An Overview of Java

Prof. P. G. Patil
Asst. Prof. Dept. of CSE
HIT, Nidasoshi

Problem Solving Approaches

- Procedure Oriented Programming
- Object Oriented Programming

Procedure Oriented Programming(POP)

- It executes a series of procedures sequentially.
- The collection of data structure is related with each other as well as with the procedures.
- This is basically a top down problem solving approach.
- This approach characterizes a program as a series of linear steps (that is, code).
- The process-oriented model can be thought of as ***code acting on data.***
- Eg: C language

Limitations of POP

- Global Data is accessible by all the functions.
- Sometimes many functions access the same set of data.
- Program become complex to write and maintain.

Object Oriented Programming System (OOPS)

- It is a collection of objects.
- In OOPS we try to model real-world objects.
- Most real world objects have internal parts (Data Members) and interfaces (Member Functions) that enables us to operate them.
- This is basically the bottom up problem solving approach.
- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
- An object-oriented program can be characterized as ***data controlling access to code.***
- Eg: C++, JAVA, C#

Difference between POP and OOP

POP	OOP
Emphasis is on procedures (functions)	Emphasis is on data
Programming task is divided into a collection of data structures and functions.	Programming task is divided into objects (consisting of data variables and associated member functions)
Procedures are being separated from data being manipulated	Procedures are not separated from data, instead, procedures and data are combined together.
A piece of code uses the data to perform the specific task	The data uses the piece of code to perform the specific task
Data is moved freely from one function to another function using parameters.	Data is hidden and can be accessed only by member functions not by external function.
Data is not secure	Data is secure
Top-Down approach is used in the program design	Bottom-Up approach is used in program design
Debugging is the difficult as the code size increases	Debugging is easier even if the code size is more

Basic concepts (features) of Object-Oriented Programming

1.Objects
2. Classes
3. Data abstraction
4. Data encapsulation
5. Inheritance
6. Polymorphism
7. Binding
8. Message passing

- **Object:**
  - Everything in the world is an object.
  - An object is a collection of variables that hold the data and functions that operate on the data.
  - The variables that hold data are called fields.
  - The functions that operate on the data are called methods.

Three OOP Principles

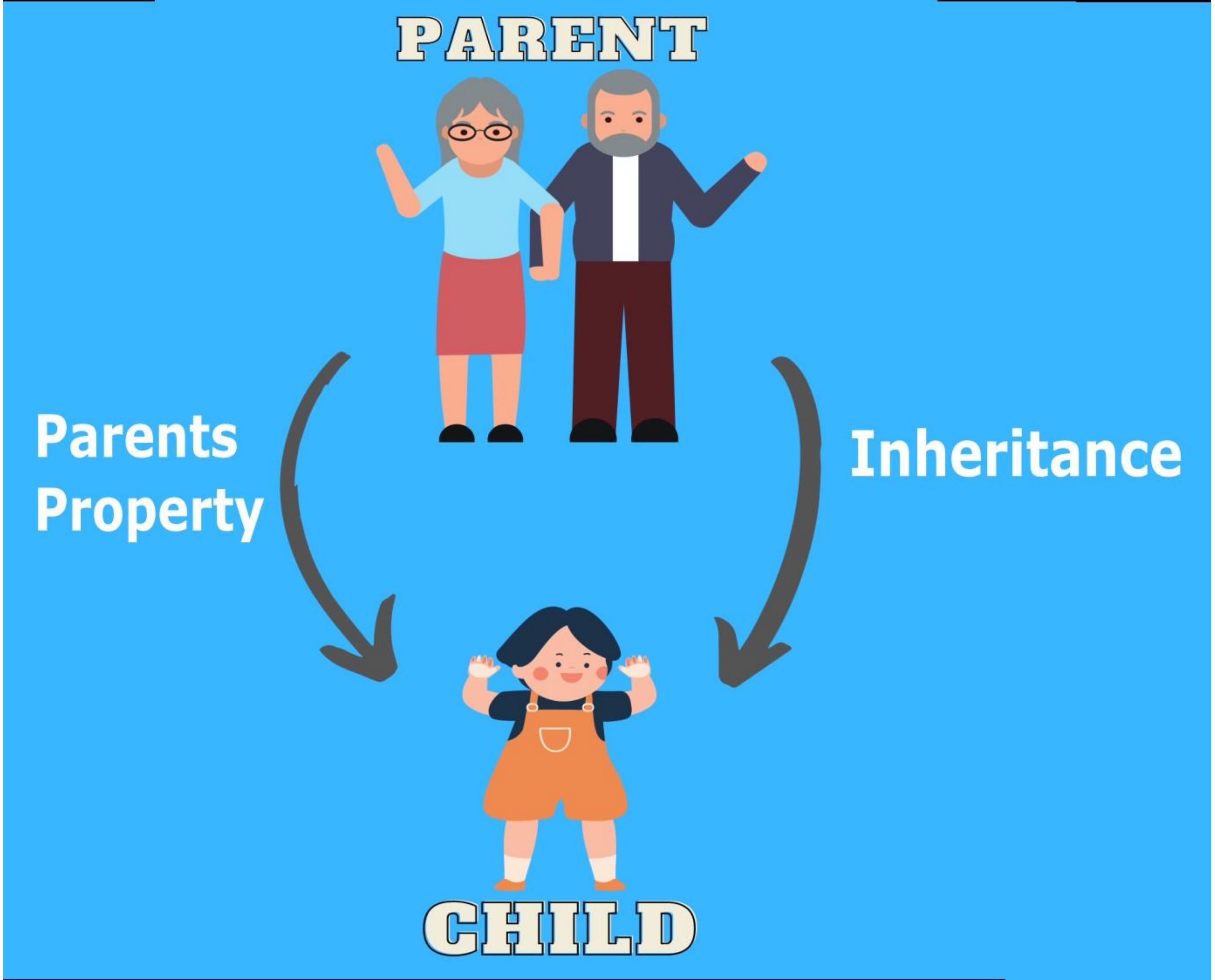
- Encapsulation
- Inheritance
- Polymorphism

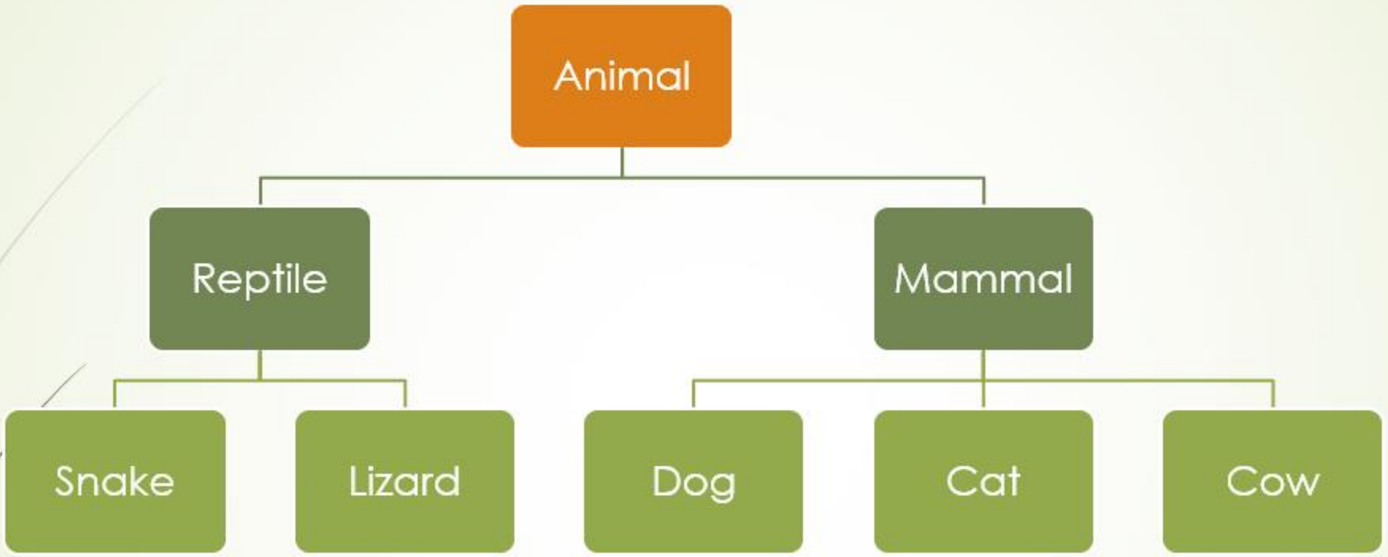
Encapsulation

- The wrapping of data & methods into a single unit(called class) is known as encapsulation.
- In Java, the basis of encapsulation is the class.
- A **class defines the structure and** behavior (data and code) that will be shared by a set of objects.
- Each object of a given class contains the structure and behavior defined by the class.

