 1991](#).
- The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.
- With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.
- Java as an “Internet version of C++”? No.
- Java was not designed to replace C++, but to solve a different set of problems. There are significant practical/philosophical differences.

Introduction to Java

- Java programming language is a high-level, object-oriented, general-purpose, and secure programming language. It was developed by James Gosling at Sun Microsystems in 1991. At that time, they called it OAK.
- Sun Microsystem changed the name to Java in 1995. In 2009, Oracle Corporation took over Sun Microsystem.
- Java is the most widely used programming language. It is designed for the distributed environment of the Internet. Java is freely accessible to users, and we can run it on all the platforms. Java follows the **WORA** (Write Once, Run Anywhere) principle, and is platform-independent.

Editions of Java

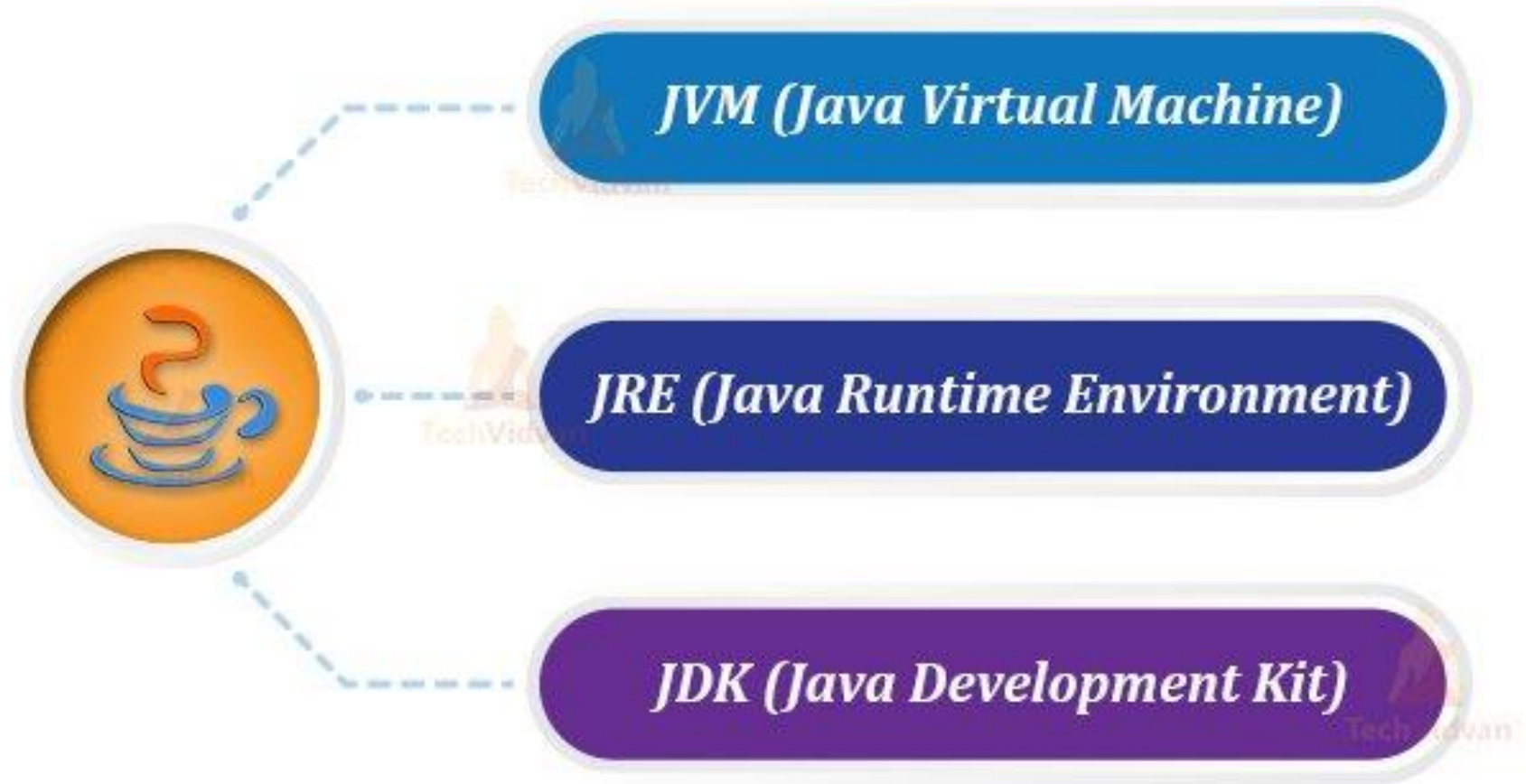
There are three editions of Java. Each Java edition has different capabilities. The editions of Java are:

1. Java Standard Editions (SE): We use this edition to create programs for a desktop computer.
2. Java Enterprise Edition (EE): We use this edition to create large programs that run on the server and to manage heavy traffic and complex transactions.
3. Java Micro Edition (ME): We use this edition to develop applications for small devices such as set-top boxes, phones, and appliances, etc

Java Technology

- There is more to Java than the language.
- Java Technology consists of:
 - 1) Java Programming Language
 - 2) Java Virtual Machine (JVM)
 - 3) Java Application Programming Interfaces (APIs)

Java Environment



Java Language Features

- 1) simple
- 2) object-oriented
- 3) robust
- 4) multithreaded
- 5) architecture-neutral
- 6) interpreted and high-performance
- 7) distributed
- 8) dynamic
- 9) secure

Java Language Features

- 1) **simple** – Java is simple because its syntax is simple and easy to understand. Java is designed to be easy for the professional programmer to learn and use.
- 2) **object-oriented** –Everything in Java is in the form of the object. In other words, it has some data and behaviour. A Java program must have at least one class and object. a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
- 3) **robust** –Java always tries to check errors at runtime and compile time. Java uses a garbage collector to provide a strong memory management system. Features like Exception handling and garbage collection make Java robust or strong. restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.

Java Language Features

- 4) **multithreaded** – supports multi-threaded programming for writing program that perform concurrent computations
- 5) **architecture-neutral** – Java Virtual Machine provides a platform independent environment for the execution of Java bytecode
- 6) **interpreted and high-performance** – Java programs are compiled into an intermediate representation – bytecode:
 - a) can be later interpreted by any JVM
 - b) can be also translated into the native machine code for efficiency.

Java Language Features

- 7) **distributed** – Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
- 8) **dynamic** – substantial amounts of run-time type information to verify and resolve access to objects at run-time.
- 9) **secure** – : Java is a secure language, as Java does not use explicit pointers. All Java programs run in the virtual machine. Moreover, Java contains a security manager that defines the access levels of Java classes. programs are confined to the Java execution environment and cannot access other parts of the computer.

Applications of Java Programming

Java is a widespread language. The following are some application areas in which we find Java usable:

- Desktop applications
- Web applications
- Mobile applications (Android)
- Cloud computing
- Enterprise applications
- Scientific applications
- Operating Systems
- Embedded systems
- Cryptography
- Smart cards
- Computer games
- Web servers and application servers

Execution Platform

What is an execution platform?

- 1) An execution platform is the hardware or software environment in which a program runs, e.g. [Windows 2000](#), [Linux](#), [Solaris](#) or [MacOS](#).
- 2) Most platforms can be described as a combination of the operating system and hardware.

