

# Overview of Java Platform

## Brief history of Java

## HISTORY OF JAVA

- Was originally developed by James Gosling at Sun Microsystems, Inc. in 1991
- Intended to let application developers "write once, run anywhere" (WORA)
- Derives its syntax from C and C++
- Originally designed for small, embedded systems in electronic appliances like set-top boxes
- Initially called as **GreenTalk**, later named **Oak** and finally renamed to “**Java**” in 1995

## EVOLUTION OF JAVA

Version	Year	Important New Features
1.0	1996	Language
1.1	1997	JDBC, Inner Classes, RMI
1.2	1998	Just In Time (JIT) compiler, Collections framework, Java Foundation Classes
1.3	2000	Enhancements
1.4	2004	Assertions
5.0	2004	Generics, “for each” Loop, varargs, autoboxing, metadata, enumerations, static import, Annotations
6	2006	JDBC 4.0, Console, Navigable Sets and Navigable Maps
7	2011	Strings in switch, Multiple Exception Handling, Try with Resources, underscore in literals, Diamond Syntax
8	2014	Lambda Expressions, Functional Interfaces, Date/Time changes, Stream Collection Types

## PROGRAMMING PLATFORM

- Java is not just a Language but a Programming Platform
- Java Platform has following components
  - Language with nice Features and pleasant syntax
  - High Quality Execution Environment that provides services like
    - Portability across operating systems
    - Automatic Garbage Collection
- Vast Library known as Java API containing lots of reusable code

## JAVA BUZZWORDS

- Simple : Partially modelled on C++, but greatly simplified considering the flaws of earlier languages
- Object Oriented:
  - Helps visualize the program using real-life objects
  - Provides modularity and reusability
- Distributed : Strong Networking capabilities integrated in java
- Robust : Easy Memory Management and early error checking and Exception handling

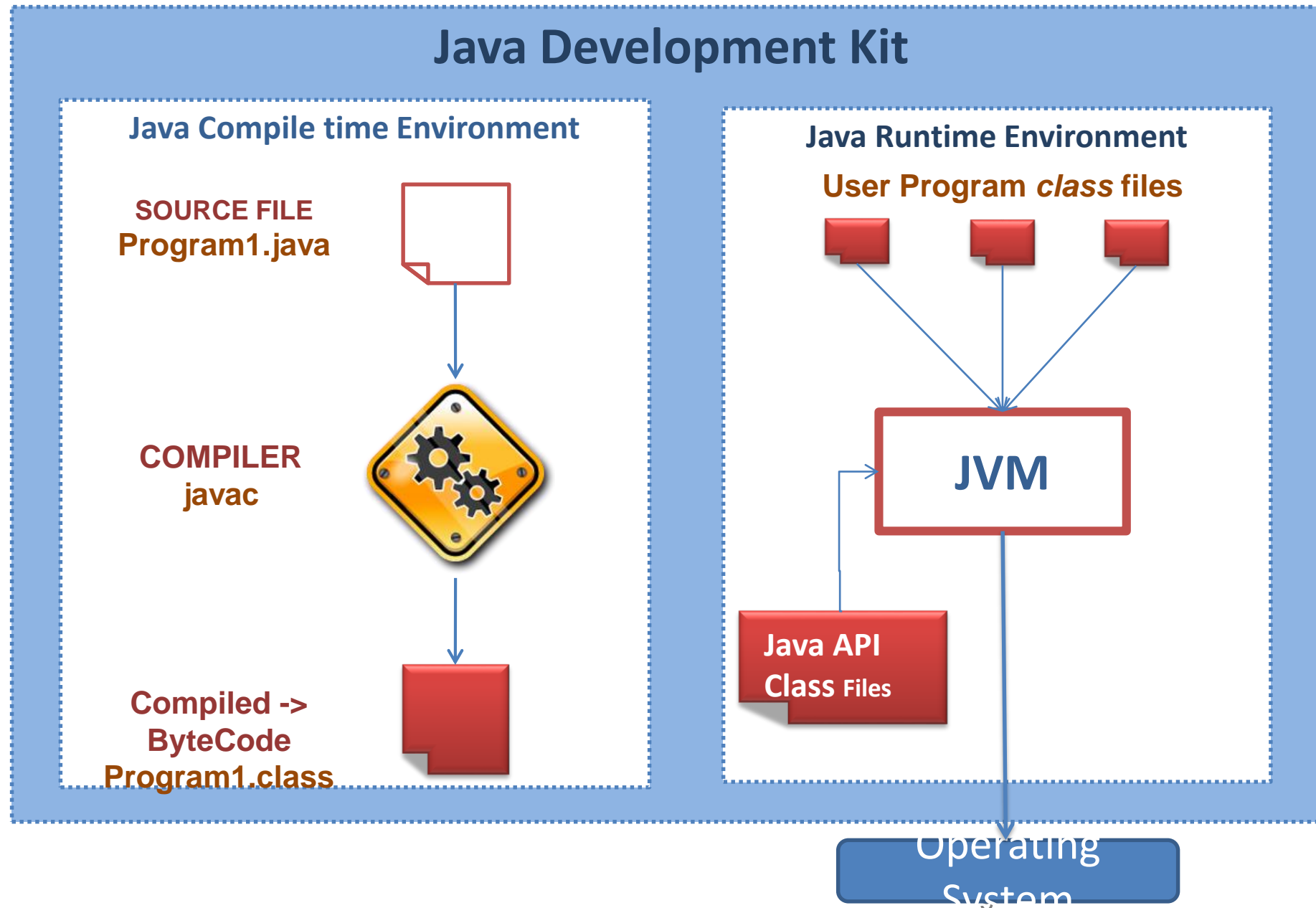
## JAVA BUZZWORDS

- Secure : Designed to prevent stack overrun, Memory corruption, unauthorized file access etc.
- Architecture Neutral/Portable: Java application developed in one platform can be used in other platforms with ease.
- High Performance
  - Uses Technology known as Just-in-time compilation
  - Java Hot Spot performance Engine includes compiler for optimizing frequently used code
- Multithreaded : Perform several tasks simultaneously within a program

- Java Development Environment



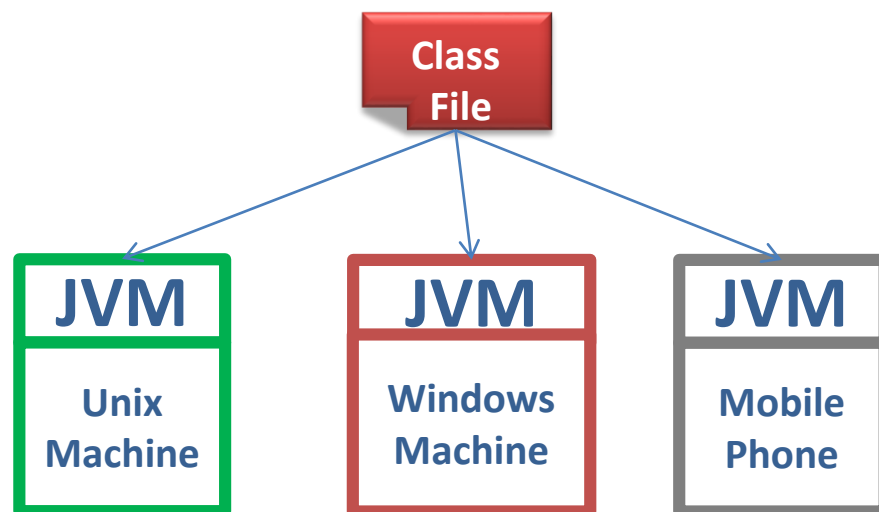
# JAVA PLATFORM



- Java compiler compiles the source code into a byte code
- The byte code will be in a file with extension `.class`
- JVM interprets the byte code into specific machine language of the underlying platform