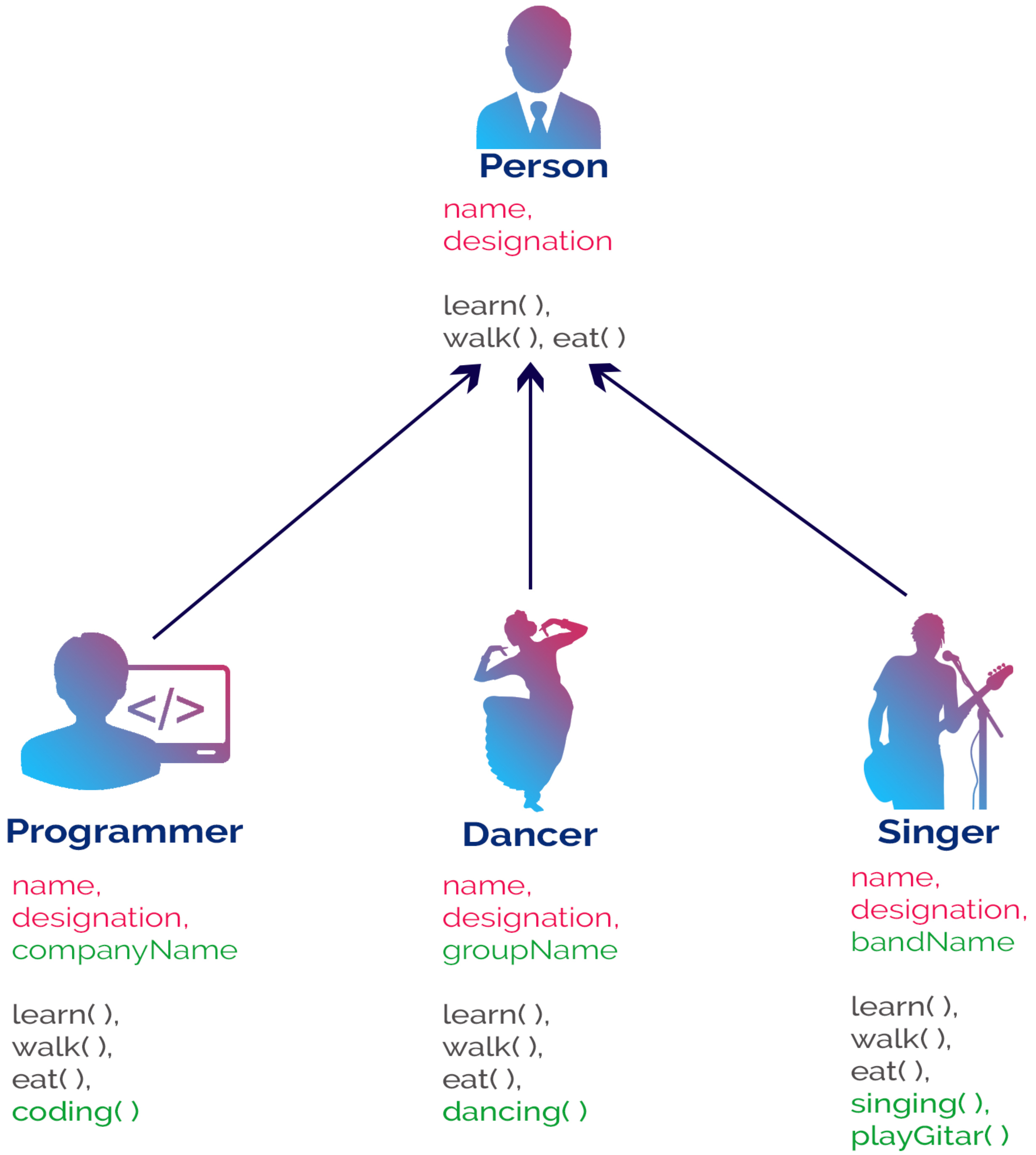
Inheritance

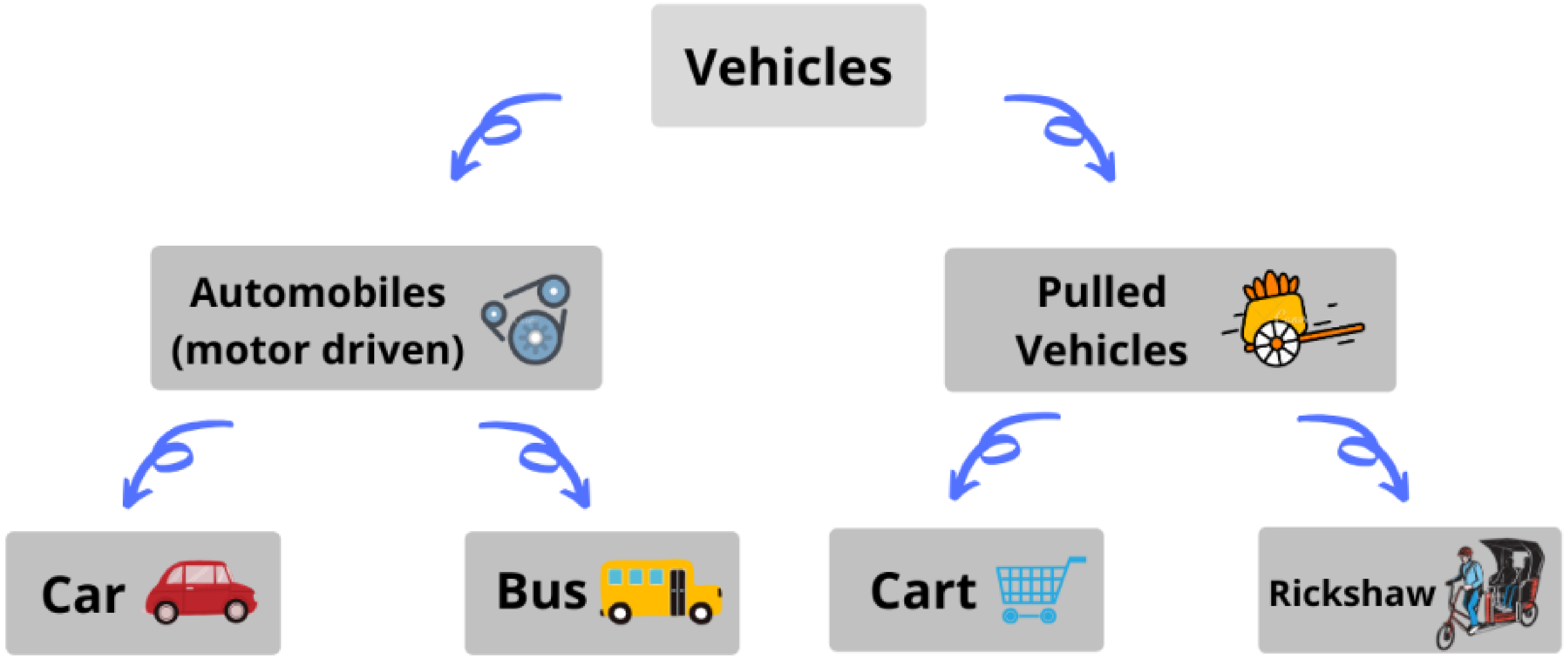
- Inheritance is the process by which one object acquires the properties of another object.
- This is important because it supports the concept of hierarchical classification.