Java Execution Platform

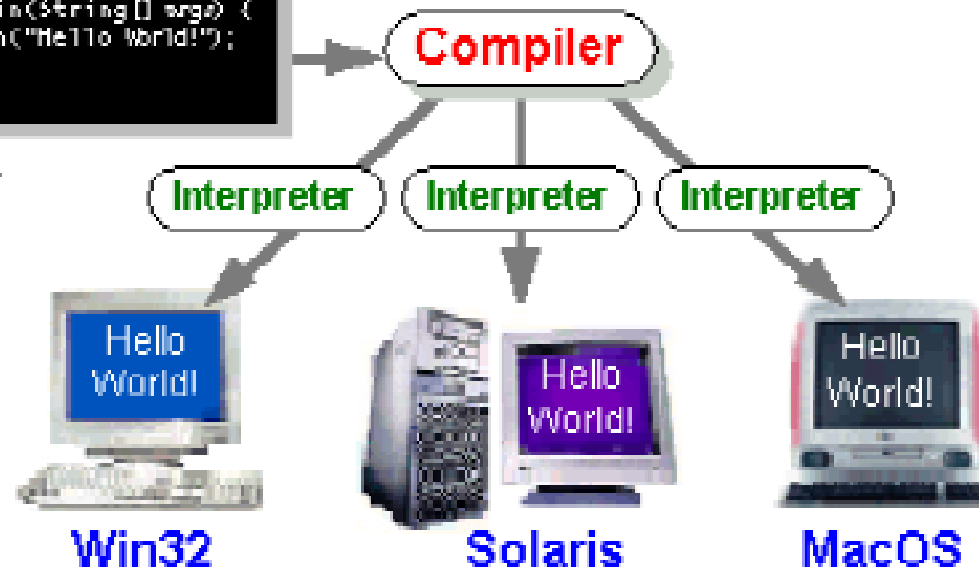
- What is Java Platform?
- 1) A software-only platform that runs on top of other hardware-based platforms.
- 2) Java Platform has two components:
 - a) **Java Virtual Machine (JVM)** – interpretation for the Java bytecode, ported onto various hardware-based platforms.
 - b) The Java **Application Programming Interface (Java API)**

Java Platform Independence

Java Program

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

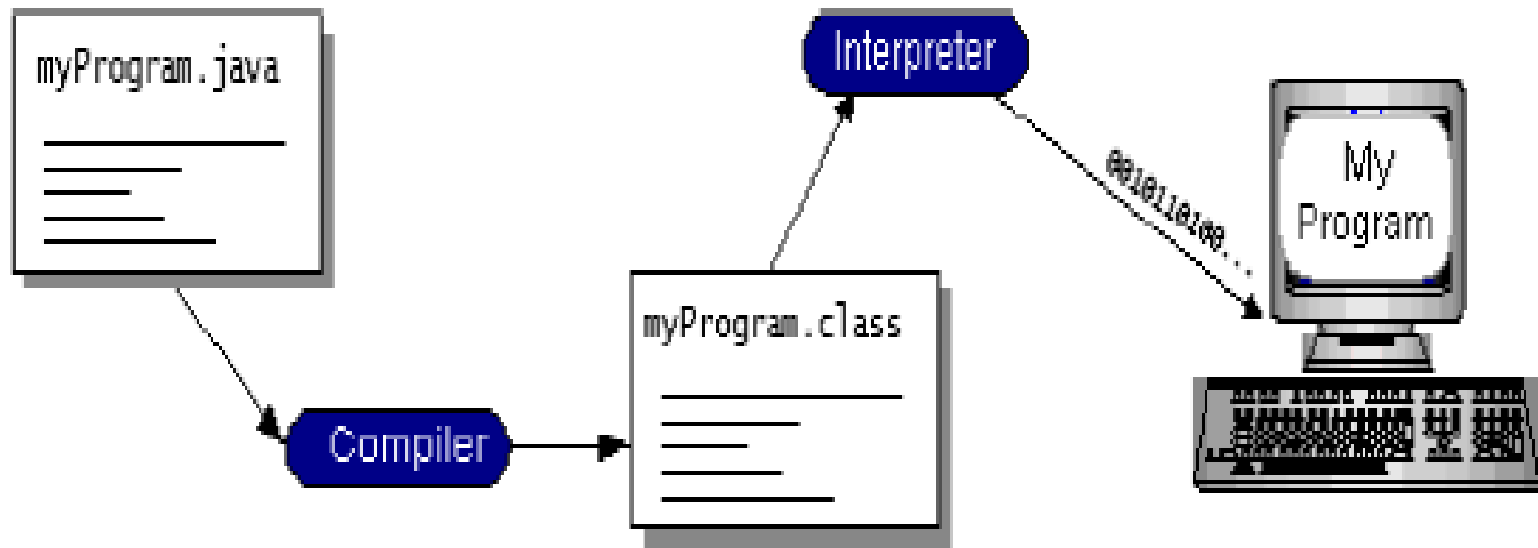
HelloWorldApp.java



Java Program Execution

- Java programs are both compiled and interpreted:
- Steps:
 - write the Java program
 - compile the program into bytecode
 - execute (interpret) the bytecode on the computer through the Java Virtual Machine
- Compilation happens once.
- Interpretation occurs each time the program is executed.

Java Execution Process



Java API

- What is Java API?
 - 1) a large collection of ready-made software components that provide many useful capabilities, e.g. graphical user interface
 - 2) grouped into **libraries (packages)** of related classes and interfaces
 - 3) together with JVM insulates Java programs from the hardware and operating system variations

Java Program Types

- Types of Java programs:
 - 1) **applications** – standalone (desktop) Java programs, executed from the command line, only need the Java Virtual Machine to run
 - 2) **applets** – Java program that runs within a Java-enabled browser, invoked through a “applet” reference on a web page, dynamically downloaded to the client computer
 - 3) **servlets** – Java program running on the web server, capable of responding to HTTP requests made through the network

Simple Java Program

A class to display a simple message:

```
class MyProgram {  
    public static void main(String args []) {  
        System.out.println("First Java program.");  
    }  
}
```

Running the Program

- Type the program, save as `MyProgram.java`.
In the command line, type:
 `> dir`
 `MyProgram.java`
 `> javac MyProgram.java`
- `> dir`
 `MyProgram.java, MyProgram.class`
 `> java MyProgram`
 First Java program.

Explaining the Process

- 1) **creating a source file** - a source file contains text written in the Java programming language, created using any text editor on any system.
- 2) **compiling the source file** - **Java compiler (javac)** reads the source file and translates its text into instructions that the Java interpreter can understand. These instructions are called **bytecode**.
- 3) **running the compiled program** - **Java interpreter (java)** installed takes as input the **bytecode** file and carries out its instructions by translating them on the fly into instructions that your computer can understand.

Java Program Explained

MyProgram is the name of the class:

```
class MyProgram {
```

//main is the method with one parameter **args** and no
//results:

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

println is a method in the standard **System** class:

```
System.out.println("First Java program.");
```

```
}
```

```
}
```

Main Method

The `main` method must be present in every Java application:

1) `public static void main(String[] args)` where:

a) `public` : keyword is an access modifier which represents visibility. It means it is visible to all. means that the method can be called by any object

b) `static` is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.

c) `void` means that the method does not return any value

- 2) When the interpreter executes an application, it starts by calling its `main` method which in turn invokes other methods in this or other classes.
- 3) The main method accepts a single argument – `a string array`, which holds all command-line parameters.

Exercise: Java Program

- 1) Personalize the `MyProgram` program with your name so that it tells you

“Hello, my name is ...”

- 2) Write a program that produces the following output:

Welcome to MES College Java Workshop!
I hope you will benefit from the training.

- 3) Here is a slightly modified version of `MyProgram`:

- ```
class MyProgram2 {
 public void static Main(String args) {
 system.out.println("First Java Program!");
 }
}
```
- The program has some errors. Fix the errors so that the program successfully compiles and runs. What were the errors?

# Classes and Objects

- A **class** is the basic building block of Java programs.  
A **class** encapsulates:
  - a) **data (attributes)** and
  - b) operations on this data (**methods**)and permits to create **objects** as its instances.

# Java Syntax

- On the most basic level, Java programs consist of:
  - a) whitespaces
  - b) identifiers
  - c) comments
  - d) literals
  - e) separators
  - f) keywords
  - g) operators
- Each of them will be described in order.

