

Java Technologies Week 1

Prepared for: DDU CE Semester 4

Prepared by: Prof. Niyati J. Buch

Topics

- Language Fundamentals:
 - The Java Environment: Java Program Development, Java Source File Structure, Compilation Executions,
- Basic Language Elements:
 - Lexical Tokens, Identifiers, Keywords, Literals, Comments, Primitive Data-types, Operators
- Array and String Handling:
 - Array basics, String Array, String class, StringBuffer and StringBuilder class, String Tokenizer Class and Object Class

What is Java?

- Programming Language
 - Java is a high level object oriented computer programming language.
- Platform
 - Any hardware and/or software environment in which program runs, is called platform.
 - Java has its own runtime environment called JRE to execute programs along with libraries (API) to create programs, it is also considered as a platform.

Version	Release date	End of Free Public Updates [1][5][6][7]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018 December 2026 for Azul [8]
Java SE 7	July 2011	July 2019	July 2022
Java SE 8 (LTS)	March 2014	March 2022 for Oracle (commercial) December 2030 for Oracle (non-commercial) December 2030 for Azul May 2026 for IBM Semeru [9] At least May 2026 for Eclipse Adoptium At least May 2026 for Amazon Corretto	December 2030 [10]
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	September 2026 for Azul October 2024 for IBM Semeru [9] At least October 2024 for Eclipse Adoptium At least September 2027 for Amazon Corretto At least October 2024 for Microsoft [11][12]	September 2026 September 2026 for Azul [8]
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK March 2023 for Azul [8]	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	September 2029 for Azul At least September 2027 for Microsoft At least TBA for Eclipse Adoptium	September 2029 or later September 2029 for Azul
Java SE 18	March 2022	September 2022 for OpenJDK	N/A
Java SE 19	September 2022	March 2023 for OpenJDK	N/A
Java SE 20	March 2023	September 2023 for OpenJDK	N/A
Java SE 21 (LTS)	September 2023	September 2028	September 2031 [10]



Java Buzzwords / Design Goals

1. Simple, Object Oriented and Familiar
2. Robust and Secure
3. Architectural Neutral and Portable
4. High Performance
5. Interpreted, Threaded and Dynamic

Simple, Familiar, Object Oriented

- **Simple**
 - Java follows C/C++ languages syntax with OOP concepts.
- **Familiar**
 - C++ was a familiar language
- **Object Oriented**
 - Inheritance, Encapsulation, Polymorphism

Robust and Secure

- **Robust**

- extensive compile-time checking (**strictly typed language**)
- run-time checking (runtime error handling aka **exception handling**)
- memory management. (Objects are created with **new** operator and there are no explicit programmer-defined pointer data types, no pointer arithmetic.)
- automatic garbage collection

Robust and Secure

- **Secure**

- JVM is Java Virtual Machine.
- Any Java program can run on any platform provided JVM is implemented on that platform.
- JVM interprets **bytecode** line-by-line into machine code
- Because JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.
- Since Java 1.1, digitally signed classes are available.
 - With a signed class, you can be sure of who wrote it. If you trust the author of the class, the class can be allowed more privileges on your machine

Architecture Neutral and Portable

- **Architecture Neutral**
 - Because of the bytecode generated by Java compiler which is not dependent on any architecture and can be interpreted on any machine and easily translated into native machine code(this task is done by JVM).
- **Portable**
 - Unlike c/c++, there are no implementation dependent aspects of the specification.
 - The sizes of the primitive data types are fixed, e.g. an int in java is always a 32-bit integer. In C/C++, int can be 16 bit or 32 bit based on the compiler.