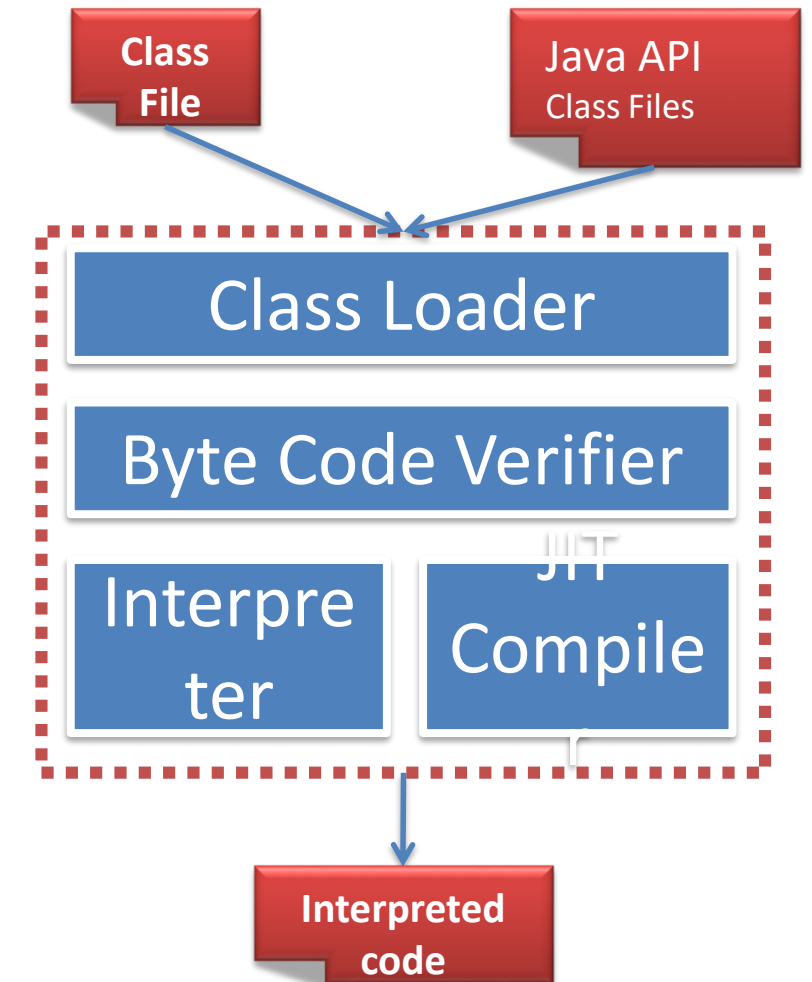
## JVM – JAVA VIRTUAL MACHINE

- JVM is Platform dependent
  - Different JVM's for different operating systems
- The Platform executes the JVM interpreted Machine language



## JVM INTERNALS

- Class Loader
  - Loads all the classes needed to execute a given class
- Byte Code Verifier
  - Verifies access violations, Illegal code, data conversions
- Interpreter
  - Executes bytecode and makes calls to the underlying hardware
- Just-In-Time (JIT) compiler
  - Part of JVM
  - JIT compiler is enabled by default, and is activated when a Java method is called
  - Compiles the bytecode of a method being called repeatedly into native platform code



## JRE - JAVA RUNTIME ENVIRONMENT

- Intended for end users executing Java Applications
- Contains
  - JVM
  - Java API's
  - Java Plugins for Web Browsers
  - Java Web Start
- Does not contain development tools

## JDK - JAVA DEVELOPMENT KIT

- Software used by programmers to develop Java Applications
- Contains JRE and other tools like compiler, debugger
- Current version : Java SE 8 JDK
- Can be downloaded from Oracle Website

<https://www.oracle.com/downloads/index.html>

## JDK EDITIONS

- Java Standard Edition (J2SE)

can be used to develop client-side standalone applications or applets.

- Java Enterprise Edition (J2EE)

can be used to develop server-side applications such as Java servlets and Java Server Pages.

- Java Micro Edition (J2ME).

can be used to develop applications for mobile devices such as cell phones.

- Getting Started

## SETTING UP JAVA

- To Execute java programs, path and classpath system variables need to be set
- **path** variable is used by the operating system to locate executable files
- The executable files of JDK are present in the folder where java is installed

Windows default path: C:\Program Files\Java\JDK<version>\bin

For windows, add the above path to the PATH environment variable

- **classpath** variable is used by compiler and JVM to locate the compiled class files  
( bytecodes)



## WRITING A SIMPLE JAVA PROGRAM

- Open any Text Editor like Notepad and create a class as below

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

## COMPILING A JAVA PROGRAM

### ➤ **Compiling with javac**

- Open a command prompt/Terminal window
- Change to the directory where the program is saved
- Type **javac HelloWorld.java**
- If you get the prompt without any message, the compilation was successful

## EXECUTING A JAVA PROGRAM

### ➤ Executing with java

- Type `java HelloWorld` at the command prompt/Terminal
- The result should be displayed

- Command Line Arguments

## COMMAND LINE ARGUMENTS

- Are used to pass information into a program when it is run.
- Accomplished by passing *command-line arguments* to **main( ) method**
- Arguments immediately follow the program's name on the command line during execution
- Stored as strings in a **String** array passed to the **args** parameter of **main( )**.
- The first argument is stored at **args[0]**, the second at **args[1]**, and so on

## COMMAND LINE ARGUMENTS

➤ The program given below displays the use of command-

```
li class CommandLineDemo {  
    public static void main(String args[]) {  
        System.out.println("First argument " + args[0]);  
        System.out.println("Second argument " + args[1]);  
    }  
}
```

To execute, give the following command with the arguments:

```
java CommandLineDemo Bangalore Hyderabad
```

- **Scanner Class Basics**

## SCANNER CLASS

- Used to read all types of numeric values, strings, and other types of data, from keyboard, file or another source.
- Defined in java.util package
- To use the Scanner class in the application, import the package as given below

```
import java.util.Scanner;
```

- To read from standard input i.e. keyboard, give the following

```
Scanner sc = new Scanner(System.in);  
String s = sc.next();  
int i = sc.nextInt();
```



## Scanner Demo

- This program accepts the student details from console and displays back to console

```
import java.util.Scanner;

class ScannerTest{
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter your rollno");
        int rollno= sc.nextInt();
        System.out.println("Enter your name");
        String name=sc.next();
        System.out.println("Enter your fee");
        double fee=sc.nextDouble();
        System.out.println("Rollno:" + rollno + " name:" + name + " fee:"+fee);
        sc.close();
    }
}
```