# Whitespaces

- A whitespace is a **space, tab or new line**.
- Java is a form-free language that does not require special indentation.
- A program could be written like this:

```
class MyProgram {
 public static void main(String[] args) {
 System.out.println("First Java program.");
 }
}
```

- It could be also written like this:
- `class MyProgram { public static void main(String[] args)`
- `{ System.out.println("First Java program."); } }`

# Identifiers

- Java identifiers:
  - a) used for class names, method names, variable names
  - b) an identifier is any sequence of letters, digits, “\_” or “\$” characters that do not begin with a digit
  - c) Java is case sensitive, so **value**, **Value** and **VALUE** are all different.
- Seven identifiers in this program:

```
class MyProgram {
 public static void main(String[] args) {
 System.out.println("First Java program.");
 }
}
```



# Comments

- Three kinds of comments:

1) Ignore the text between `/*` and `*/`:

`/* text */`

2) Documentation comment (`javadoc` tool uses this kind of comment to automatically generate software documentation):

`/** documentation */`

3) Ignore all text from `//` to the end of the line:

`// text`

# Comments

- ```
/**  
 * MyProgram implements application that displays  
 * a simple message on the standard output device.  
 */  
class MyProgram {  
    /* The main method of the class.*/  
    public static void main(String[] args) {  
        //display string  
        System.out.println("First Java program.");  
    }  
}
```

Literals

- A literal is a constant value of certain type.
- It can be used anywhere values of this type are allowed.
- Examples:

a) 100

b) 98.6

c) 'X'

d) "test"

```
class MyProgram {  
    public static void main(String[] args) {  
        System.out.println("My first Java program.");  
    }  
}
```

Literals

- Literals express constant values.
- The form of a literal depends on its type:

1) integer types

2) floating-point types

3) character type

4) boolean type

5) string type

Literals: Integer Types

Writing numbers with different bases:

1) decimal – 123

2) octal – 0173

3) hexadecimal – 0x7B

Integer literals are of type `int` by default.

Integer literal written with “L” (e.g. 123L) are of type `long`.

Literals: Floating-Point Types

Two notations:

- 1) standard – 2000.5
- 2) scientific – 2.0005E3

Floating-point literals are of type `double` by default.

Floating-point literal written with “F”

(e.g. 2.0005E3F) are of type `float`.

Literals: Boolean

Two literals are allowed only: `true` and `false`.

Those values do not convert to any numerical representation.

In particular:

- 1) `true` is not equal to `1`
- 2) `false` is not equal to `0`

Literals: Characters

- Character literals belong to the Unicode character set.
- Representation:
 - 1) visible characters inside quotes, e.g. 'a'
 - 2) invisible characters written with **escape sequences**:
 - a) \ddd octal character ddd
 - b) \uxxxx hexadecimal Unicode character xxxx
 - c) \' single quote
 - d) \" double quote
 - e) \\ backslash
 - f) \r carriage return
 - g) \n new line
 - h) \f form feed
 - i) \t tab
 - j) \b backspace

Literals: String

- String is not a simple type.
- String literals are character-sequences enclosed in double quotes.
- Example:

“Hello World!”

Notes:

- 1) escape sequences can be used inside string literals
- 2) string literals must begin and end on the same line
- 3) unlike in C/C++, in Java `String` is not an array of characters

Separators

()	parenthesis	lists of parameters in method definitions and invocations, Precedence in expressions, type casts
{ }	braces	block of code, class definitions, method definitions, local scope, automatically initialized arrays
[]	Brackets	declaring array types, referring to array values
;	semicolon	terminating statements, chain statements inside the “for” statement
,	comma	separating multiple identifiers in a variable declaration
.	period	separate package names from subpackages and classes, separating an object variable from its attribute or method

Keywords

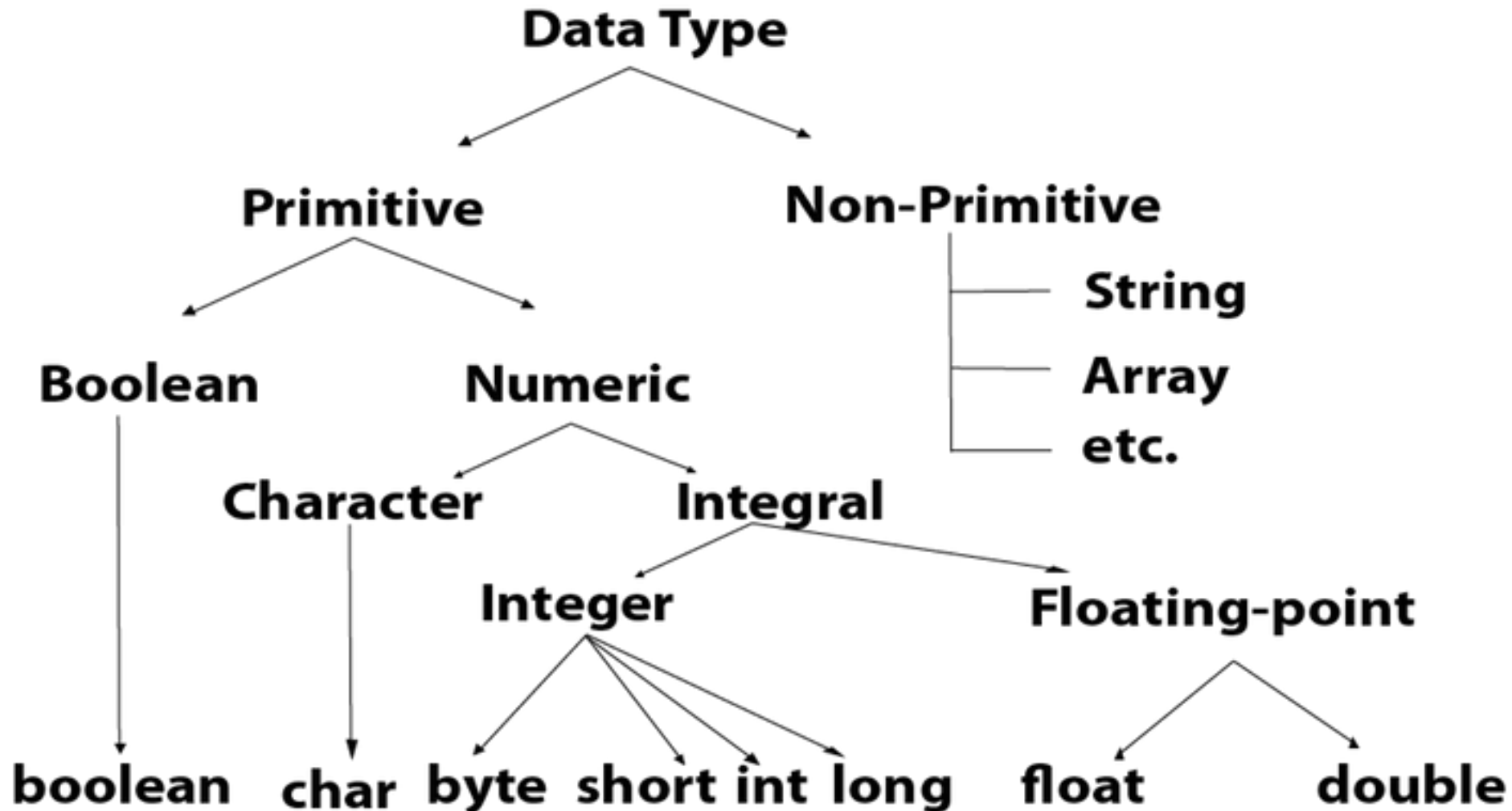
- Keywords are reserved words recognized by Java that cannot be used as identifiers. Java defines 49 keywords as follows:

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

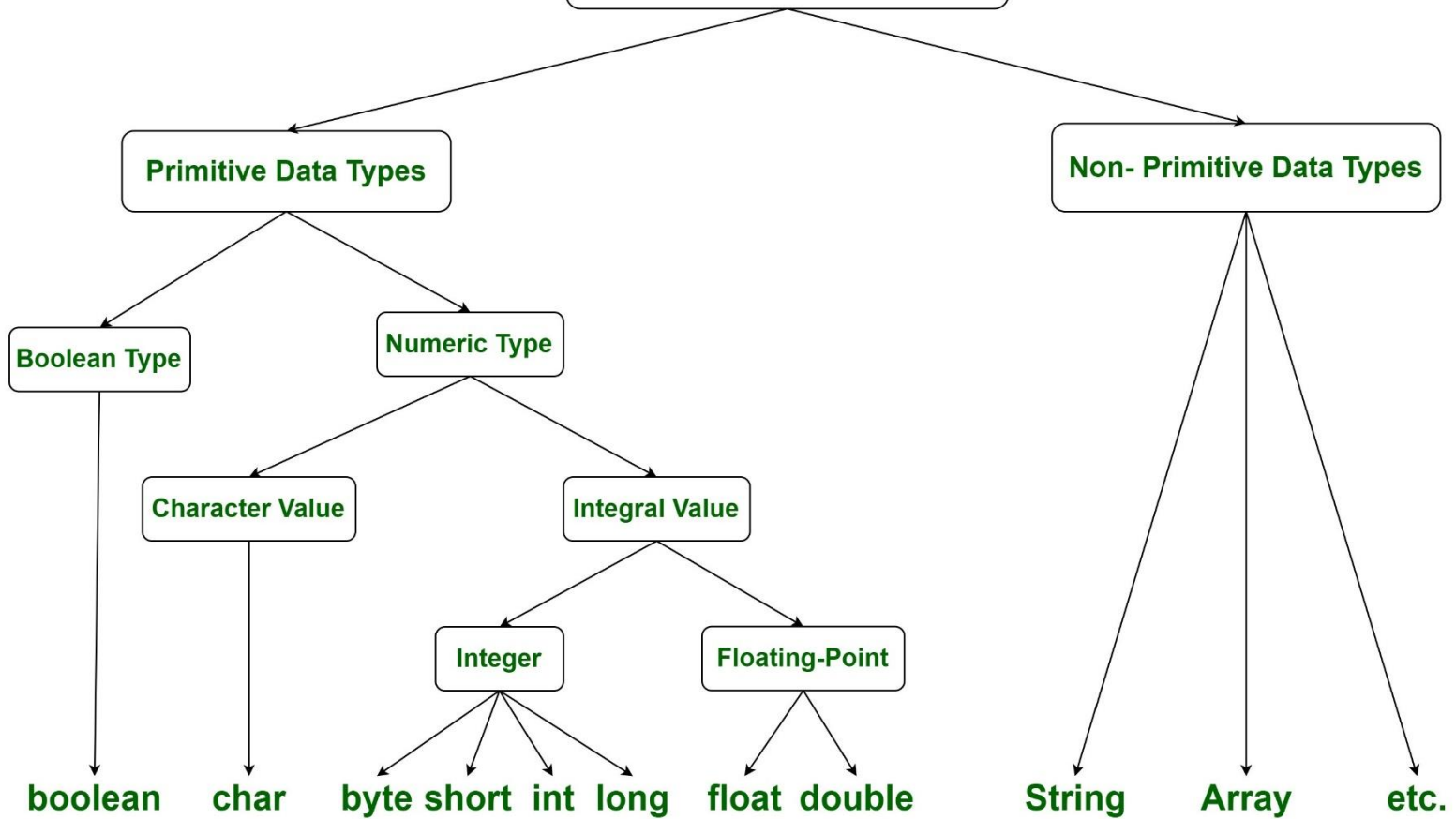
Strong Typing

- Java is a strongly-typed language:
 - a) every variable and expression has a type
 - b) every type is strictly defined
 - c) all assignments are checked for type-compatibility
 - d) no automatic conversion of non-compatible, conflicting types
 - e) Java compiler type-checks all expressions and parameters
 - f) any typing errors must be corrected for compilation to succeed

Simple Types



Data Types in Java



Simple Types

- Java defines eight simple types:
 - 1) `byte` – 8-bit integer type
 - 2) `short` – 16-bit integer type
 - 3) `int` – 32-bit integer type
 - 4) `long` – 64-bit integer type
 - 5) `float` – 32-bit floating-point type
 - 6) `double` – 64-bit floating-point type
 - 7) `char` – symbols in a character set
 - 8) `boolean` – logical values `true` and `false`

TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\', '\n', ' \b'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

Simple Type: byte

8-bit integer type.

Syntax:

```
byte byteVar;
```

Range: -128 to 127.

Example:

```
byte b = -15;
```

Usage: particularly when working with data streams.

Simple Type: short

16-bit (2 byte) integer type.

Syntax:

```
short shortVar;
```

Range: -32768 to 32767.

Example:

```
short c = 1000;
```

Usage: probably the least used simple type.

Simple Type: int

32-bit integer type.

Syntax:

```
int intVariable;
```

Range: -2147483648 to 2147483647.

Example: `int b = -50000;`

Usage:

- 1) Most common integer type.
- 2) Typically used to control loops and to index arrays.
- 3) Expressions involving the `byte`, `short` and `int` values are `promoted` to `int` before calculation.

Simple Type: long

The range of a long is quite large.

64-bit (8 byte) integer type.

Syntax:

```
long longVariable;
```

Range: -9223372036854775808 to
9223372036854775807.

Example:

```
long lo = 1000000000000000000;
```

Usage:

1) useful when **int** type is not large enough to hold the desired value

Simple Type: float

32-bit (4 byte) floating-point number.

Syntax:

```
float floatVar;
```

Values: upto 7 decimal digits

Range: 1.4e-045 to 3.4e+038.

Example:

```
float f = 1.5;
```

Usage:

- 1) fractional part is needed
- 2) large degree of precision is not required

Simple Type: double

64-bit (8 byte) floating-point number.

Syntax:

```
double doubleVar;
```

Values: Up to 16 decimal digits

Range: 4.9e-324 to 1.8e+308.

Example:

```
double pi = 3.1416;
```

Usage:

- 1) accuracy over many iterative calculations
- 2) manipulation of large-valued numbers

Example: double

- `// Compute the area of a circle.`
- `class Area {`
 `public static void main(String args[]) {`
 `double pi = 3.1416; // approximate pi value`
 `double r = 10.8; // radius of circle`
 `double a = pi * r * r; // compute area`
 `System.out.println("Area of circle is " + a);`
 `}`
- `} O/p =Area of circle is 366.436224`

Simple Type: char

16-bit data type used to store characters.

Syntax:

```
char charVar;
```

Range: 0 to 65536.

Example:

```
char c = 'a';
```

Usage:

- 1) Represents both ASCII and Unicode character sets; Unicode defines a character set with characters found in (almost) all human languages.
- 2) Not the same as in C/C++ where `char` is 8-bit and represents ASCII only.

Example: char

// Demonstrate char data type.

```
class CharDemo {  
    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);  
    }  
}
```

O/p ch1 and ch2 : X Y

Another Example: char

It is possible to operate on `char` values as if they were integers:

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

Simple Type: boolean

Two-valued type of logical values.

Range: values `true` and `false`.