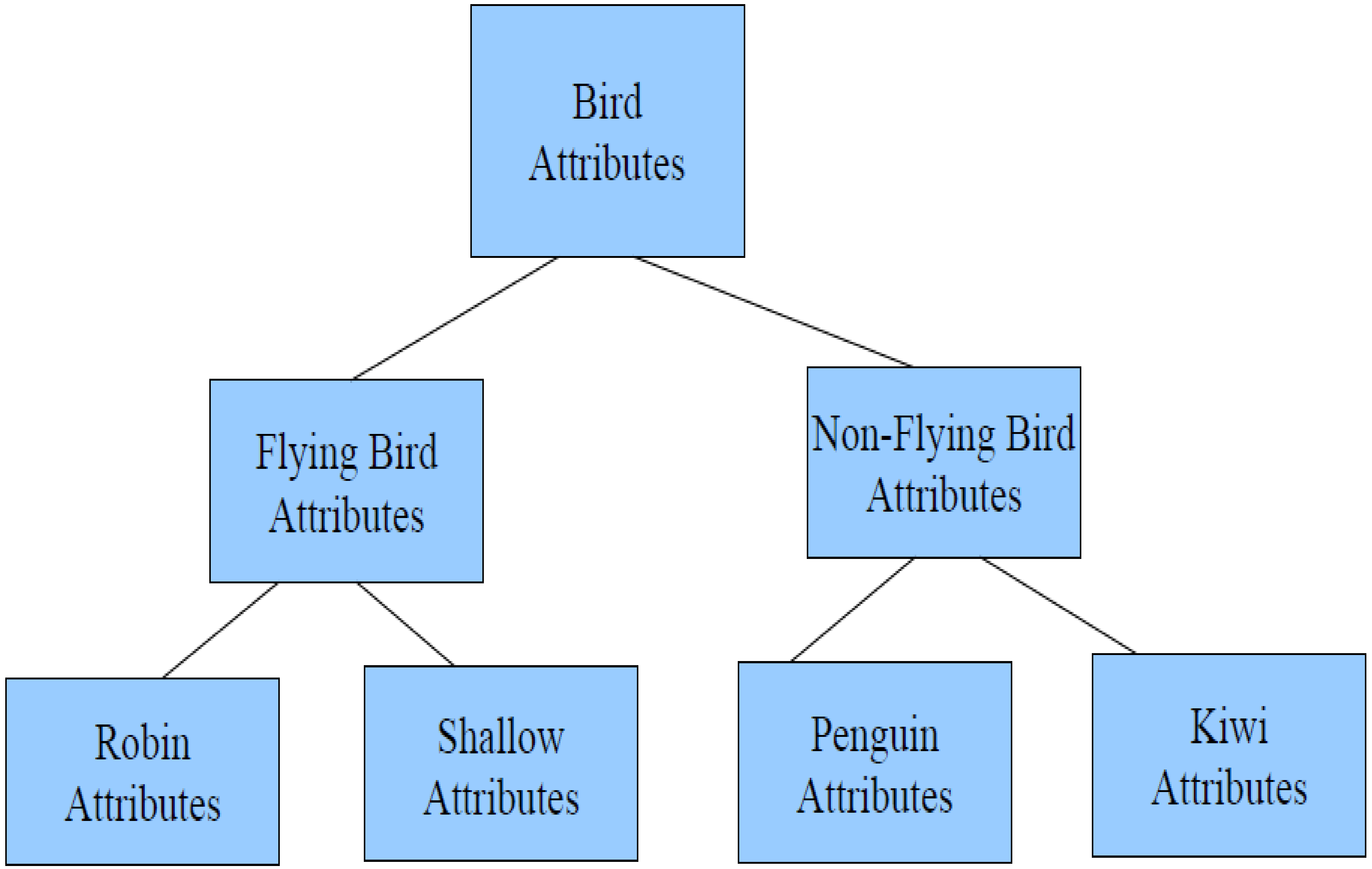

PARENT
Parents Property
Inheritance
CHILD



Inheritance

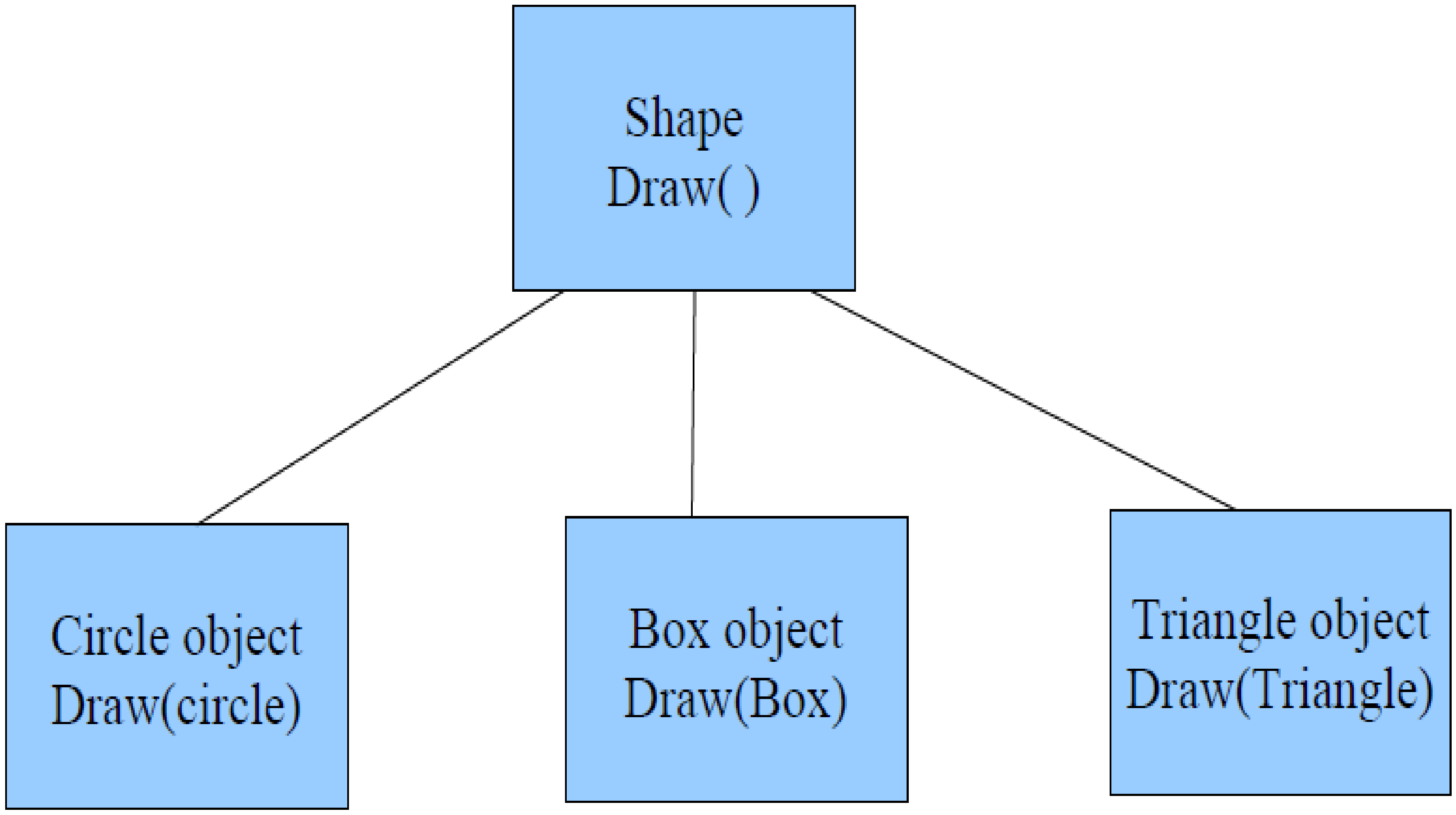

Person
name,
designation
learn( ),
walk( ), eat( )
Programmer
name,
designation,
companyName
learn( ),
walk( ),
eat( ),
coding( )
</>
Dancer
name,
designation,
groupName
learn( ),
walk( ),
eat( ),
dancing( )
Singer
name,
designation,
bandName
learn( ),
walk( ),
eat( ),
singing( ),
playGitar( )


Vehicles
Automobiles (motor driven)
Pulled Vehicles
Car
Bus
Cart
Rickshaw


Bird Attributes
Flying Bird Attributes
Non-Flying Bird Attributes
Robin Attributes
Shallow Attributes
Penguin Attributes
Kiwi Attributes

Polymorphism

- Polymorphism (from Greek, 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.
- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface.
- Eg: Method overloading & overriding.


Shape
Draw( )
Circle object
Draw(circle)
Box object
Draw(Box)
Triangle object
Draw(Triangle)

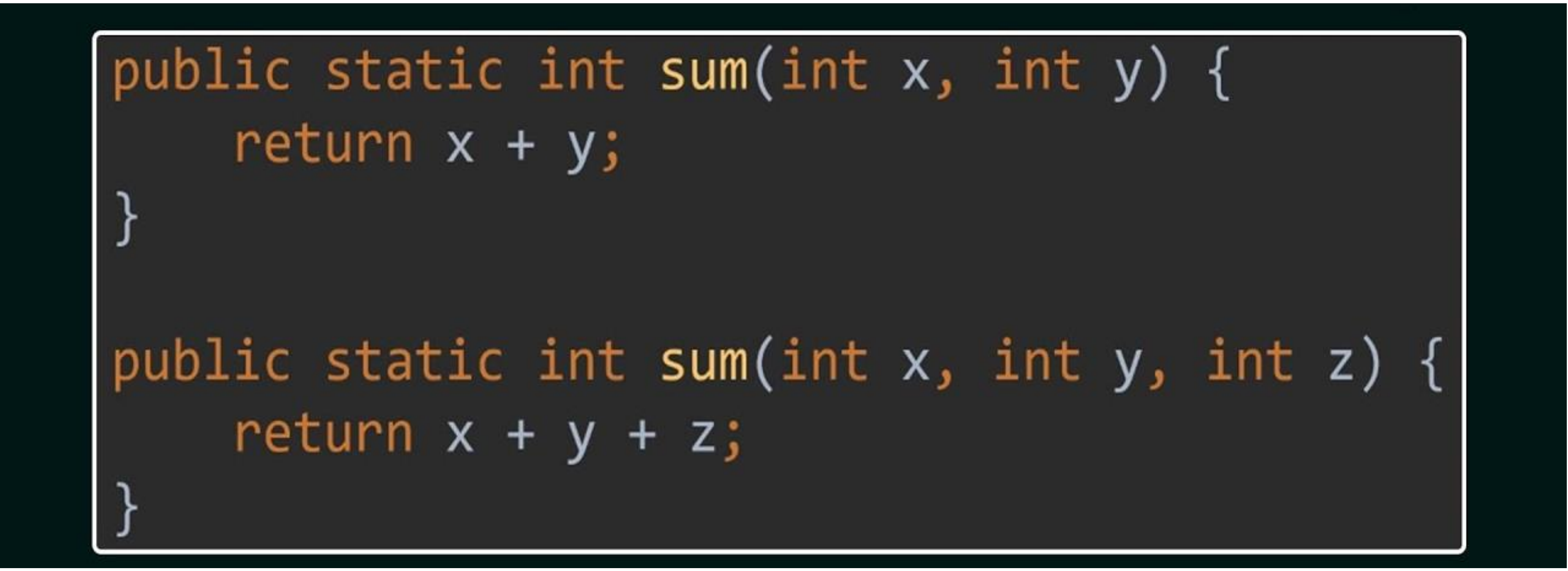
```
class Cat{

public void Sound(){
System.out.println("meow");
}

//overloading method
public void Sound(int num){
    for(int i=0; i<2;i++){
      System.out.println("meow");
     }
   }
}
```