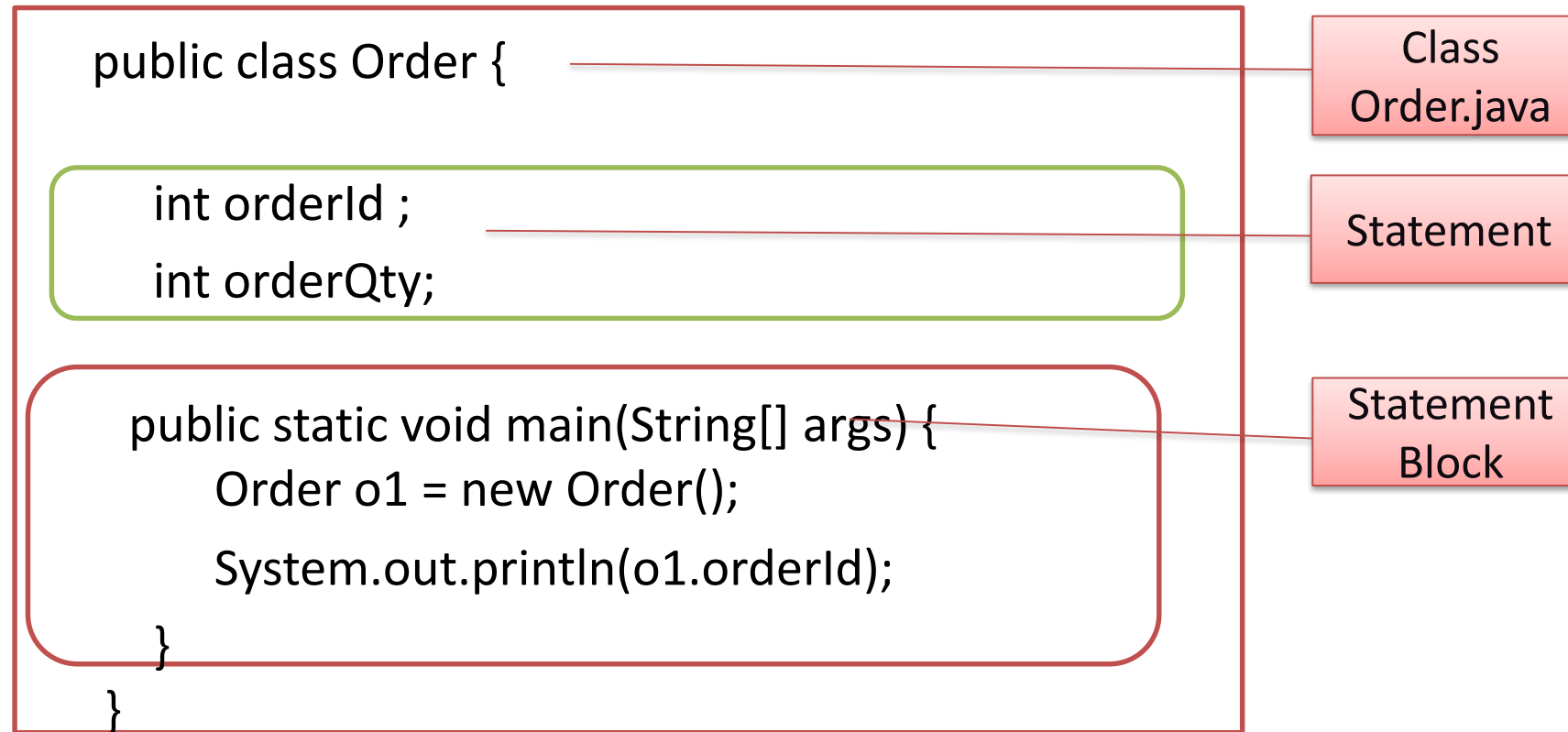
# Data Types, Variables, Operators



## CONCEPT

*Elements of a class*

## BASIC ELEMENTS OF JAVA CLASS



### Statement

- Combination of keywords and variables
- End with a semicolon
- Provide an instruction to the Java interpreter

### Statement block

- Consists of a number of statement's enclosed in curly brackets
- Can be nested within another block

## COMMENTS

- Provide information about the code
- Helps in easily understanding and maintaining code
- Ignored by the java compiler

### **Single Line Comment** - //

```
// Prints value of OrderId
```

### **Multiline Comment** - /\* \*/

```
/* Below Code is used to  
calculate simple interest */
```

- javadoc comments are used by Javadoc tool for creating Java documentation webpages

### **Javadoc comment** - /\*\* \*/

```
/**  
 * Method setQty is used to set the order quantity  
 */
```

## IDENTIFIERS

- Used to uniquely identify variables, methods, classes etc
- Rules
  - Case sensitive
  - Can only begin with a alphabet, “\$”, “\_” | **ex: \$price, \_name**
  - Can contain alphanumeric characters
  - Should not be java keyword | **ex : public, class, float, int**
- Naming Conventions
  - Class name should follow PascalCase | **ex : Order, Account**
  - Methods and variables should follow camelCase | **ex : deposit(), getDetails()**
  - Use full words instead of abbreviations

## KEYWORDS

- Keywords are reserved words that are predefined in the language
- Cannot use keywords as identifiers