Example:

```
boolean b = (1<2);
```

Usage:

- 1) returned by relational operators, such as `1<2`
- 2) required by branching expressions such as `if` or `for`

Example: boolean

```
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);  
        if (b) System.out.println("executed");  
            b = false;  
        if (b) System.out.println("not executed");  
            System.out.println("10 > 9 is " + (10 > 9));  
    }  
}
```

Variables

- 1) **declaration** – how to assign a type to a variable
- 2) **initialization** – how to give an initial value to a variable
- 3) **scope** – how the variable is visible to other parts of the program
- 4) **lifetime** – how the variable is created, used and destroyed
- 5) **type conversion** – how Java handles automatic type conversion
- 6) **type casting** – how the type of a variable can be narrowed down
- 7) **type promotion** – how the type of a variable can be expanded

Basic Variable Declaration

- form of variable declaration: Basic

Variable data type must be one of the following:
-A simple data type;
-A user defined data type called a class type;

Optional initial value

`dataType` `identifier` `[=value] ;`

Variable identifier must
-Confirm to identifier rules;

Variable Declaration

- We can declare several variables at the same time:

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

- Examples:

```
int a, b, c;
```

```
int d = 3, e, f = 5;
```

```
byte hog = 22;
```

```
double pi = 3.14159;
```

```
char kat = 'x';
```

Constant Declaration

- A variable can be declared as final:

`final double PI = 3.14;`

- The value of the final variable cannot change after it has been initialized:

~~`PI = 3.13;`~~

Variable Identifiers

- Identifiers are assigned to variables, methods and classes.
- An identifier:
 - 1) starts with a letter, underscore `_` or dollar `$`
 - 2) can contain letters, digits, underscore or dollar characters
 - 3) it can be of any length
 - 4) it must not be a keyword (e.g. `class`)
 - 5) it must be unique in its scope
- Examples: `identifier`, `userName`, `_sys_var1`, `$change`
- The code of Java programs is written in Unicode, rather than ASCII, so letters and digits have considerably wider definitions than just a-z and 0-9.

Naming Conventions

- Conventions are not part of the language.
- Naming conventions:
 - 1) variable names begin with a lowercase letter
 - 2) class names begin with an uppercase letter
 - 3) constant names are all uppercase
- If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter.
- The underscore character is used only to separate words in constants, as they are all caps and thus cannot be case-delimited.

Variable Initialization

During declaration, variables may be optionally initialized.

Initialization can be static or dynamic:

1) static initialize with a literal:

```
int n = 1;
```

2) dynamic initialize with an expression composed of any literals, variables or method calls available at the time of initialization:

```
int m = n + 1;
```

The types of the expression and variable must be the same.

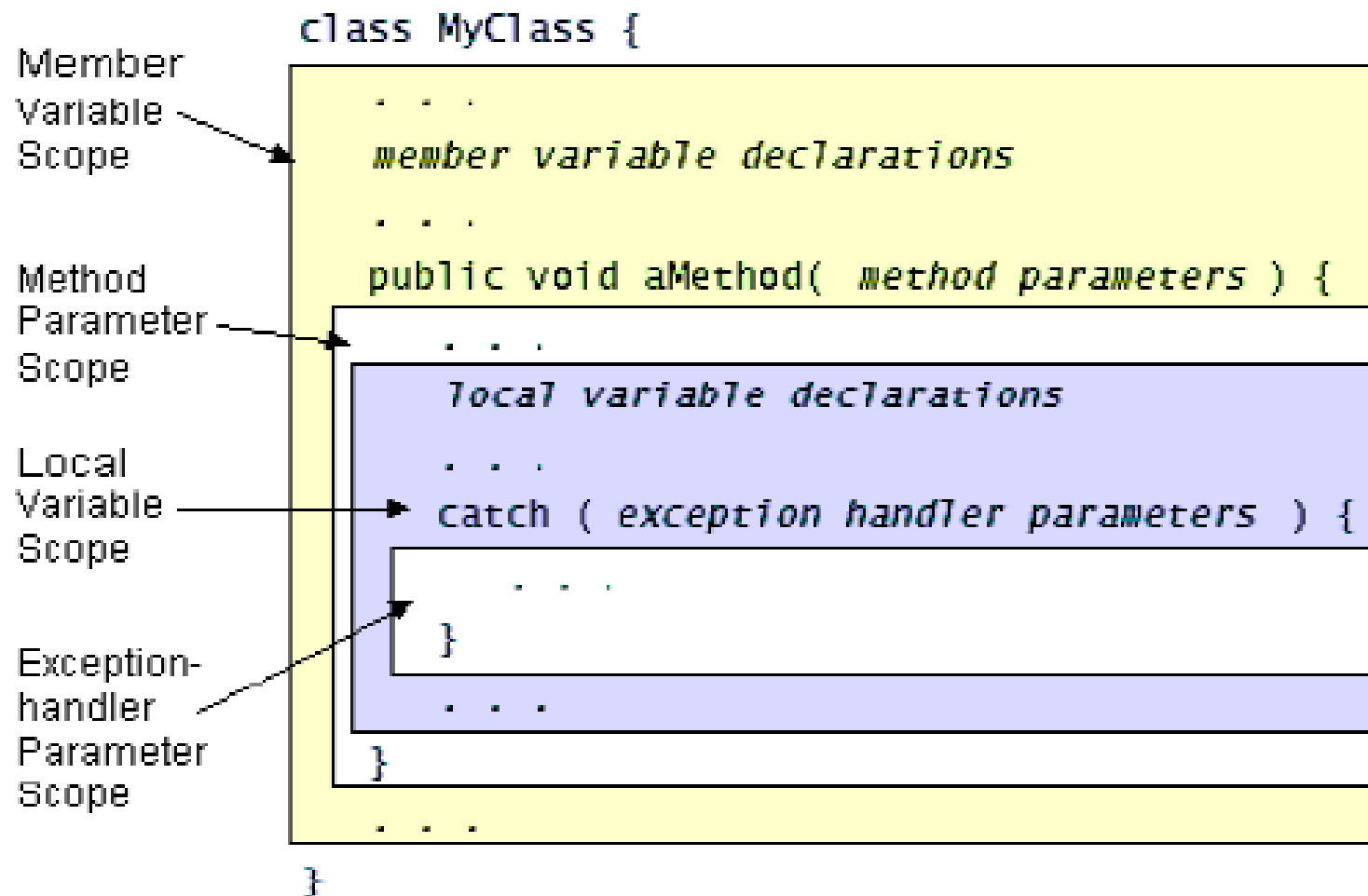
Example: Variable Initialization

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

Variable Scope

- Scope determines the visibility of program elements with respect to other program elements.
- In Java, scope is defined separately for classes and methods:
 - 1) variables defined by a class have a “global” scope
 - 2) variables defined by a method have a “local” scope
- We consider the scope of method variables only; class variables will be considered later.

Variable Scope




Scope Definition

- A scope is defined by a block:

```
{  
    ...  
}
```

- A variable declared inside the scope is not visible outside:

```
{  
    int n;  
}  
n = 1;
```



Example: Variable Scope

- ```
class Scope {
 public static void main(String args[]) {
 int x;
 x = 10;
 if (x == 10) {
 int y = 20;
 System.out.println("x and y: " + x + " " + y);
 x = y * 2;
 }
 System.out.println("x is " + x + "y is" + y);
 }
}
```
- }



# Variable Lifetime

- Variables are created when their scope is entered by control flow and destroyed when their scope is left:
  - 1) A variable declared in a method will not hold its value between different invocations of this method.
  - 2) A variable declared in a block loses its value when the block is left.
  - 3) Initialized in a block, a variable will be re-initialized with every re-entry.
- Variables lifetime is confined to its scope!

# Example: Variable Lifetime

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

# Type Differences

- Suppose a value of one type is assigned to a variable of another type.