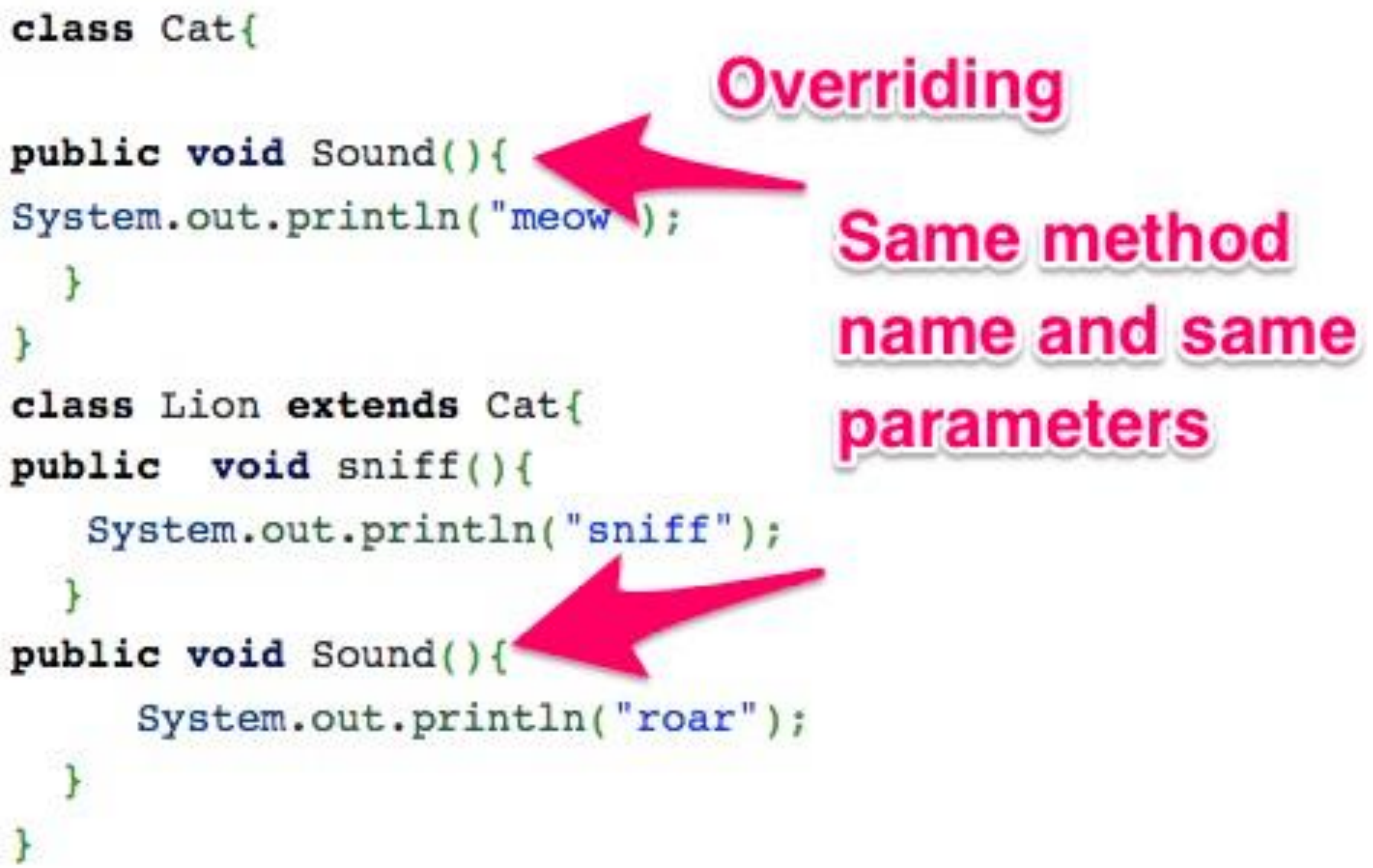
OVERLOADING

Same method name but different parameters



```java
public static int sum(int x, int y) {
    return x + y;
}

public static int sum(int x, int y, int z) {
    return x + y + z;
}
```

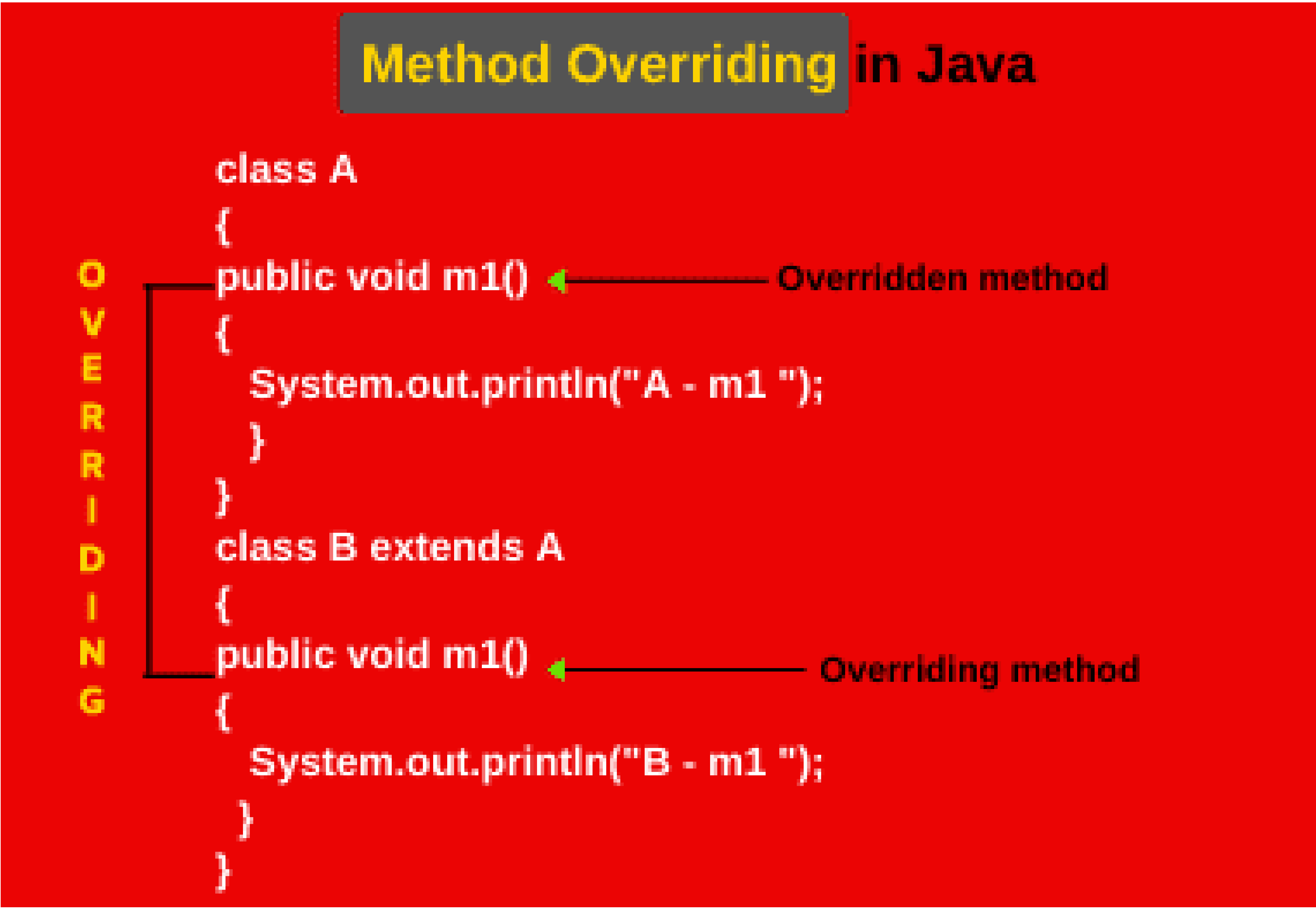


```
class Cat{

public void Sound(){
System.out.println("meow");
  }
}
class Lion extends Cat{
public  void sniff(){
   System.out.println("sniff");
  }
public void Sound(){
     System.out.println("roar");
  }
}
```

Overriding

Same method name and same parameters


Method Overriding in Java

```
class A
{
public void m1()
{
   System.out.println("A - m1 ");
   }
}
class B extends A
{
public void m1()
{
   System.out.println("B - m1 ");
  }
}
```

Introduction to JAVA

- Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations.
- Java is modeled after C++ .
- It was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan in 1991.
- They took 18 months to develop the first working version & was initially named as “Oak” & was renamed as “JAVA” in 1995.

- Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level (more about this later).
- Java also provides for portable programming with applets.
- Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive.
- Applets can be used to create animations, figures, or areas that can respond to input from the reader, games, or other interactive effects on the same Web pages among the text and graphics.

Java Features:

(1) Compiled and Interpreted

(2) Architecture Neutral/Platform independent and portable

(3) Object oriented

(4) Robust and secure.

(5) Distributed.

(6) Familiar, simple and small.

(7) Multithreaded and interactive.

(8) High performance

(9) Dynamic and extensible.

Java Environment:

- Java environment includes a large number of development tools and hundreds of classes and methods.
- The Java development tools are part of the systems known as Java development kit (JDK) and the classes and methods are part of the Java standard library known as Java standard Library (JSL) also known as application program interface (API).