High Performance

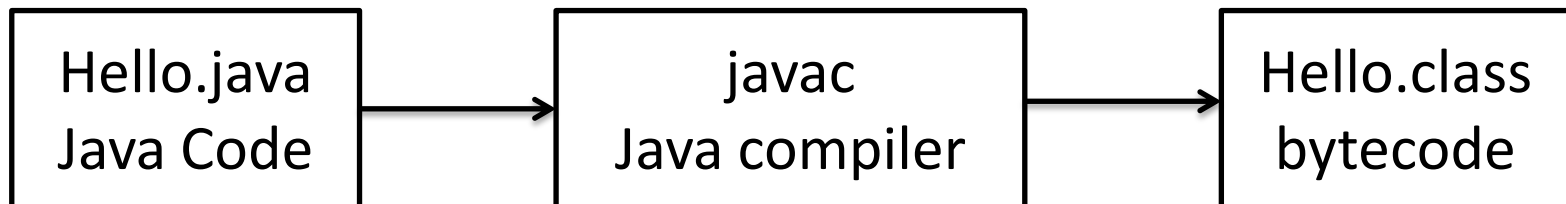
- **JIT (just in time compiler)**
 - selected portions of bytecode are compiled into executable code on demand basis during execution.
 - Most modern JVMs provide JIT support
 - JIT kicks in when JVM determines that some part of the code (usually a method) has been called a certain number of times (JIT threshold)
 - For Hotspot VM:- client mode(1500 invocations), server mode(10,000 invocations), etc.

Interpreted, Threaded, and Dynamic

- **Interpreted**
 - bytecode is line-by-line interpreted on any system that implements the JVM.
- **Threaded**
 - multithreading and multi-process synchronization is available. Though the ultimate implementation is by the underlying operating system.
- **Dynamic**
 - libraries can freely add new methods and instance variables without any effect on their clients.

Bytecode

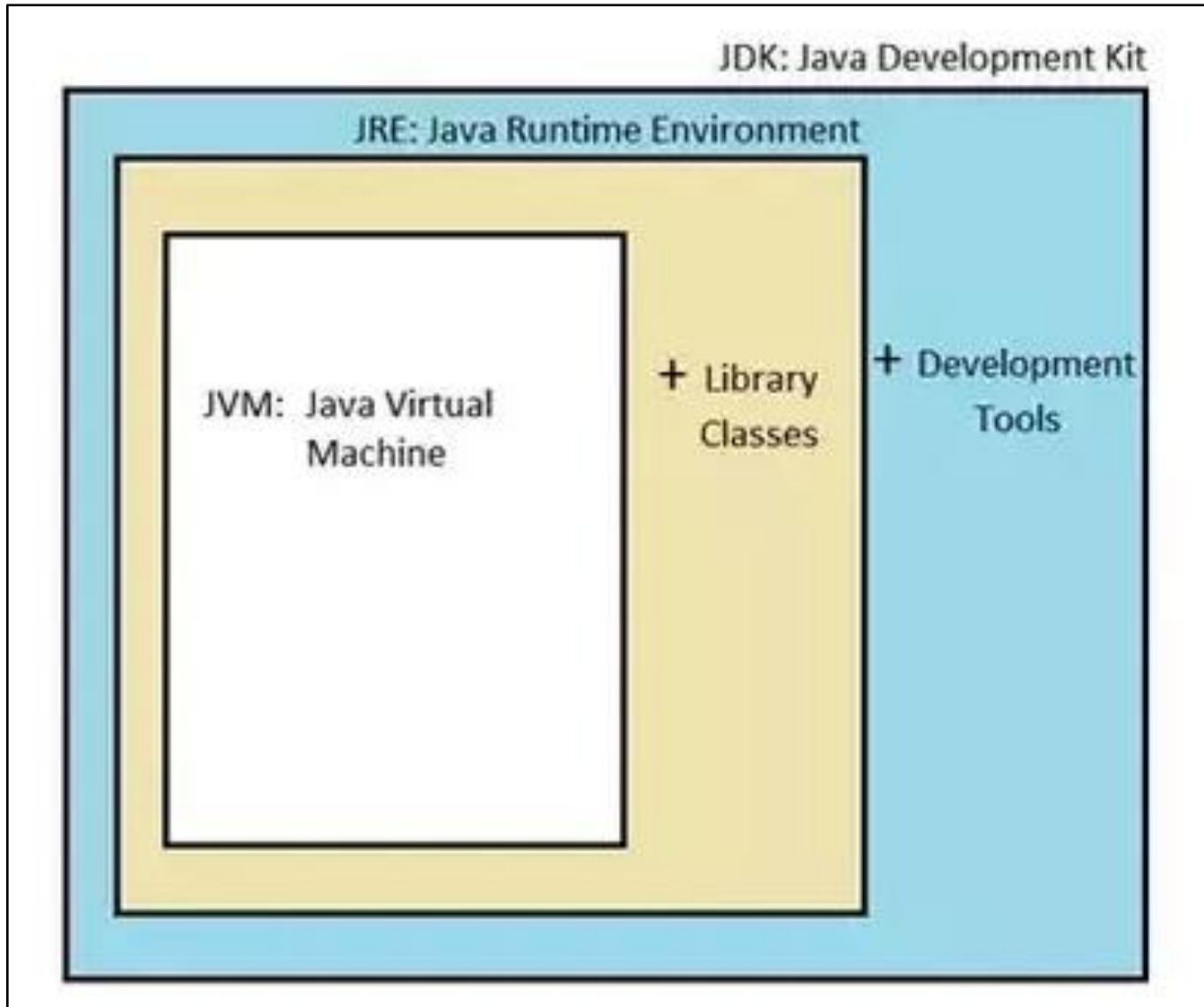
- Bytecode is output of java compiler and it is not an executable code.
- Bytecode is a highly optimized set of instructions designed to be executed by Java runtime system known as JVM (Java Virtual Machine).