T1 t1;

T2 t2 = t1;

- What happens? Different situations:
  - 1) types T1 and T2 are incompatible
  - 2) types T1 and T2 are compatible:
    - a) T1 and T2 are the same
    - b) T1 is larger than T2
    - c) T2 is larger than T1

# Type Compatibility

- When types are compatible:
  - 1) integer types and floating-point types are compatible with each other
  - 2) numeric types are not compatible with `char` or `boolean`
  - 3) `char` and `boolean` are not compatible with each other
- Examples:

```
byte b;
```

```
int i = b;
```

```
char c = b;
```

# Widening Type Conversion

- Java performs automatic type conversion when:
  - 1) two types are compatible
  - 2) destination type is larger than the source type
- Example:

```
int i;
```

```
double d = i;
```

# Narrowing Type Conversion

- When:
  - 1) two types are compatible
  - 2) destination type is smaller than the source type then Java will not carry out type-conversion:

```
int i;
```

```
byte b = i;
```

- Instead, we have to rely on manual type-casting:

```
int i;
```

```
byte b = (byte)i;
```

# Type Promotion Rules

- 1) byte and short are always promoted to int
  - 2) if one operand is long , the whole expression is promoted to long
  - 3) if one operand is float ,the entire expression is promoted to float
  - 4) if any operand is double , the result is double
- Danger of automatic type promotion:  

```
byte b = 50;
b = b * 2;
```
  - What is the problem?

# Example: Type Promotion

- ```
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("result = " + result);  
    }  
}
```


Operators Types

- Java operators are used to build value expressions.
- Java provides a rich set of operators:
 - 1) assignment
 - 2) arithmetic
 - 3) relational
 - 4) logical
 - 5) bitwise
 - 6) other

Assignment Operator

A binary operator:

```
variable = expression;
```

It assigns the value of the expression to the variable.

The types of the variable and expression must be compatible.

The value of the whole assignment expression is the value of the expression on the right, so it is possible to chain assignment expressions as follows:

```
int x, y, z;
```

```
x = y = z = 2;
```

Arithmetic Operators

- Java supports various arithmetic operators for:
 - 1) integer numbers
 - 2) floating-point numbers
- There are two kinds of arithmetic operators:
 - 1) **basic**: addition, subtraction, multiplication, division and modulo
 - 2) **shortcut**: arithmetic assignment, increment and decrement

Table: Basic Arithmetic Operators

<code>+</code>	<code>op1 + op2</code>	adds <code>op1</code> and <code>op2</code>
<code>-</code>	<code>op1 - op2</code>	subtracts <code>op2</code> from <code>op1</code>
<code>*</code>	<code>op1 * op2</code>	multiplies <code>op1</code> by <code>op2</code>
<code>/</code>	<code>op1 / op2</code>	divides <code>op1</code> by <code>op2</code>
<code>%</code>	<code>op1 % op2</code>	computes the remainder of dividing <code>op1</code> by <code>op2</code>

Arithmetic Assignment Operators

- Instead of writing

`variable = variable operator expression;`

- for any arithmetic binary operator, it is possible to write shortly

`variable operator= expression;`

- Benefits of the assignment operators:

1) `save some typing`

2) `are implemented more efficiently by the Java run-time system`

Table: Arithmetic Assignments

<code>+=</code>	<code>v += expr;</code>	<code>v = v + expr;</code>
<code>-=</code>	<code>v -= expr;</code>	<code>v = v - expr;</code>
<code>*=</code>	<code>v *= expr;</code>	<code>v = v * expr;</code>
<code>/=</code>	<code>v /= expr;</code>	<code>v = v / expr;</code>
<code>%=</code>	<code>v %= expr;</code>	<code>v = v % expr;</code>

Increment/Decrement Operators

- Two unary operators:
 - 1) `++` increments its operand by 1
 - 2) `--` decrements its operand by 1
- The operand must be a numerical variable.
- Each operation can appear in two versions:
- **prefix** version evaluates the value of the operand **after** performing the increment/decrement operation
- **postfix** version evaluates the value of the operand **before** performing the increment/decrement operation

Table: Increment/Decrement

<code>++</code>	<code>v++</code>	return value of <code>v</code> , then increment <code>v</code>
<code>++</code>	<code>++v</code>	increment <code>v</code> , then return its value
<code>--</code>	<code>v--</code>	return value of <code>v</code> , then decrement <code>v</code>
<code>--</code>	<code>--v</code>	decrement <code>v</code> , then return its value

Example: Increment/Decrement

```
class IncDec {  
    public static void main(String args[]) {  
        int a = 1; int b = 2; int c, d;  
        c = ++b;  
        d = a++;  
        c++;  
        System.out.println("a= " + a);  
        System.out.println("b= " + b);  
        System.out.println("c= " + c);  
    }  
}
```

Relational Operators

- Relational operators determine the relationship that **one operand** has to the **other operand**, specifically equality and ordering.
- The outcome is always a value of type **boolean**.
- They are most often used in branching and loop control statements.

Table: Relational Operators

<code>==</code>	equals to	apply to any type
<code>!=</code>	not equal to	apply to any type
<code>></code>	greater than	apply to numerical types only
<code><</code>	less than	apply to numerical types only
<code>>=</code>	greater than or equal	apply to numerical types only
<code><=</code>	less than or equal	apply to numerical types only

Logical Operators

- Logical operators act upon **boolean** operands only.
- The outcome is always a value of type **boolean**.
- In particular, 1 and 2 and 1 or 2 logical operators occur in two forms:
 - 1) full **op1 & op2** and **op1 | op2** where both **op1** and **op2** are evaluated
 - 2) short-circuit - **op1 && op2** and **op1 || op2** where **op2** is only evaluated if the value of **op1** is insufficient to determine the final outcome

Table: Logical Operators

<code>&</code>	<code>op1 & op2</code>	logical AND
<code> </code>	<code>op1 op2</code>	logical OR
<code>&&</code>	<code>op1 && op2</code>	short-circuit AND
<code> </code>	<code>op1 op2</code>	short-circuit OR
<code>!</code>	<code>! op</code>	logical NOT
<code>^</code>	<code>op1 ^ op2</code>	logical XOR

Example: Logical Operators

```
class LogicalDemo {  
    public static void main(String[] args) {  
        int n = 2;  
        if (n != 0 && n / 0 > 10)  
            System.out.println("This is true");  
        else  
            System.out.println("This is false");  
    }  
} Another example is needed
```

Control Flow

- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:
 1. **selection** statements allow the program to choose different parts of the execution based on the outcome of an expression
 2. **Iteration** statements enable program execution to repeat one or more statements.
 3. **Jump** statements enable your program to execute in a non-linear fashion

Selection Statements

- Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.
- Java provides four selection statements:
 1. if
 2. if-else
 3. if-else-if
 4. switch

if Statement

- General form:
 if (expression) statement;
- If `expression` evaluates to `true`, execute `statement`, otherwise do nothing.
- The expression must be of type `boolean`.

Simple/Compound Statement

The component statement may be:

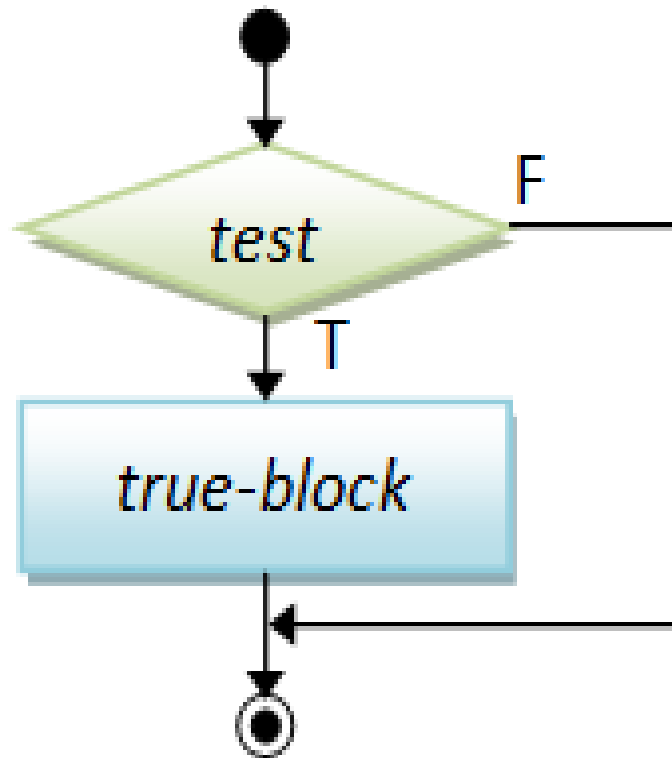
1) **simple**

if (expression) statement;