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

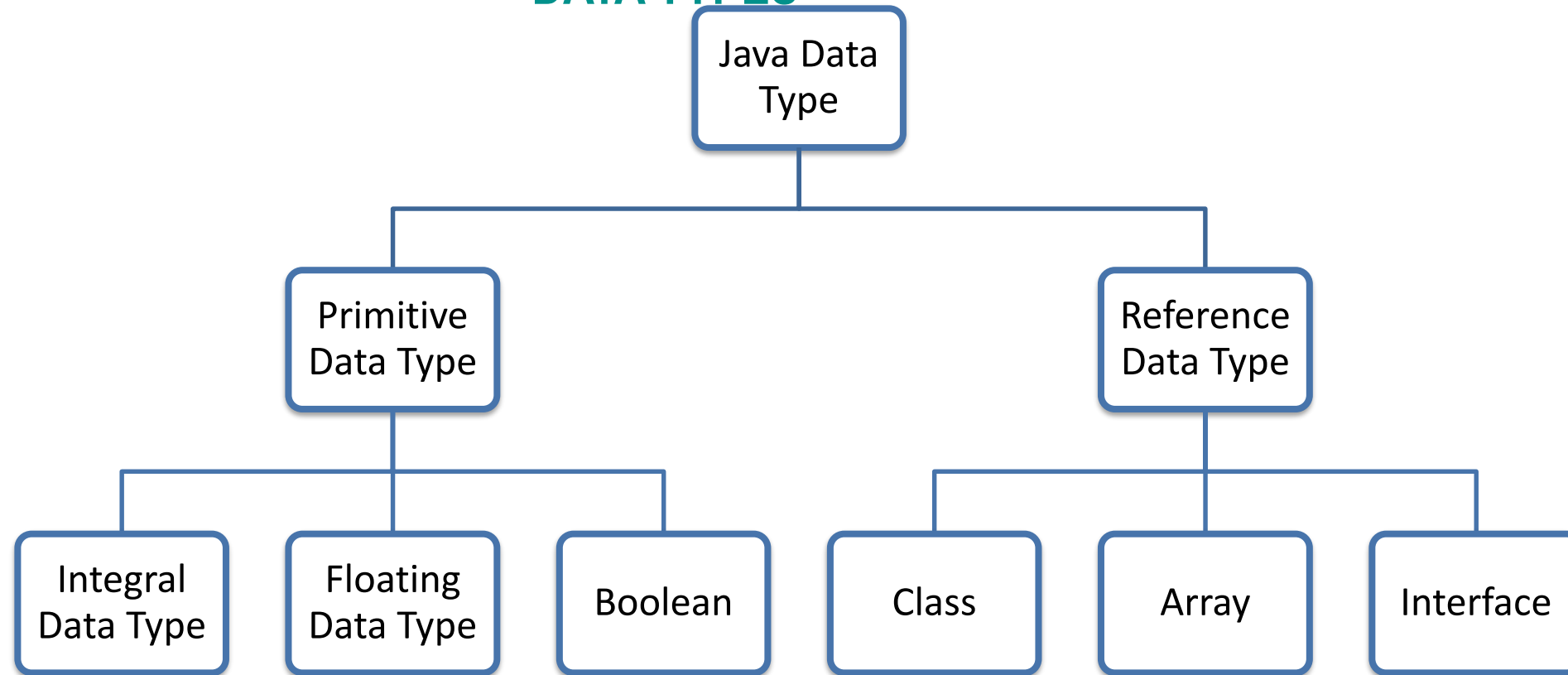


## CONCEPT

# *Data Types and Variables*

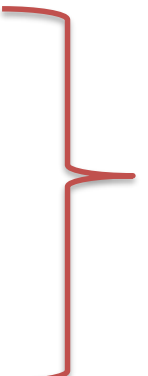




## DATA TYPES



- **Primitive data type** are predefined data types, which hold the value of the declared data type
- **Reference data type** is a data type that is used to store the reference (address) of an object

PRIMITIVE DATA TYPES

Type	Bit depth	Value range	Default value	
byte	8	-128 to 127	0	 Integral
short	16	-32,768 to 32,767	0	
int	32	$(-2^{31})$ to $(2^{31} - 1)$	0	
long	64	$(-2^{63})$ to $(2^{63} - 1)$	0L	
float	32	Varies	0.0f	 Floating
double	64	varies	0.0d	
boolean	Depends on JVM	True or false	false	 Boolean
char	16 unsigned Unicode character	0 to 65535	0	

## VARIABLE

- Basic unit of storage in a Java program
- Container that holds data and can be used throughout a program
- Defined by the combination of a data type, an identifier and an optional initializer

int empld = 1001;

data  
type

identifier

operator

Literal

## VARIABLE DECLARATION

```
byte quantity = 5;
```

byte

quantity

00000101

```
short quantity = 5;
```

short

quantity

000000000000000101

```
int count = 10;
```

```
long noOfEmployees = 1000L; // long literal has to suffixed with 'l' or 'L'
```

```
int hexCount = 0x1F; // hex literal starts with 0x
```

```
int octCount = 011; // octal literal starts with 0
```

```
int octCount = 10_000.00; // numeric literals can have _ as separators
```

*Default value for all the integer literals is integer*

## VARIABLE DECLARATION

```
float price = 1208.976f;    // Float literal has to suffixed with 'f' or 'F'  
double price = 2000.25;  
double price = 2000.25d;  
double range = 2e+05;
```

*Default value for all floating-point literals is double*

```
char gender = 'M';    // character literals should be within single quotes  
char space = '\u0020';  
  
boolean result = true; // valid boolean literal are true or false
```

## FINAL VARIABLES

- Variable can be declared as **final**
- Final variables cannot be changed, once initialized
- Common coding convention is to choose uppercase letters for static and final variable

```
final int BUF_SIZE = 1024;
```

## REFERENCE DATA TYPES

- Store the memory address of an object
- Also known as derived data types
- Java API contains lot of predefined reference data types
  - ex: Array, String
- Classes, interfaces, enums etc. defined by a programmer are called user defined data types
- Variables of reference data types are called reference variables
  - Reference variables can be assigned to null

## PREDEFINED REFERENCE DATA TYPE

- String is a predefined class in Java API
- String is an object that represents a sequence of characters

```
String empName = new String("John");  
  
String name = "Charlie";  
  
String salutation = "Hello" + name;  
  
String first = null;
```



## USER DEFINED REFERENCE DATA TYPES

- Class is an example of user defined data type

```
class Employee {  
    int empId;  
}
```

- Objects of the Employee class can be created in the following way

```
Employee emp1 = new Employee();
```

- **Employee** is the user defined reference data type
- **emp1** is the reference variable



**CONCEPT**

*Operators*

## OPERATORS

- Java provides a wide range of operators for performing variety of operations
- Operators are classified based on the number of operands
  - Unary operators – have single operand
  - Binary operators – have two operands
  - Ternary operators – have three operands



## CONCEPT

*Unary operators*

## UNARY OPERATORS

- Work on single operand
- The following are unary operators
  - Increment and decrement : `expr++`, `expr--`, `++expr`, `--expr`
  - Sign indicator : `+expr`, `-expr`
  - Bitwise Not : `~`
  - Logical Not : `!`

## INCREMENT AND DECREMENT OPERATORS

- The increment operator increases its operand by one.

```
x++;          // equivalent to x = x + 1
```

- The decrement operator decreases its operand by one.

```
x--;          // equivalent to x = x - 1
```

## POSTFIX AND PREFIX

- postfix increment/decrement operator - follow the operand

```
int a = 10, b = 20, y = 0, z = 0;  
y = a++;           // y = 10 and a becomes 11  
z = b--;           // z = 20 and b becomes 19
```

- prefix increment/decrement operator - precede the operand

```
int a = 10, b = 20, y = 0, z = 0;  
y = ++a;           // y and a becomes 11  
z = --b;           // z and b becomes
```

## SIGN INDICATORS

- The operator + and – are used to denote the sign

```
int a = +10;           // '+' is used to represent the value as  
                        a positive number
```

```
int b = -2;            // '-' is used to represent the value as  
                        a negative number
```



## LOGICAL NOT

- Operates only on **boolean** operand.
- Negates the boolean value

```
boolean b1 = true;  
boolean b2 = !b1           // b2 becomes false
```



## CONCEPT

*Binary operators*

## BINARY OPERATORS

Work on two operands on either side of the operator

- Arithmetic
- Relational
- Logical
- Shift and bitwise
- Assignment

## ARITHMETIC OPERATORS

- Used in mathematical expressions
- The operands must be of a numeric type.
- **char** types can be used as it is a subset of int

Description	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%

```
int radius = 0;           // Declare radius
double area = 0;          // Declare area
radius = 20;              // Assign a radius
area = radius * radius * 22 / 7; // Compute area
```

## RELATIONAL OPERATORS

- Used to compare operands
- expression evaluates to true or false

```
int x = 10;  
int y = 5;  
  
(x == y)      false  
(x > y)       true
```

Relational Operators	Description
>	tests if the left-hand value is greater than the right.
>=	tests if the left-hand value is greater than or equal to the right.
<	tests if the left-hand value is less than the right.
<=	tests if the left-hand value is less than or equal to the right.
==	tests equality and evaluates to true when two values are equal.
!=	tests inequality and evaluates to true if the two values are not equal.