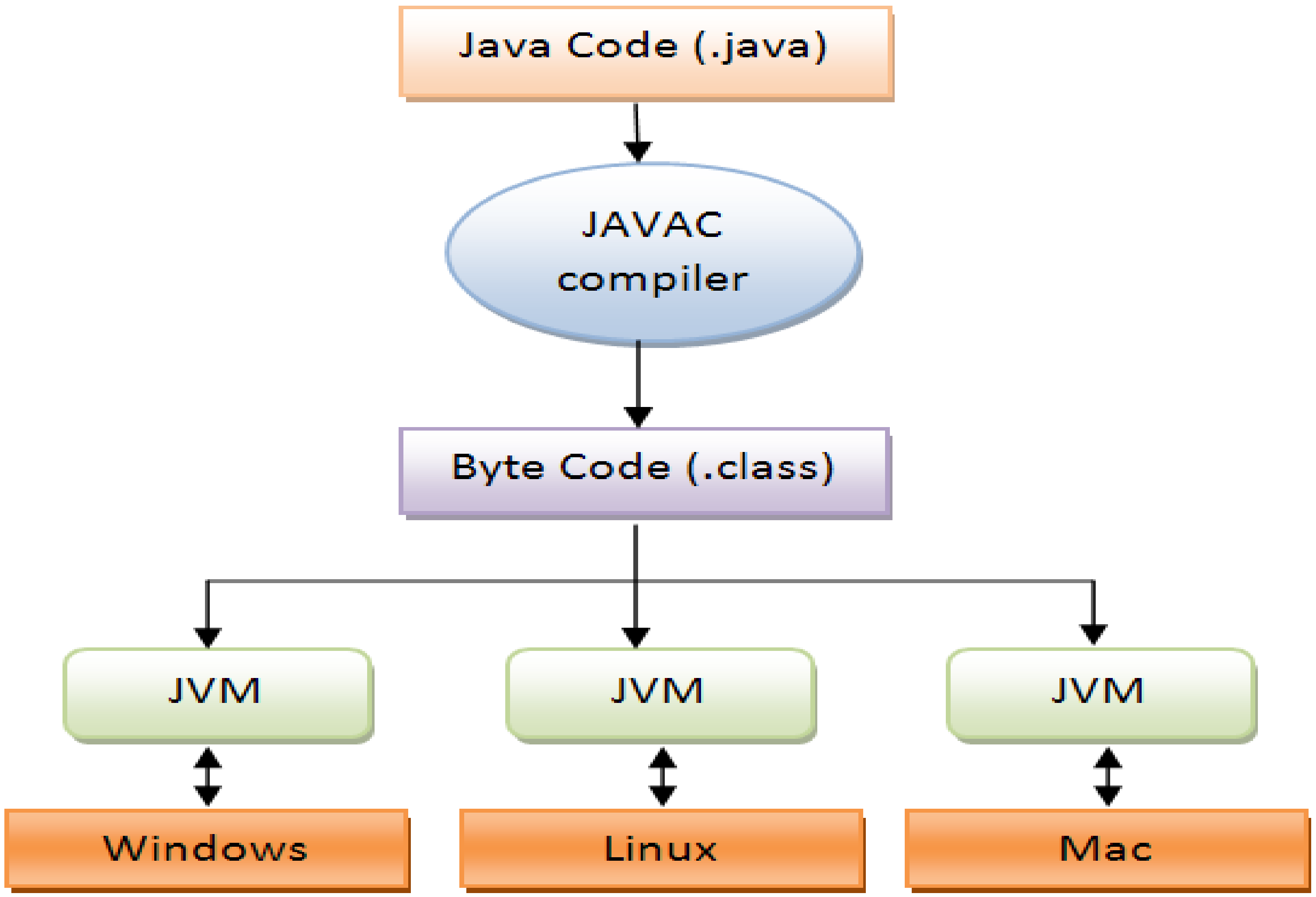
Java Development kits

- Java development kit comes with a number of Java development tools. They are:
  - (1) Appletviewer: Enables to run Java applet.
  - (2) javac: Java compiler.
  - (3) java : Java interpreter.
  - (4) javah : Produces header files for use with native methods.
  - (5) javap : Java disassembler.
  - (6) javadoc : Creates HTML documents for Java source code file.
  - (7) jdb : Java debugger which helps us to find the error.

Java API:

- Java standard library includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:
  - (a) Language support Package.
  - (b) Utilities packages.
  - (c) Input/output packages
  - (d) Networking packages
  - (e) AWT packages.
  - (f) Applet packages.

JVM(Java Virtual Machine)



Java Building and running Process:

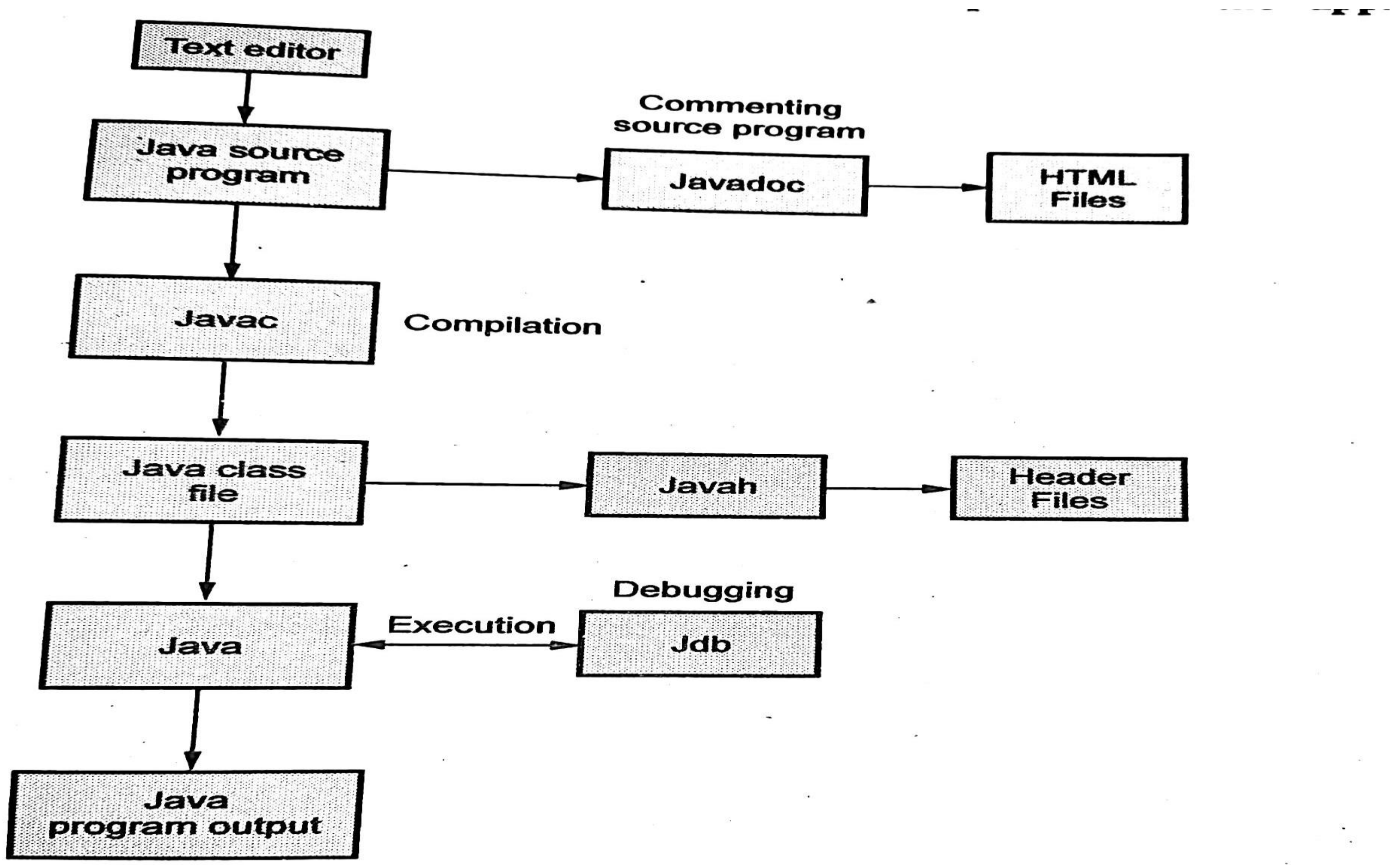


Fig. 3.2.1 Execution process of application program

Java Program structure:

It consists of 6 stages. They are:

(1) Documentation section: The documentation section contains a set of comment lines describing about the program.

(2) Package statement: The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the class defined here belong to the package.

package student;

(3) Import statements: Import statements instruct the compiler to load the specific class belongs to the mentioned package.

import student.test;

(4) Interface statements: An interface is like a class but includes a group of method declaration. This is an optional statement.

(5) Class definition: A Java program may contain multiple class definition The class are used to map the real world object.

(6) Main method class: The main method creates objects of various classes and establish communication between them. On reaching to the end of main the program terminates and the control goes back to operating system.

Simple Java Programs:

- Implementing a java program involves a series of steps. They include:
  - Creating the program
  - Compiling the program
  - Running the program

1. Creating the program

- We can create a program using any text editor. Consider the following program:

```
/* First Java program */
Class Test
{
    public static void main(String args[ ])
    {
      System.out.println(“Hello world”);
    }
}
```

- We must save this program in a file called Test.java
- This file is called as source file.

2. Compiling the program

- To compile the program, we must run the Java compiler javac with the name of the source file on the command line as shown below.

```
C:\jdk1.4\bin>javac Test.java
```

- If everything is ok, Java compiler creates a file called Test.class containing the bytecodes of the program.

3. Running the Program

- We need to use the Java interpreter to run a stand-alone application. At the command prompt, type

```
C:\jdk1.4\bin>java Test
```

- Now interpreter looks for the main method in the program & begins execution from there. When executed our program displays the following:
- Output: Hello world

Lexical Issues

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.
- **Whitespace:**
  - It is a free-form language. This means that you do not need to follow any special indentation rules.
  - In Java, whitespace is a space, tab, or newline.

- **Identifiers:**
  - Identifiers are used for class names, method names, and variable names.
  - An identifier may be any descriptive sequence of only uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
  - **Rules:**
    - An identifier must begin with an alphabet or underscore or dollar-sign character.
    - Second alphabet onwards can be combination of alphabets, digits, underscore or dollar-sign.
    - Java is case-sensitive, so VALUE is a different identifier than Value.
    - It must not be a keyword.
    - Some examples of valid identifiers are
      - eg: Avg, count, a4, basic_sal, $test etc.

- **Separators:**
  - In Java, there are a few characters that are used as separators.
  - The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements.
  - The separators are as below:

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.