2) **compound**

```
if (expression) {  
    statement;  
}
```

Simple/Compound Statement



Simple if Statement

Example:

```
class IfStat{  
    public static void main(String args[]) {  
        boolean b;  
        b = true;  
        System.out.println("b is " + b);  
        if (b) System.out.println("executed");  
        b = false;  
        if (b) System.out.println("not executed");  
    }  
}
```

Compound if Statement

- Example:

```
class IfState1{  
    public static void main(String args[]) {  
        int a=10;  
        if ( a>0) {  
            System.out.println(" a is +ve num");  
        }  
    }  
}
```

- }

if-else Statement

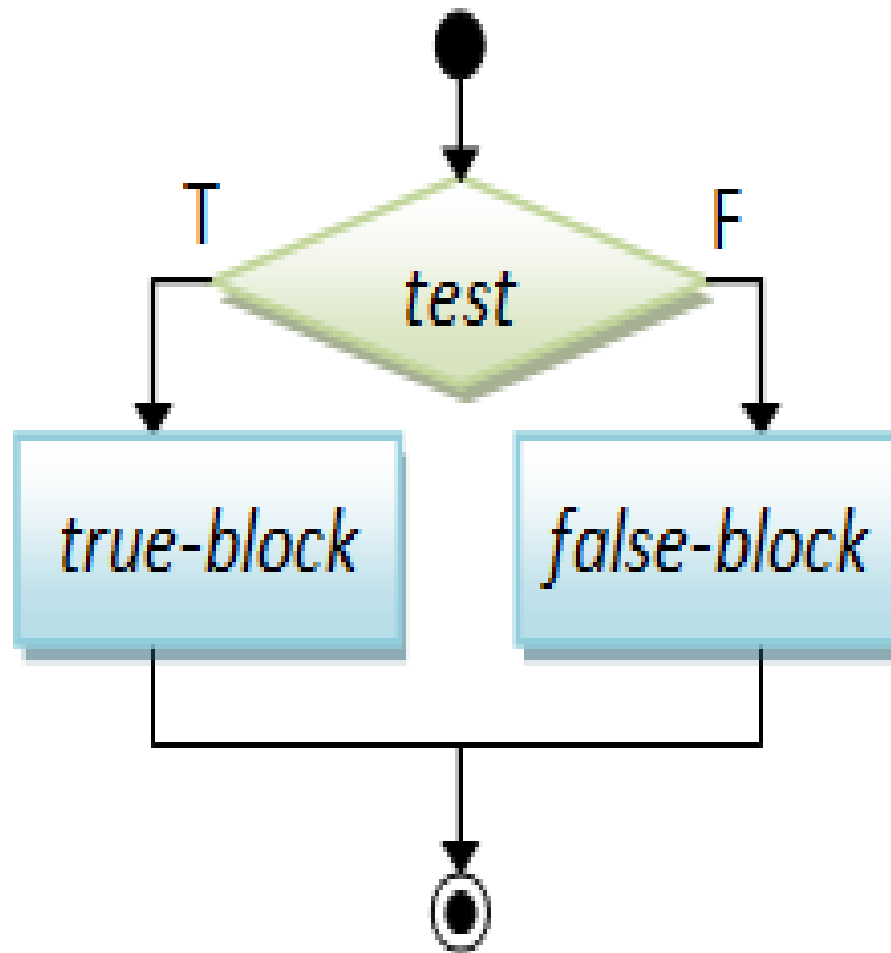
- If the conditional expression is true, the target of the if will be executed;
- otherwise, if it exists, the target of the else will be executed.
- At no time will both of them be executed.
- The conditional expression controlling the if must produce a boolean result.

if-else Statement

- **Syntax:**

```
if(conditional_expression){  
    <statements>;  
    ...;  
    ...;  
}  
else{  
    <statements>;  
    ....;  
    ....;  
}
```

if-else Statement



if else Statement

- Example:

```
class IfElse{  
    public static void main(String args[]) {  
        int a=-9;  
        if ( a>0) {  
            System.out.println(" a is +ve num");  
        }  
        else{  
            System.out.println(" a is - ve num");  
        }  
    }  
}
```

Nested if Statement

- A **nested if** is an if statement that is the target of another if or else.
- **Nested ifs** are very common in programming.
- When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

Nested if Statement

```
if ( test condition 1)
{ a
    //If the test condition 1 is TRUE
    then, it will check for test condition 2
    if ( test condition 2)
    {b
        //If the test condition 2 is TRUE,
        these statements will be executed
        Test condition 2 True statements;
    }b
    else
    {c
        //If the test condition 2 is FALSE,
        these lines will be executed
        Test condition 2 False statements;
    }c
```

```
}a
else
    {d
        //If the test condition 1 is FALSE
        then these lines will be executed
        Test condition 1 False statements;
    }d
```

//If the test condition 1 is FALSE then these lines will be executed

Test condition 1 False statements;

If Condition1 is FALSE, then STATEMENT3 will execute.

If Test Condition1 is TRUE, it will check for the Test Condition2 expression is TRUE, then STATEMENT1 will execute

Else STATEMENT2 executed.

Nested if Statement

- ```
class Example4_2{
 public static void main(String Args[]){
 int a = 3;
 if (a <= 10 && a > 0){
 System.out.println("Number is valid.");
 if (a < =5)
 System.out.println("From 1 to 5");
 else
 System.out.println("From 6 to 10");
 }
 else
 System.out.println("Number is not valid");
 }
}
```

# else-if ladder Statement

- A common programming construct that is based upon the nested if is the if-else-if ladder.

- It looks like this:

```
if(condition)
```

```
 statement;
```

```
else if(condition)
```

```
 statement;
```

```
else if(condition)
```