JVM, JRE and JDK

- Any Java program can run on any platform provided JVM (**Java Virtual Machine**) is implemented on that platform.
- JVM interprets bytecode line-by-line into machine code.
- JRE (**Java Runtime Environment**) contains JVM and libraries.
- If you want to only run Java programs, JRE is sufficient.
- To develop as well as run you need JDK (**Java Development Kit**).
- It contains JRE and development tools such as compilers and debuggers.

JVM, JRE and JDK



API and IDE

- **API (Application Program Interface)** contains predefined classes and interfaces for developing Java programs.
- Latest API: **Stable release:** Java SE 17
- **An integrated development environment (IDE)** is a software application that provides comprehensive facilities to computer programmers for software development.
- An IDE normally consists of a source code editor, build automation tools, and a debugger

First Java Program

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

- Save program with the file name **MyFirstProgram.java**

- Compile it :

javac MyFirstProgram.java

- Execute it :

java MyFirstProgram

Basic Language Elements

- Lexical Tokens
- Identifiers
- Keywords
- Literals
- Comments
- Primitive Data-types
- Operators

Lexical Tokens

- A **lexical token** is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
- In Java, we have the following lexical tokens
 - Keywords
 - Identifiers
 - Literals
 - Operators
 - Special Symbols

Identifiers

- Identifiers are used to **name** things, such as classes, variables, and methods.
- An **identifier** may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.)
- They **must not begin with a number**, lest they be confused with a numeric literal.
- Beginning with JDK 8, the use of an underscore by itself as an identifier is not recommended.
- Java is **case-sensitive**.
- Keywords (52 keywords) and reserve words (true, false, null) cannot be used as identifiers.

Keywords

- 52 reserved words = 49(in use) + 1(in preview) + 2(not in use)

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

Literals

- A constant value in Java is created by using a *literal representation of it*.
- A **literal** can be used anywhere a value of its type is allowed.
- E.g.
 - 100 is an integer literal
 - 98.2 is a floating point literal (double literal)
 - 98.2f or 98.2F is a float literal
 - 'A' is a character literal
 - "this is a string" is a string literal
 - true and false are boolean literals
 - null is the null literal

Comments

- Single line

//this is a single line comment

- Multi-line

/*this is a multiline comment
with multiple lines

*/

- Documentation comment

/**

*this is a documentation comment

*@author XYZ

*/

Data types in Java

- Java is a **strongly typed language**.
- **Every variable** has a type, **every expression** has a type and every type is **strictly** defined.
- All assignments are **explicit** and parameters can be passed in method call.
- All assignments are **checked for type compatibility** by the Java compiler.