## LOGICAL OPERATORS

- logical operators are used with boolean expressions
- Short-circuit operator stops evaluating expression once the result is known

Description	Operator	Short-circuit
Logical And	&	&&
Logical OR		

```
int x = 10;   int y = 5;  int z = 15;

(x > y ) | (x > z)      true
(x > y ) && (x > z)      false
!(x > y)                false
(++x > z) && (++y > z)    false
```

## ASSIGNMENT AND COMPOUND ASSIGNMENT OPERATORS

Operator	Purpose
=	Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

```
int x = 10; int y = 6;  
x += 20;      | x = x + 20 -> 30  
x -= 5        | x = x - 5  -> 5  
x *= y/2      | x = x * (y/2) -> 30
```



**CONCEPT**

*Ternary Operator*



## TERNARY OPERATOR

- Works on three operands
- Replaces if-else statement.
- The general form is *expression1 ? expression2 : expression3*

```
String result = ( a % 2 == 0 ) ? "Even" : "Odd";
```

**equivalent of**

```
String result = null;  
if ( a % 2 == 0 )  
    result = "Even"  
else  
    result = "Odd";
```



**CONCEPT**

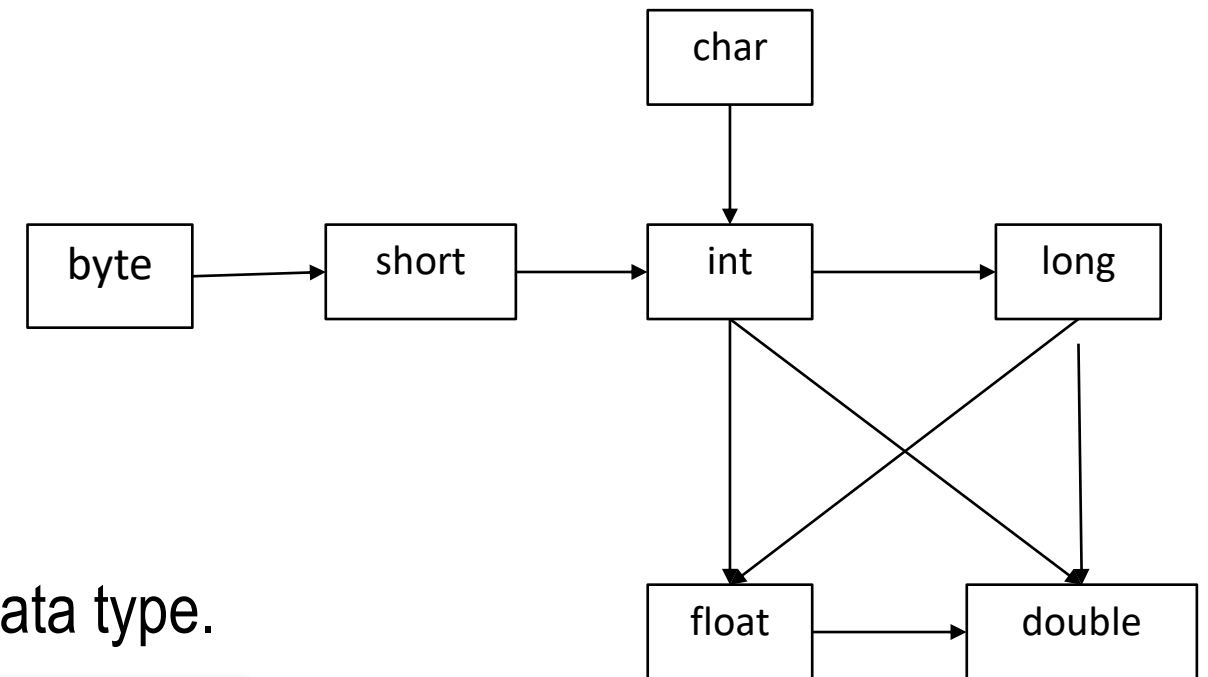
***TYPE CASTING***

## TYPE CASTING

- Type Casting allows to convert primitive values from one data type to another
- Type Casting can be implicit or explicit
- Implicit casting (Widening conversion)
  - Conversion happens automatically
  - Takes place when
    - Two types are compatible
    - Destination data type is larger than the source data type.

```
short s = 200;  
int x = s;
```

```
float f = 100.25f;  
double d = f;
```



## EXPLICIT CASTING (NARROWING CONVERSION)

- When the destination data type is smaller than the source data type, conversion doesn't happen automatically
- Explicit casting needs to be done as shown below

```
long longNum = 1234567;  
int  intNum = longNum;           //won't work, needs explicit casting  
  
int  intNum = (int) longNum;    // explicit casting
```

```
byte b = 50;  
byte c = b * b;                  // error, needs explicit  
casting  
byte c = (byte) (b * b);        // explicit casting
```

## Programming Constructs - Control Structure

- A control structure refers to the way in which the programmer specifies the order of executing the statements
- The following approaches can be chosen depending on the problem statement:
  - Sequential - In a sequential approach, all the statements are executed in the same order as it is written
  - Selectional - In a selectional approach, based on some conditions, different set of statements are executed
  - Iterational (Repetition) - In an iterational approach certain statements are executed repeatedly



**CONCEPT**

*Selectional Constructs*

## SELECTION CONSTRUCT - OVERVIEW

- Selection constructs allows the programmer to control the flow of the program's execution based upon conditions evaluated during run time
- Java supports two selection statements:
  - if – else construct
  - switch - case construct

## IF STATEMENT

- Conditional branching statement
- Can be used to route program execution through two different paths.
- Syntax

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

If the *condition* is true, then *statement1* is executed.  
Otherwise, *statement2* is executed.

```
if (condition) {
    statement1;
    statement2;
}
else {
    statement3;
    statement4;
}
```



## IF STATEMENT

- The following snippet finds whether the given number is odd or even.

```
public static void main(String args[]) {  
    int num = 13;  
    if ( num % 2 == 0 )  
        System.out.println(num + " is even ");  
    else  
        System.out.println(num + " is odd ");  
}
```

## NESTED IF STATEMENT

- An 'if-else' statement embedded within another 'if-else' statement is called as nested 'if'.
- Code snippet to find the maximum among three values

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

## ELSE-IF LADDER CONSTRUCT

- The 'else if' statement is to check for a sequence of conditions
- When one condition is false, it checks for the next condition and so on
- When all the conditions are false the 'else' block is executed
- The statements in that conditional block are executed and the other 'if' statements are skipped

### SYNTAX

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

## ELSE-IF LADDER EXAMPLE

- Code snippet to display the day of the week based on the week code

```
int weekCode = 5;

if ( weekCode == 1 )
    System.out.println("Monday");
else if ( weekCode == 2 )
    System.out.println("Tuesday");
else if ( weekCode == 3 )
    .
    .
    .
else if ( weekCode == 7 )
    System.out.println("Sunday");
```