Java Keywords

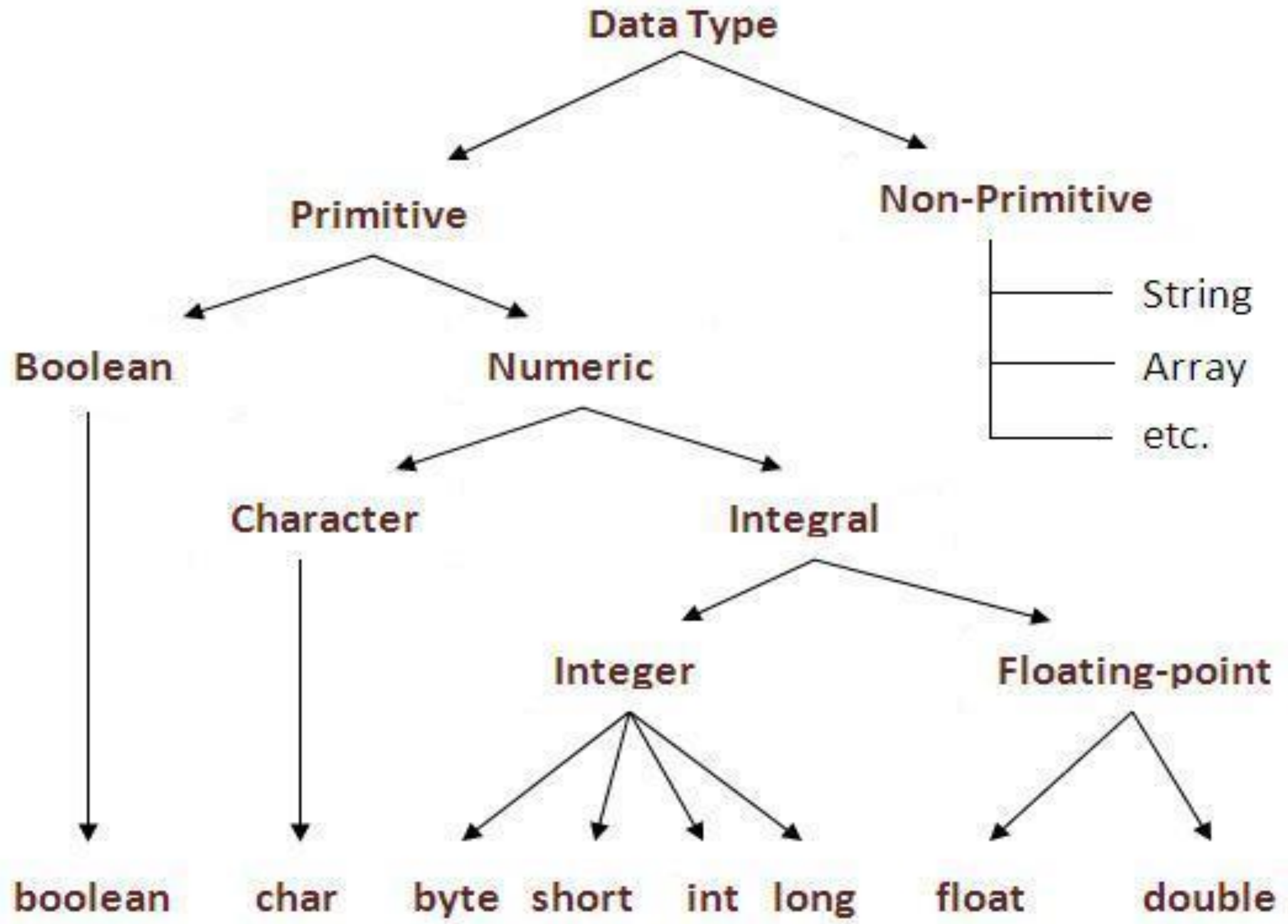
- There are 50 keywords currently defined in the Java language.
- These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.
- These keywords cannot be used as names for a variable, class, or method.
- The keywords **const** and **goto** are reserved but not used.
- In addition to the keywords, Java reserves the following: **true,** **false,** and **null.** These are values defined by Java.

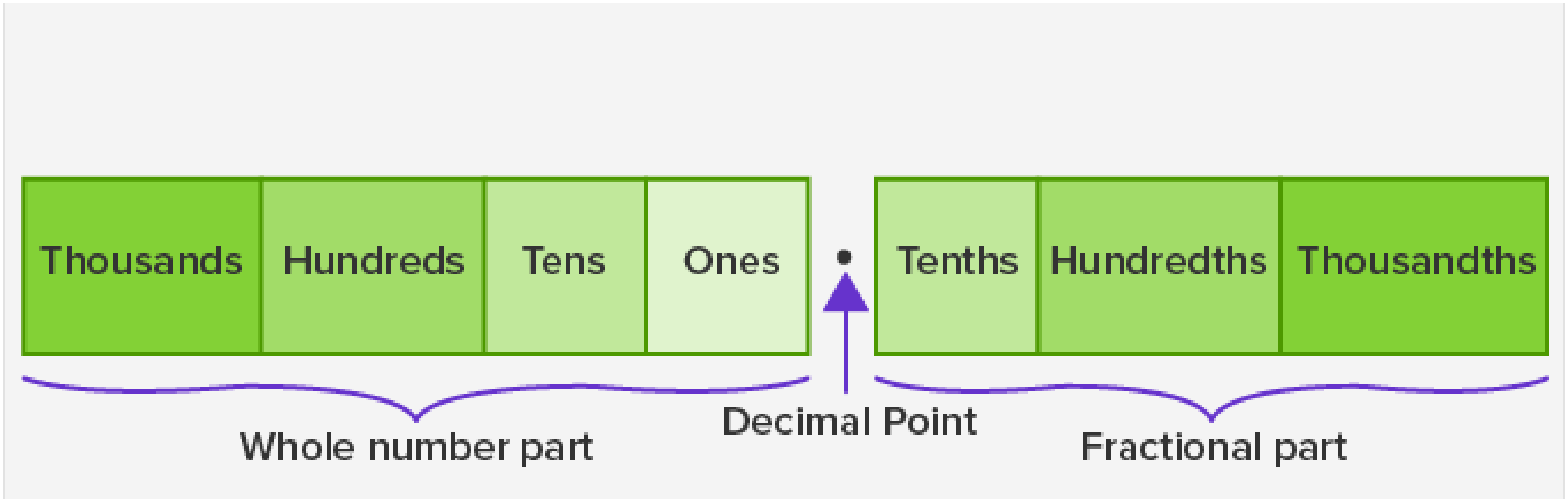
abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile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

TABLE 2-1 Java Keywords

Data Types in Java:

- In java, data types are classified into two categories :
  - 1. Primitive Data type
  - 2. Non-Primitive Data type


Data Type
Primitive
Non-Primitive
String
Array
etc.
Boolean
Numeric
Character
Integral
Integer
Floating-point
boolean
char
byte
short
int
long
float
double


Thousands
Hundreds
Tens
Ones
.
Tenths
Hundredths
Thousandths
Whole number part
Decimal Point
Fractional part

- The main difference between primitive and non-primitive data types are:
  - Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
  - Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
  - A primitive type has always a value, while non-primitive types can be null.
  - A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
  - The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Primitive Type Keyword

Type	Size in bytes	Range	Default Value
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	-2,147,483,648 to 2,147,483, 647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	approximately ±3.40282347E+38F (6-7 significant decimal digits) Java implements IEEE 754 standard	0.0f
double	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)	0.0d
char	2 bytes	0 to 65,536 (unsigned)	'\u0000'
boolean	Not precisely defined*	true or false	false

Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

Variables:

- A variable is an identifier that denotes a storage location used to store a data value.
- A variable may have different value in the different phase of the program.
- To declare one identifier as a variable there are certain rules.

Declaring & Initializing Variable

- One variable should be declared before using.
- The syntax is

  data-type variablename1;

  data-type variblaname1, ……,……,……,…, variablenameN;
- **Initializing a variable:** A variable can be initialize in two ways.
- They are
  - (a) Initializing by Assignment statements.
  - (b) Initializing by Read statements.

- Initializing by assignment statements:
  - One variable can be initialize using assignment statements. The syntax is :

    Variable-name = Value;
- Initialization of this type can be done while declaration
- Initializing by read statements:
  - Using read statements we can get the values in the variable.

Scope of Variable:

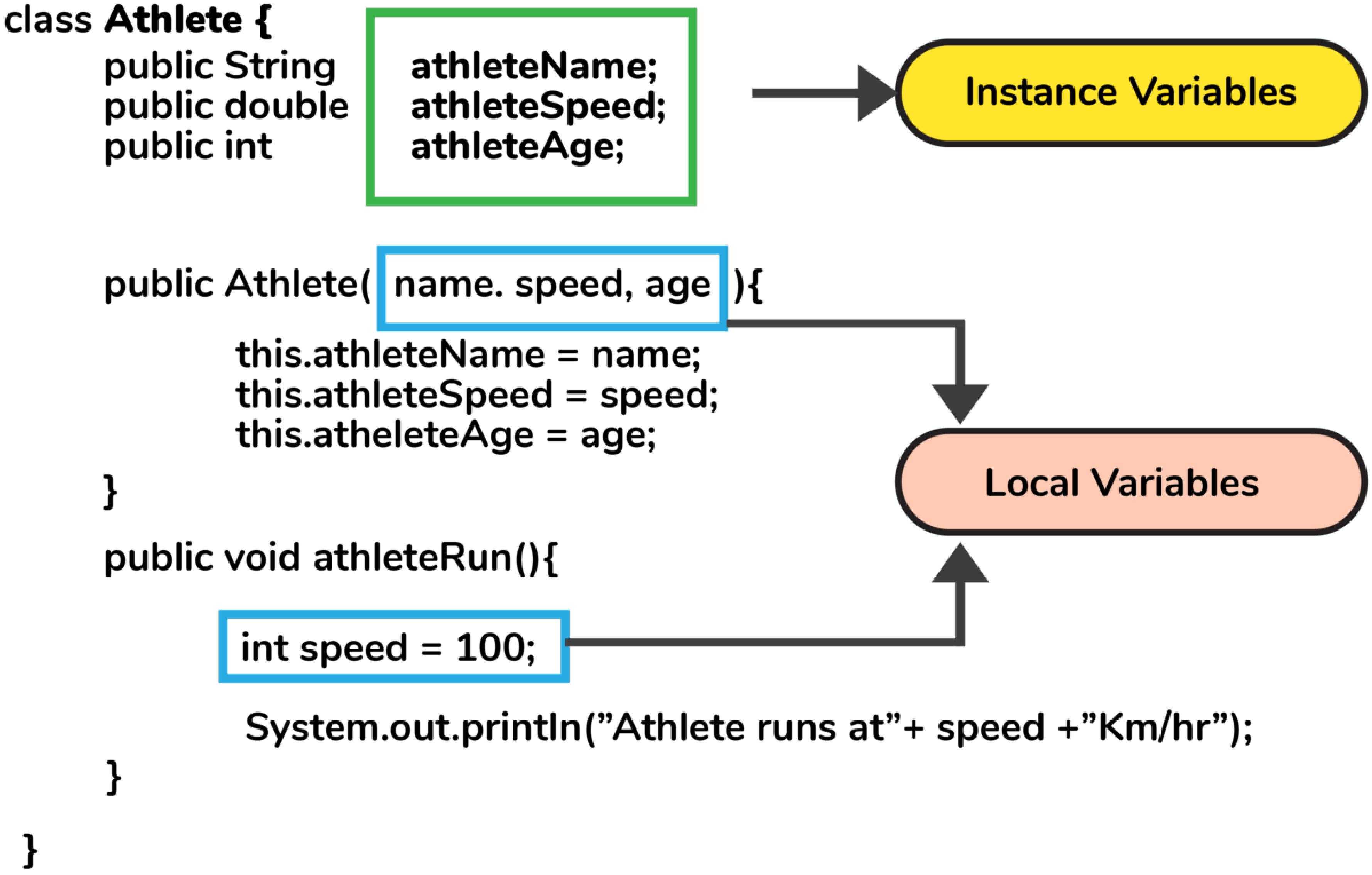
- Java variable is classified into three types. They are
  - (a) Instance Variable
  - (b) Local Variable
  - (c) Class Variable