8 Primitive Types

1. Integers

- whole valued signed number
- byte, short, int, long

2. Floating point numbers

- numbers with fractional precision
- float, double

3. Characters

- symbols in character set letters and numbers
- char

4. Boolean

- special type to represent true/false values
- boolean



boolean datatype

- Possible values:- **true, false**
- It is the type returned by all **relational operators**.
- When a boolean value is output by `println()`, "true" or "false" is displayed.
- The value of a boolean variable is sufficient, by itself, to control the if statement.

There is no need to write an if statement like this:

```
if(b == true) ...
```

Why an explicit range and mathematical behaviour of Primitive types???

- In C, the size of integer varies based on execution environment.
- In Java, it is not so. All the data types have strictly defined range.
- **Because of Java's portability requirement.**

Name	Width (bits)	Width (bytes)
long	64	8
int	32	4
short	16	2
byte	8	1
double	64	8
float	32	4
char	16	2

Automatic Type Conversion

- It is possible when both conditions are satisfied:
 1. The two types are compatible.
 2. The destination type is larger than the source type.
- No automatic conversion from integer type to char or boolean
- char and boolean are not compatible with each other.

Casting Incompatible types

- For narrowing conversion, i.e. destination type is smaller than source type cast or explicit type conversion is used.
- **(target-type) value**

Automatic Type Promotion

1. byte, short and char are promoted to int.
2. If any one operand is long, the whole expression is promoted to long;
if one operand is float, the whole expression is promoted to float;
if one operand is double, the whole expression is promoted to double.

Scope and Lifetime of a variable

- **Scope** is the region or section of code where a variable can be accessed.
- A block defines a scope.
- A **block** is begun with an opening curly brace and ended by a closing curly brace.
- **Lifetime** is the time duration where an object/variable is in a valid state/valid memory.

```
class ScopeTest{
    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
        System.out.println("x is " + x); // x is still known here.
    }
}
```

Types of Variables

1. Instance variable
2. Class variable
3. Local variable

Instance Variables

- A variable which is declared inside a class and outside all the methods and blocks is an instance variable.
- The general scope of an instance variable is throughout the class **except in static methods**.
- The lifetime of an instance variable is until the object stays in memory.

Class Variables

- A variable which is declared inside a class, outside all the blocks and is marked **static** is known as a class variable.
- The general scope of a class variable is throughout the class.
- The lifetime of a class variable is until the end of the program or as long as the class is loaded in memory.

Local Variables

- All other variables which are not instance and class variables are treated as local variables including the parameters in a method.
- The scope of a local variable is within the block in which it is declared.
- The lifetime of a local variable is until the control leaves the block in which it is declared.

Question

```
class VariableTest{  
    int a;  
    public static void  
    main(String args[]){  
        int x;  
        {  
            int y;  
        }  
    }  
}
```

- Decide the type of variables: a, x, y
 - Instance variable
 - Class variable
 - Local variable

Answer

```
class VariableTest{  
    int a;  
    public static void  
    main(String args[]){  
        int x;  
        {  
            int y;  
        }  
    }  
}
```

- Decide the type of variables: a, x, y
- a is instance variable
- x and y are local variables

Special Symbols

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.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

Operators

- Arithmetic Operators
- Bitwise Operators
- Relational Operators
- Boolean logical Operators
- Assignment Operator
- ? Operator

Short-Circuit Logical Operators

- The secondary versions of the Boolean AND and OR operators, and are commonly known as **short-circuit logical operators**.
- The **OR** operator results in true when A is true, no matter what B is. Similarly, the **AND** operator results in false when A is false, no matter what B is.
- If you use the **||** and **&&** forms, rather than the **|** and **&** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.
- The formal specification for Java refers to the short-circuit operators as the **conditional-and** and the **conditional-or**.