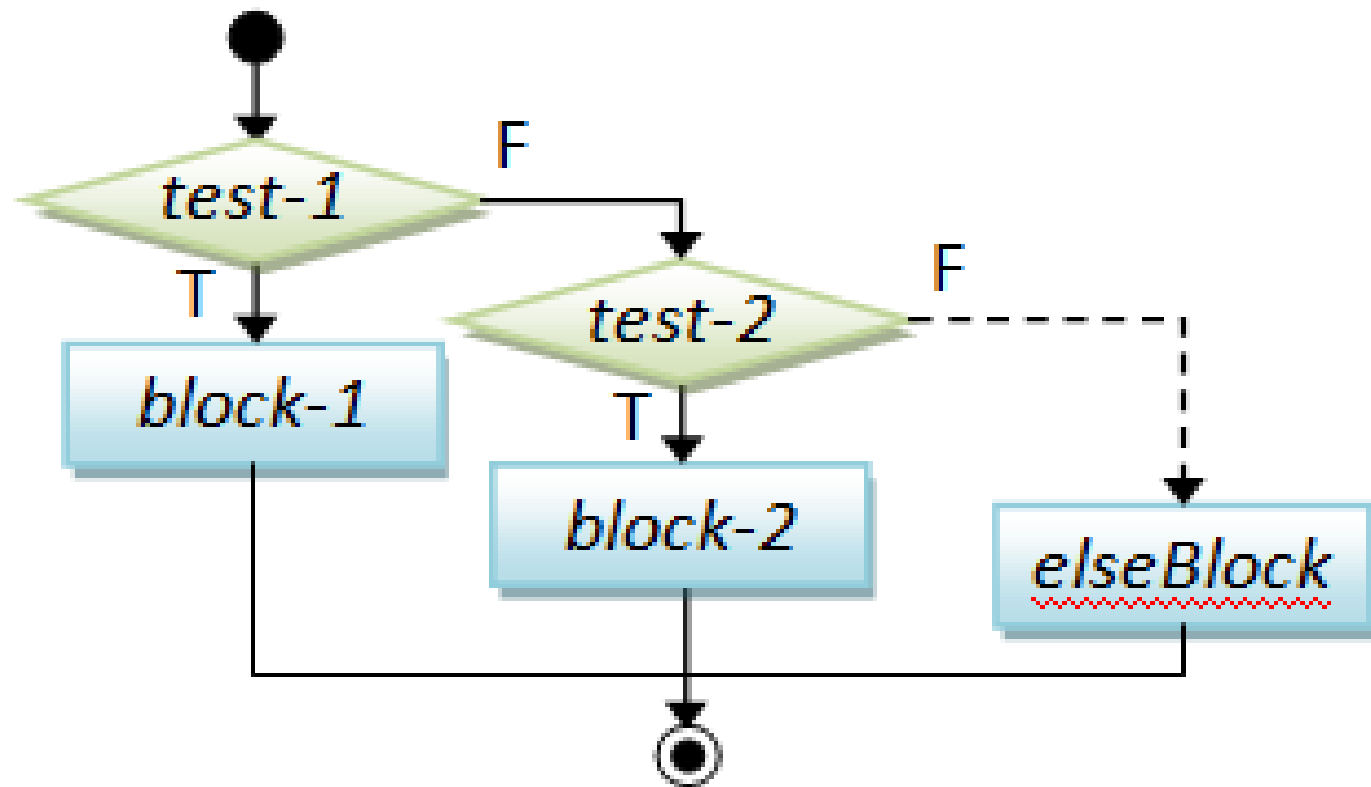
```
 statement;
```

- ...

```
else
```

```
 statement;
```

# else-if ladder Statement





# else-if ladder Statement

- **Semantics:**
  1. statements are executed **top-down**
  2. As soon as a **true condition** is found, the statement associated with it is executed, and the rest of the ladder is **bypassed**.
  3. If none of the conditions is **true**, the final **else** statement will be executed.
- The final **else** often acts as a **default condition**; that is, if all other conditional tests fail, the last else statement is performed.

# else-if ladder Statement

```
class Ladder {
 public static void main(String args[]) {
 int x;
 x = Integer.parseInt(args[0]);
 if(x==1) System.out.println("x is one");
 else if(x==2) System.out.println("x is two");
 else if(x==3) System.out.println("x is three");
 else if(x==4) System.out.println("x is four");
 else
 System.out.println("x is not between 1 and 4");
 }
}
```

- }

# switch Statement

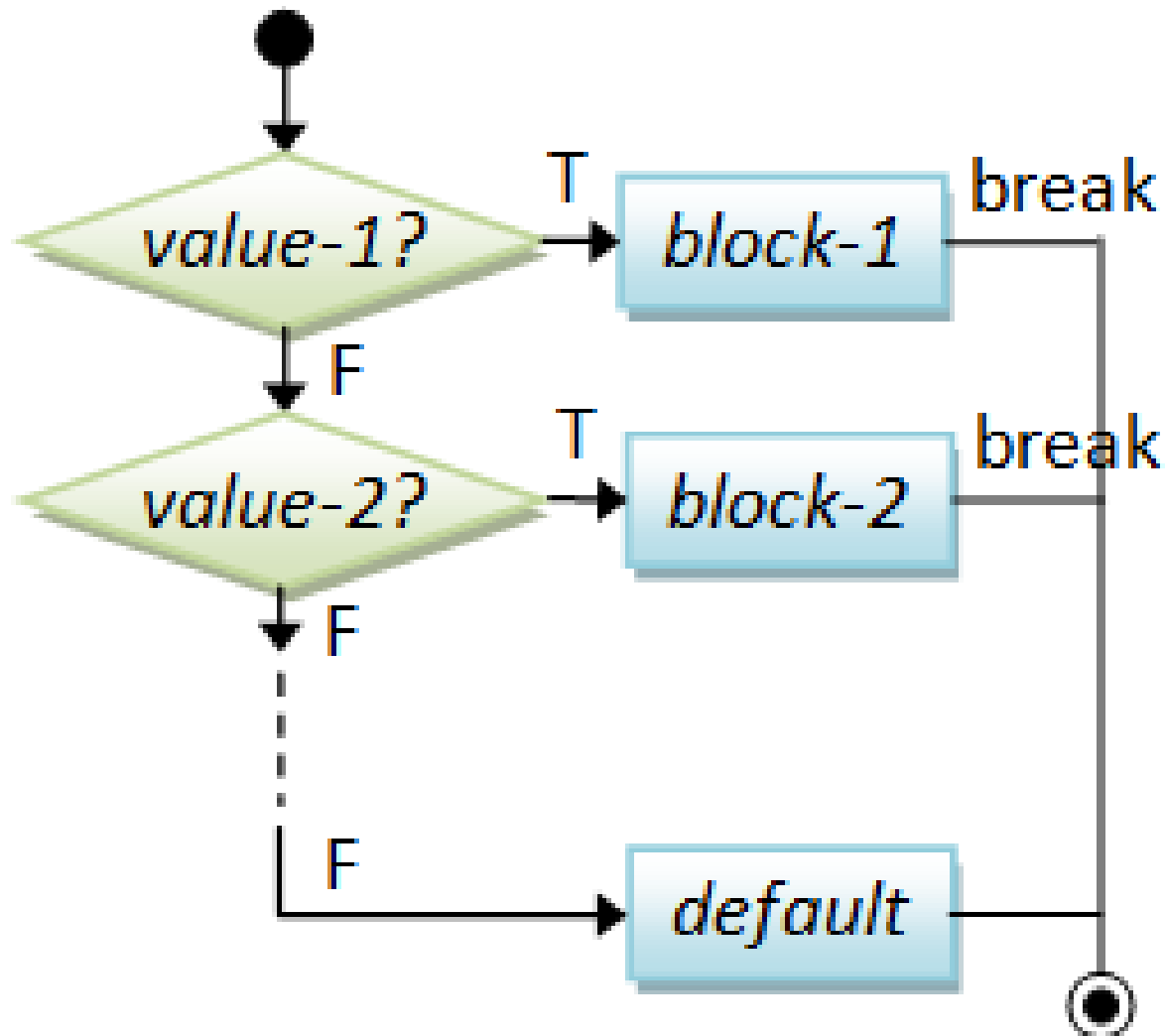
- `switch` provides a better alternative than `if-else-if` when the execution follows several branches depending on the value of an expression.
- The `switch` provides for a `multiway branch`. Thus, it enables a program to select among several alternatives.
- the value of an expression is successively tested against a list of constants.
- When a match is found, the statement sequence associated with that match is executed.

# switch Statement

- The general form of the switch statement is

- `switch` (expression) {
  - `case` constant1:
    - statement sequence
  - `break`;
  - `case` constant2:
    - statement sequence
  - `break`;
  - `case` constant3:
    - statement sequence
  - `break`;
  - ...
  - `default`:
    - statement sequence
- }

# switch Statement



# switch Statement

- Assumptions:

1. **expression** must be of type **byte**, **short**, **int** or **char**
2. each of the case values must be a literal of the compatible type
3. **case** values must be **unique**

Semantics:

1. **expression** is evaluated
  2. its value is compared with each of the **case** values
  3. if a **match** is found, the statement following the **case** is executed
  4. if no **match** is found, the statement following **default** is executed
- **break** makes sure that only the **matching** statement is executed.
  - Both **default** and **break** are **optional**.

# switch Statement

- `class SwitchDemo {`  
    `public static void main(String args[]) {`  
        `int i;`  
        `i = Integer.parseInt(args[0]);`  
        `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;`  
            `case 4: System.out.println("i is four"); break;`  
            `default: System.out.println("i is five or more");`  
            `}`  
        `}`  
    `}`