## SWITCH – CASE CONSTRUCT

- The 'switch' statement is a selectional construct that selects a choice from the set of available choices
- Provides an alternative to if-else-if ladder
- The expression must be of type
  - Java 6 - byte, short, int, char or enum
  - Java 7 – Strings included
- case statements must have literals compatible with the expression

### SYNTAX

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

## SWITCH – CASE EXAMPLE

- Code snippet to display the day of the week based on the week code

```
int weekCode = 5;
switch (weekCode) {
case 1:
    System.out.println("Monday");
    break;
case 2:
    .
    .
case 7:
    System.out.println("Sunday");
    break;
default:
    System.out.println("Invalid code");
}
```

## SWITCH – FALLTHROUGH

- The break statement is optional
- If break is omitted, and a case matches, execution will continue to the next cases below the matched case statement

display week day or week end based on the week code

```
int weekCode = 5;
switch (weekCode) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        System.out.println("Week Day");
    break;
    case 6:
    case 7:
        System.out.println("Week End");
        break;
    default:
        System.out.println("Invalid code");
}
```



**CONCEPT**

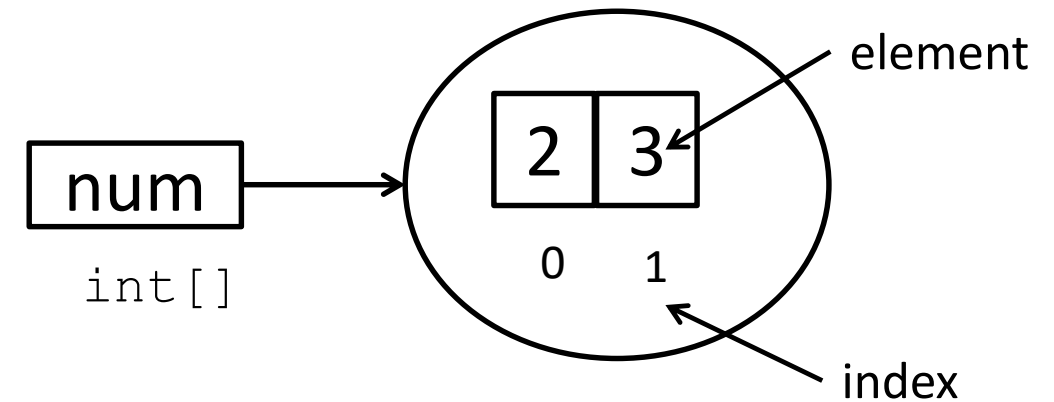
## *Array Basics*



## ARRAY

- Container object that holds a fixed number of values of a same type
- Size of array is fixed after declaration and cannot be changed

```
int[] num = {2,3};    // Array of 2 elements
```



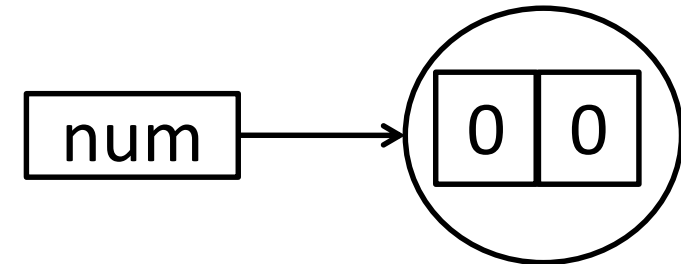
- Elements are accessed using array index

```
num[0] = 5;           //setting element value using index  
  
System.out.println(num[1]); //accessing element using index
```

## ARRAY – CREATING USING NEW OPERATOR

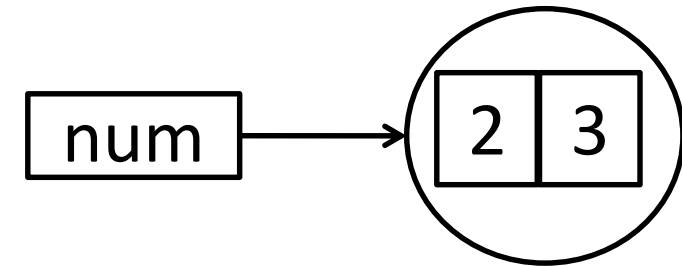
- **new** keyword is used to create an object in Java

```
int[] num = new int[2];
```



- Elements of array are initialized with default values based on the data type

```
num[0] = 2;  
num[1] = 3;
```



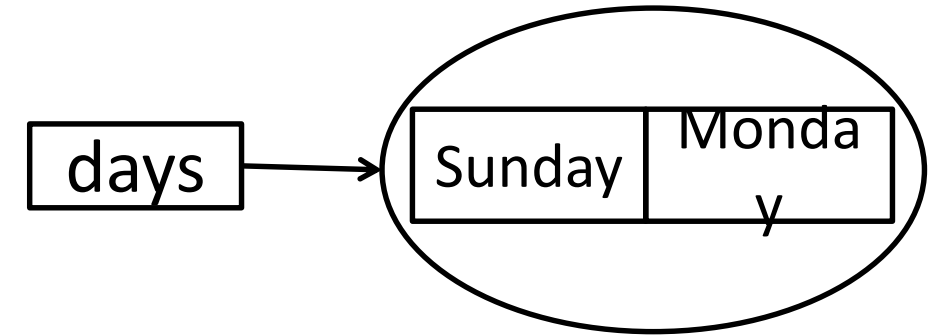
- length property gives the size of the array

```
System.out.println(num.length);    // 2
```

## ARRAY OF REFERENCE DATA TYPES

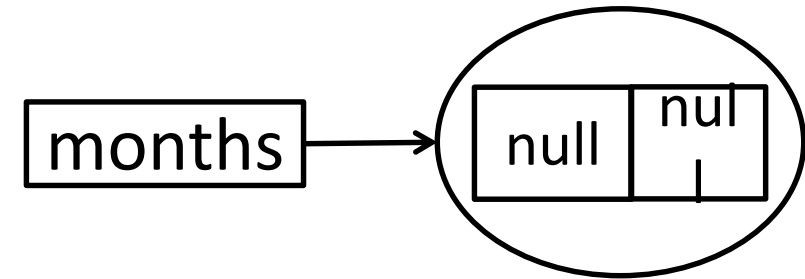
- Array can be of primitive or reference data types

```
String[] days = {"Sunday", "Monday"};
```

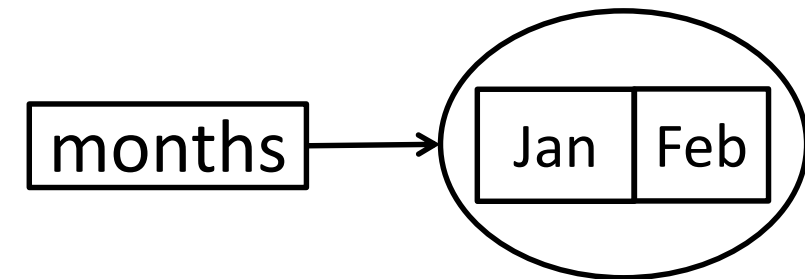


- Creating using new operator

```
String[] months = new String[2];
```



```
months[0] = "Jan";  
months[1] = "Feb";
```