Precedence of Java operators

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+(unary)	-(unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op=					
Lowest						

Parenthesis

- In addition to **altering the normal precedence** of an operator, parentheses can sometimes be used to help **clarify the meaning** of an expression.
- Adding redundant but clarifying parentheses to complex expressions can help **prevent confusion** later.
- Parentheses (redundant or not) **do not degrade** the performance of your program.
- Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

Control Statements

- **Selection Statements**
 - if
 - switch
- **Iteration Statements**
 - while
 - do-while
 - for
 - for-each
- **Jump Statements**
 - break
 - continue
 - return

For-each version of for loop

- The general form of **the for-each version** of the for is:
for (*type itr-var* : *collection*) statement-block
- Here, ***type*** specifies the type and ***itr-var*** specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
The collection being cycled through is specified by ***collection***.

For-each with arrays

```
int num[]={1,2,3,4,5,6,7,8,9};
int sum = 0;
for(int i=0; i<9; i++)
    sum+=num[i];
```

```
int num[]={1,2,3,4,5,6,7,8,9};
int sum = 0;
for(int x : num)
    sum += x;
```

```
int a[][] = new int [2][3];
for(int i=0; i<2; i++){
    for(int j=0; j<3; j++){
    }
}
```

```
int a[][] = new int [2][3];
for(int x[] : a){
    for(int y : x) {
    }
}
```

An important note for for-each

- The iteration variable is “**read-only**” as it relates to the underlying array.
- An assignment to the iteration variable has no effect on the underlying array.
- In other words, you can’t change the contents of the array by assigning the iteration variable a new value.

```
class NoChange {  
    public static void main(String args[]) {  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        for(int x: nums) {  
            System.out.print(x + " ");  
            x = x * 10; // no effect on nums  
        }  
        System.out.println();  
        for(int x : nums)  
            System.out.print(x + " ");  
        System.out.println();  
    }  
}
```

OUTPUT:

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

break, continue and return

- When used inside a set of nested loops, the **break** statement will only break out of the innermost loop.
- **Labeled break** (as a civilized form of goto)
Syntax: **break label;**
- The label can be any valid Java identifier.
- A labeled break statement can be used to exit from a set of nested blocks.
- Similarly, **continue** can be used with/without a label.
- **return** statement in main() method causes the execution to return to the Java run-time system, since it is the run-time system that calls main().


```
class Break {  
    public static void main(String args[]) {  
        boolean t = true;  
        first: {  
            second: {  
                third: {  
                    System.out.println("Before the break.");  
                    if(t)  
                        break second; // break out of second block  
                    System.out.println("This won't execute");  
                }  
                System.out.println("This won't execute");  
            }  
            System.out.println("This is after second block.");  
        }  
    }  
}
```

OUTPUT:

Before the break.

This is after second block.

```
1. class Return {  
2.     public static void main(String args[]) {  
3.         boolean t = true;  
4.         System.out.println("Before the return.");  
5.         if(t)  
6.             return; // return to caller  
7.         System.out.println("This won't execute.");  
8.     }  
9. }
```

OUTPUT:

Before the return.

```
1. class Return {
2.     public static void main(String args[]) {
3.         boolean t = true;
4.         System.out.println("Before the return.");
5.         //if(t)
6.             return; // return to caller
7.         System.out.println("This won't execute.");
8.     }
9. }
```

OUTPUT: does not compile

Return.java:7: error: unreachable statement

```
    System.out.println("This won't execute.");
    ^
```

1 error

Array

- An **array** is a group of like typed variables that are referred to by a common name.
- Array can have one or more dimension.
- Specific element of array is accessed by its index.

One dimensional array

- **Declaration Syntax : type var_name[];**

E.g.: int marks[];

here value of marks is set as null i.e. array with no value.