INSTANCE VARIABLE VERSUS LOCAL VARIABLE

INSTANCE VARIABLE	LOCAL VARIABLE
A variable that is bounded to the object itself	A variable that is typically used in a method or a constructor
It is possible to use access modifiers for the instance variables	It is not possible to use access modifiers for the local variables
Can have default values	Do not have default values
Instance variables create when creating an object	Local variables create when entering the method or a constructor
Instance variables destroy when destroying the object	Local variables destroy when exiting the method or a constructor

Instance vs Local Variables

InterviewBit



```
class Athlete {
        public String    athleteName;
        public double    athleteSpeed;
        public int       athleteAge;


        public Athlete( name. speed, age ){

                this.athleteName = name;
                this.athleteSpeed = speed;
                this.atheleteAge = age;

        }

        public void athleteRun(){

                int speed = 100;

                System.out.println("Athlete runs at"+ speed +"Km/hr");
        }

 }
```

Summary of scope and lifetime of variables

Variable Type	Scope	Lifetime
Instance variable	Throughout the class except in static methods	Until the object is available in the memory
Class variable	Throughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Arrays in Java

- **Array** which stores a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- **Declaring Array Variables:**
  - To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference.
  - Here is the syntax for declaring an array variable:

```
dataType[]    arrayRefVar;
           or
dataType   arrayRefVar[];
```

- **Example:**
  - The following code snippets are examples of this syntax:

```
int[] myList;
        or
int myList[];
```

- **Creating Arrays:**
  - You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new datatype [arraySize] ;
```

- Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

- Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

long form

declaration

```
double[] a;
```

creation

```
a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = 0.0;
```

initialization

short form

```
double[] a = new double[N];
```

initializing declaration

```
int[] a = { 1, 1, 2, 3, 5, 8 };
```

Processing Arrays:

- When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Array Initialization

- Arrays can be initialized when they are declared.
- The process is much the same as that used to initialize the simple types.
- An array initializer is a list of comma-separated expressions surrounded by curly braces.
- The commas separate the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use new.

```
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.");
    }
}
```

Average of Array Elements

```
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);
}
}
```

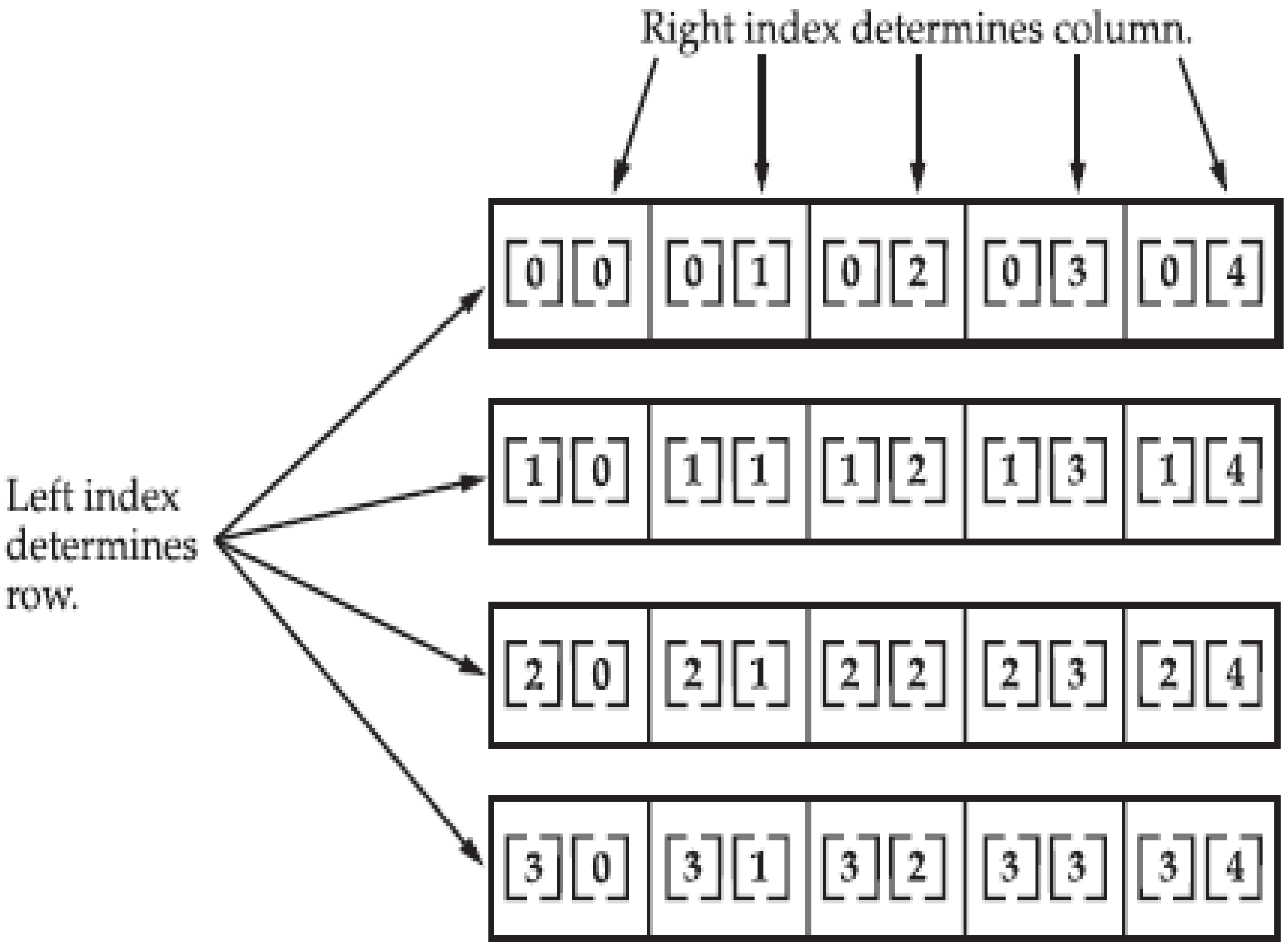
Multidimensional Arrays

- In Java, *multidimensional arrays are actually arrays of arrays.*
- *These, as you might expect, look* and act like regular multidimensional arrays.
- Here is the syntax for declaring an array variable:
  - dataType[][]   arrayRefVar;   or
  - dataType   arrayRefVar[][];

- For example, the following declares a two dimensional array variable called **twoD.**

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

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



Given: int twoD [ ] [ ] = new int [4] [5] ;

```
// Demonstrate a two-dimensional array.
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 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.
- For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal.

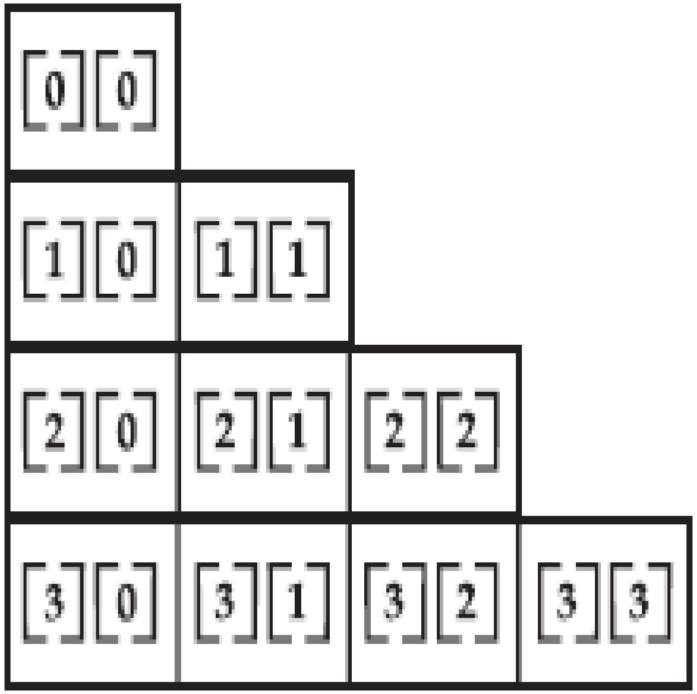
```
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
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.
- To do so, simply enclose each dimension’s initializer within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.