**CONCEPT**

*Iterational Constructs*

## ITERATIONAL CONSTRUCTS - OVERVIEW

- Iterational constructs allow a programmer to repeatedly execute the same set of instructions until a termination condition is met
- Are commonly called as *loops*.
- Java's iterational construct statements are
  - for
  - while
  - do-while

## WHILE LOOP

- Is the most fundamental loop statement.
- The boolean expression is evaluated first and the loop body repeats until the boolean expression is false
- Entry Controlled Loop

```
while(boolean expression) {  
  
    // body of loop  
  
}
```

## DO - WHILE LOOP

- Executes the body of a loop at least once, even if the boolean expression evaluates to false
- Can be used when the condition needs to be checked at the end of the loop rather than at the beginning.
- Exit Controlled Loop

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

## FOR LOOP

- The general form of the **for** loop is given below:

```
for( initialization; condition; iteration )  
    statement;
```

- *The initialization* portion sets the loop control variable to an initial value.
- The *condition* is a boolean expression that checks the loop control variable. If the outcome is true, the loop continues to iterate. Otherwise, the loop terminates.
- The *iteration* expression determines how the loop control variable is changed during each iteration.



## FOR LOOP

- Code snippet to find the sum of first ten natural numbers

```
int sum = 0;
for( int i = 1; i <= 10; i++) {
    sum += i;
}
System.out.println(" The sum is " + sum );
```



**CONCEPT**

*Branching Statements*

## BRANCHING STATEMENTS


- Transfer the control to another part of the program.
- Java supports following branching statements:
  - break
  - continue

## BREAK

- The break statement is used to force the termination of a loop or exit a switch

### Add the numbers entered by user until user enters -1

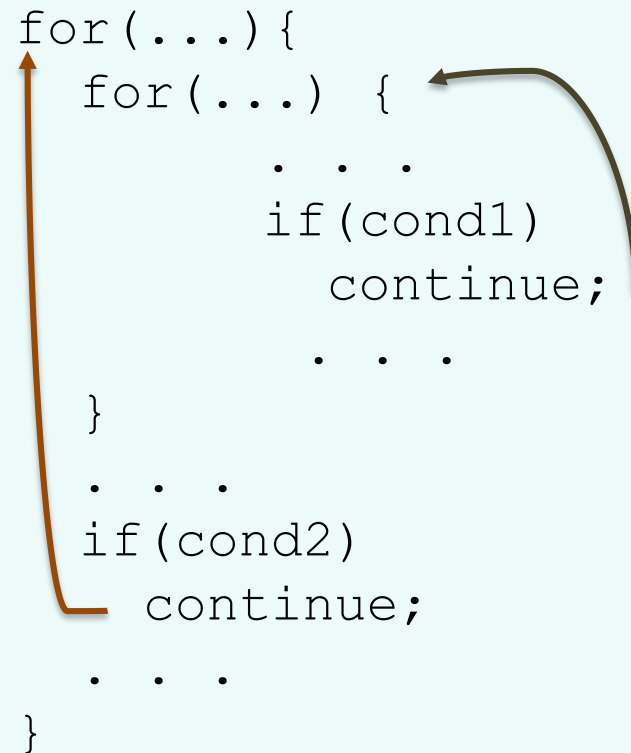
```
Scanner sc= new Scanner(System.in);  
int num = 0; int sum = 0;  
while (true) {  
    System.out.println("Enter the number");  
    num = sc.nextInt();  
    sum += num;  
    if(num == -1) break;  
}  
System.out.println(sum);
```



## CONTINUE

- continue statement forces the next iteration of the loop to take place and skips the code between continue statement and the end of the loop.

```
for (...) {  
    for (...) {  
        . . .  
        if (cond1)  
            continue;  
        . . .  
    }  
    . . .  
    if (cond2)  
        continue;  
    . . .  
}
```



## ENHANCED FOR LOOP

- The enhanced for loop is a specialized for loop that simplifies looping through an array or a collection

```
for(declaration : expression) {  
    ..  
}
```

```
String[] months = {"jan", "feb", "mar"};  
for (String name : months) {  
    System.out.println(name);  
}
```

- **expression:**
  - Must be an array or Collection which has to be looped through
  - Could be a method call that returns an array or collection
- **declaration:**
  - Variable, of a type compatible with the elements of the array or collection

# Arrays



## CONCEPT

*One Dimensional array*



## ARRAYS - OVERVIEW

- Is a group of like-typed variables that are referred to by a common name.
- Offer a convenient means of grouping related information
- May have one or more dimensions.
- A specific element is accessed by its index.
- The first element is always at index 0.

## ARRAYS - OVERVIEW

- The size can be decided at the runtime
- Once initialized, the size cannot grow or shrink.
- All array elements are initialized to the respective default values
- Are initialized while the array is created

# ONE DIMENSIONAL ARRAY

## ➤ Array declaration syntax

```
type var-name[];
```

## ➤ *type* determines the data type of each element of the array

Example:

```
String days[];           //days represents a String array  
int [] num;              //num represents an integer array
```

- The declaration does not actually create an array.
- It declares a reference which can refer to an array object.

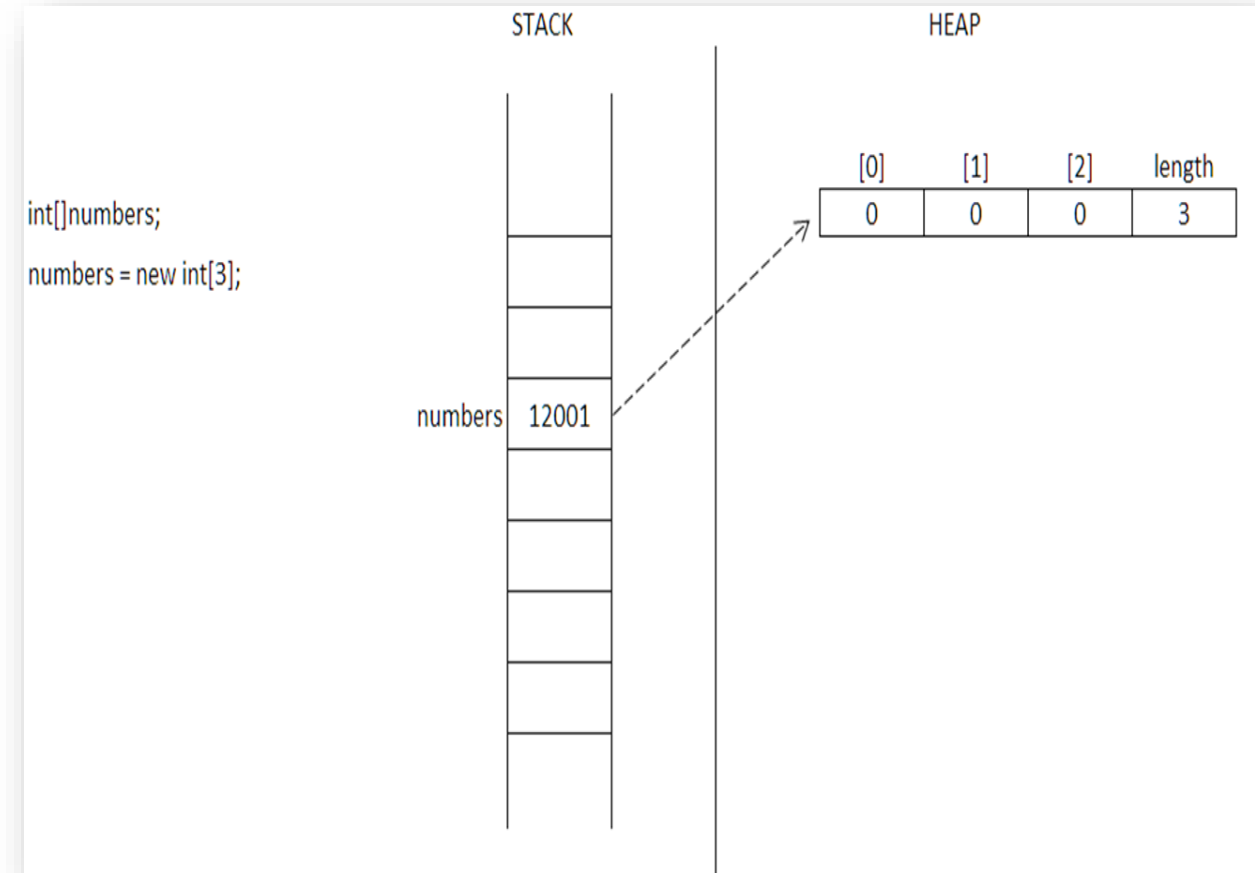
## ONE DIMENSIONAL ARRAY

- Use 'new' keyword to allocate memory for the array elements:

**<var\_name> = new < type> [<array size>];**

```
num = new int[3];
```

- If the array size is negative, a **NegativeArraySizeException** is thrown.



## ONE DIMENSIONAL ARRAY

- Throws `ArrayIndexOutOfBoundsException` when an index greater than the size of the array is specified

Example:

```
int a[ ] = new int[5];  
a[10] = 10;           // array index out of bounds
```

## ONE DIMENSIONAL ARRAY

- Assigning an array initializer list to an array reference after declaration will result in an error.

```
int anArray;  
anArray = {22, 33, 44}; // error
```

- You can use 'new' along with array initializer list to recreate the array.

```
anArray = new int[]{22, 33, 44};
```

## ONE DIMENSIONAL ARRAY

- Code snippet to find the biggest element in the given array;

```
int ar[]={23,3,45,67,89,34};  
int max=ar[0];  
for(int i=1;i<ar.length;i++){  
    if (ar[i] > big)  
        big = a[i];  
}
```



## CONCEPT

*Two Dimensional array*



## TWO DIMENSIONAL ARRAYS

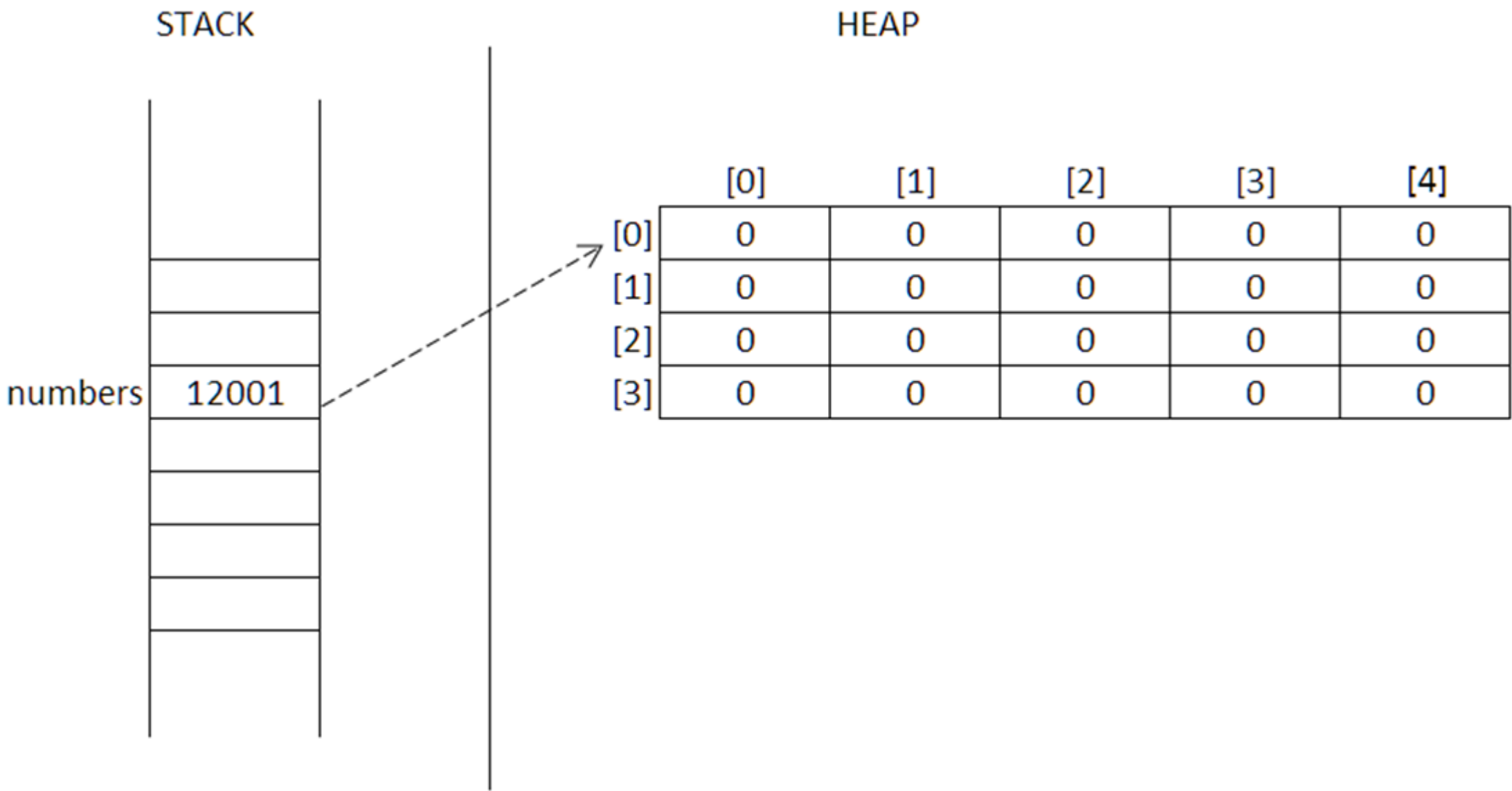
- Is an array of arrays.
- To declare a two dimensional array variable, specify each index using another set of square brackets.
- This allocates a 4 by 5 array and assigns it to matrix.

The following declares a two dimensional array

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

- Internally this matrix is implemented as an *array of arrays* of **int**.

# TWO DIMENSIONAL ARRAYS



## TWO DIMENSIONAL ARRAYS

- Code snippet that prints the array elements in the form a matrix

```
int a[][] = {{1,2,3},{4,5,6},{7,8,9}};  
for (int i =0;i<a.length;i++) {  
    for (int j=0;j<a[i].length;j++) {  
        System.out.print(a[i][j]+"\\t");  
    }  
    System.out.println();  
}
```