```
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();
}
}
}
```

- When you run this program, you will get the following 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

A Second Short Program

```java
class Example2 {
  public static void main(String args[]) {
    int num; // this declares a variable called num

    num = 100; // this assigns num the value 100

    System.out.println("This is num: " + num);

    num = num * 2;

    System.out.print("The value of num * 2 is ");
    System.out.println(num);
  }
}
```

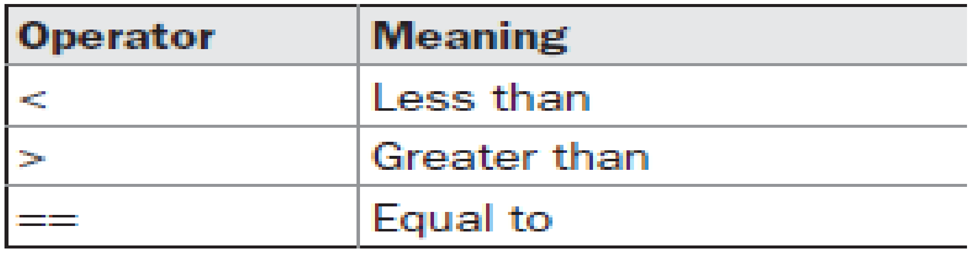
When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

Two Control Statements

- **The if Statement**
  - The Java if statement works much like the IF statement in any other language.
  - Further, it is syntactically identical to the if statements in C, C++, and C#. Its simplest form is shown here:
  - **if(condition) statement;**
  - Here, condition is a Boolean expression. If condition is true, then the statement is executed.
  - If condition is false, then the statement is bypassed. Here is an example:
  - if(num < 100)
  - System.out.println("num is less than 100");

- Java defines a full complement of relational operators which may be used in a conditional expression. Here are a few:



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

```
class IfSample {
  public static void main(String args[]) {
    int x, y;

    x = 10;
    y = 20;

    if(x < y) System.out.println("x is less than y");

    x = x * 2;
    if(x == y) System.out.println("x now equal to y");

    x = x * 2;
    if(x > y) System.out.println("x now greater than y");

    // this won't display anything
    if(x == y) System.out.println("you won't see this");
  }
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

The for Loop

- The simplest form of the for loop is shown here:
- **for(*initialization; condition; iteration) statement;***
- In its most common form, the initialization portion of the loop sets a loop control variable to an initial value.
- The condition is a Boolean expression that tests the loop control variable.
- If the outcome of that test is true, the for loop continues to iterate. If it is false, the loop terminates.
- The iteration expression determines how the loop control variable is changed each time the loop iterates.

```
class ForTest {
  public static void main(String args[]) {
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("This is x: " + x);
  }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

Using Blocks of Code

- Java allows two or more statements to be grouped into blocks of code, also called code blocks.
- This is done by enclosing the statements between opening and closing curly braces.
- Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.

Consider this if statement:

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

```
class BlockTest {
  public static void main(String args[]) {
    int x, y;

    y = 20;

    // the target of this loop is a block
    for(x = 0; x<10; x++) {
      System.out.println("This is x: " + x);
      System.out.println("This is y: " + y);
      y = y - 2;
    }
  }
}
```

The Java Class Libraries

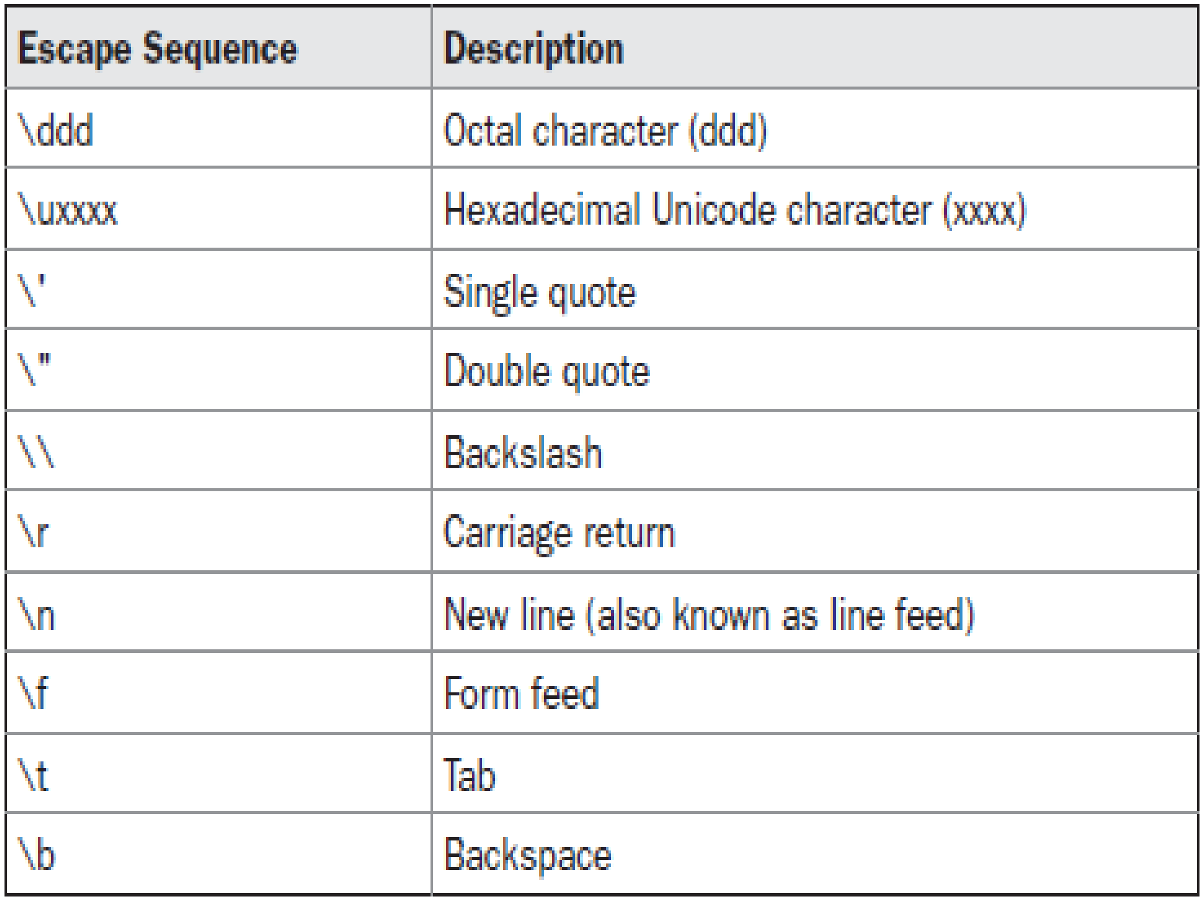
- In Java programs, we make use of two of Java's built-in methods: **println( ) and print( ).**
- As mentioned, these methods are members of the System class, which is a class predefined by Java that is automatically included in your programs.
- The Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics.
- The standard classes also provide support for windowed output.

Java Is a Strongly Typed Language

- Java is a strongly typed language.
- 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.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Character Escape Sequences

TABLE 3-1
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

The Scope and Lifetime of Variables

- So far, all of the variables used have been declared at the start of the main( ) method.
- However, 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.
- Thus, each time you start a new block, you are creating a new 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.
- The scope defined by a method begins with its opening curly brace.
- However, if that method has parameters, they too are included within the method's scope.
- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- Thus, when you declare a variable within a scope, you are localizing that variable and **protect**ing it from unauthorized access and/or modification.

- Scopes can be nested.
- For example, each time you create a block of code, you are creating a new, nested scope.
- When this occurs, the outer scope encloses the inner scope.
- This means that objects declared in the outer scope will be **visible** to code within the inner scope.
- However, the reverse is not true.
- Objects declared within the inner scope will not be visible outside it.

```
// Demonstrate block scope.
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);
  }
}
```

Lifetime of Variable

- Within a block, variables can be declared at any point, but are valid only after they are declared.
- Thus, if you define a variable at the start of a method, it is available to all of the code within that method.
- Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

- variables are created when their scope is entered, and destroyed when their scope is left.
- This means that a variable will not hold its value once it has gone out of scope.
- Therefore, variables declared within a method will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left.
- Thus, the lifetime of a variable is confined to its scope.
- If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

```
// Demonstrate lifetime of a variable.
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);
    }
  }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

- Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

```
// This program will not compile

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

- It is often necessary to store a value of one type into the variable of another type.
- In these situations the value that to be stored should be casted to destination type.
- Assigning a value of one type to a variable of another type is known as **Type Casting.**
- Type casting can be done in two ways.
  - Automatic Type Conversion (**Widening** Casting)
  - Explicit Type Conversion (**Narrowing** Casting)

Automatic Type Conversion

- 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 b = 10;

  int a = b;

$$\begin{tikzcd}
\text{byte} \arrow[r] & \text{short} \arrow[r] & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\
{} \arrow[rrrrr, "\text{widening}"'] & & & & & {}
\end{tikzcd}$$

Explicit Type Conversion

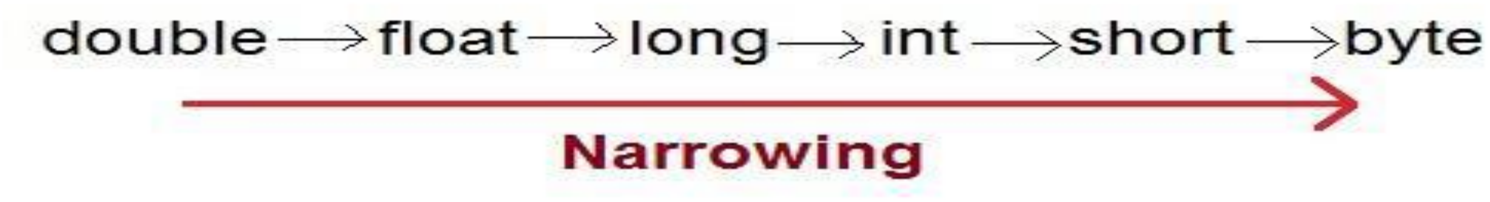
- The conversion will not be performed automatically, between a byte and an int.
- This kind of conversion is sometimes called a **narrowing** conversion, since you are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast.
- A cast is simply an explicit type conversion. It has this general form:

(target-type) value

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

- Ex :

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



```java
// Demonstrate casts.
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);
  }
}
```

This program generates the following 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

- In addition to assignments, there is another place where certain type conversions may occur: **in expressions**.
- In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.
- Ex:

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

- As useful as the automatic promotions are, they can cause confusing compile-time errors.
- For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

- In cases where you understand the consequences of overflow, you should use an explicit cast, such as

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

The Type Promotion Rules

- Java defines several type promotion rules that apply to expressions.
  - All byte, short, and char 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);
  }
}
```

A Few Words About Strings

- The **String** type is used to declare string variables.
- You can also declare arrays of strings.
- A quoted string constant can be assigned to a String variable.
- A variable of type String can be assigned to another variable of type String.
- You can use an object of type String as an argument to println( ).
- For example, consider the following fragment:

```
String str = "this is a test";
System.out.println(str);
```