- To link marks with an actual physical array of integers, allocation is required using new.
- **new** is a special operator that allocates memory.

array_var = new type[size];

E.g. marks = new int [150];

- Declaration and allocation can be combined:
int marks[] = new int[150];

One dimensional array

- All array indexes start at **zero**.
- Arrays can be initialized when they are declared.
E.g. `int marks[]={74,82,96,100,68,85}`
- The size of array is determined by the number of elements specified in the array initialization and the **new** operator is not required.
- Length/size of the array can be determined by **length property (not length method)**.

Multi-dimensional array

- E.g. `int a[][]=new int[4][5];`
- When you allocate memory for multidimensional array, you need only specify the memory for only specify the memory for the first (leftmost) dimension.
- **Arrays in which different rows have different lengths are called ragged (jagged) arrays.**

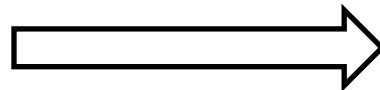
```
int i[][]=new int[4][];
```

```
i[0]=new int[1];
```

```
i[1]=new int[2];
```

```
i[2]=new int[3];
```

```
i[3]=new int [4];
```



```
[0][0]
```

```
[1][0] [1][1]
```

```
[2][0] [2][1] [2][2]
```

```
[3][0] [3][1] [3][2] [3][3]
```

Multi-dimensional array

- It is possible to initialize multi dimensional arrays like:

```
int a[][]={{1,2,3},{4,5,6},{7,8,9}};
```

- Alternative array declaration syntax

```
int a[]=new int[5]; Or int []a=new int[5];
```

- Similarly,

```
int b[][]=new int [4][5]; Or int [][]b = new int[4][5];
```

```
int a[], b[], c[]; Or int [] a,b,c;
```


Questions

1. Which of the two is correct?
 - a) `public static void main(String []args)`
 - b) `public static void main(String args[])`

2. What will be the output?

```
int arr[][] = new int[4][5];  
int length = arr.length;  
System.out.println(length);
```

Answer

1. Which of the two is correct?

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

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

2. What will be the output?

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

```
int length = arr.length;
```

```
System.out.println(length);
```

OUTPUT: 4

Try this too...

```
int arr[][] = new int[4][5];  
int length = arr.length;  
System.out.println(length);  
System.out.println(arr[0].length);  
System.out.println(arr[3].length);
```

```
public class ArrayDemo {  
    int arr[];  
  
    public ArrayDemo() {  
    }  
  
    public ArrayDemo(int[] arr){  
        this.arr = arr;  
    }  
  
    void method1(int a[]) {  
        arr = a;  
    }  
  
    void show() {  
        for (int i = 0; i < arr.length; i++)  
            System.out.println(arr[i]);  
    }  
}
```

```
public static void main(String args[]) {  
    int x[] = {1, 2, 3, 4};  
    ArrayDemo obj1 = new ArrayDemo(x);  
    obj1.show();  
  
    ArrayDemo obj = new ArrayDemo();  
    obj.arr = x;  
    for (int i = 0; i < obj.arr.length; i++) {  
        obj.arr[i]++;  
    }  
    obj.show();  
  
    for (int i = 0; i < x.length; i++) {  
        System.out.println(x[i]);  
    }  
} //end of main  
} //end of class
```

```
public class ArrayDemo {  
    int arr[];  
  
    public ArrayDemo() {  
    }  
  
    public ArrayDemo(int[] arr){  
        this.arr = arr;  
    }  
  
    void method1(int a[]) {  
        arr = a;  
    }  
  
    void show() {  
        for (int i = 0; i < arr.length; i++)  
            System.out.println(arr[i]);  
    }  
}
```

```
public static void main(String args[]) {  
    int x[] = {1, 2, 3, 4};  
    ArrayDemo obj1 = new ArrayDemo(x);  
    obj1.show(); // OUTPUT : 1 2 3 4  
  
    ArrayDemo obj = new ArrayDemo();  
    obj.arr = x;  
    for (int i = 0; i < obj.arr.length; i++) {  
        obj.arr[i]++;  
    }  
    obj.show(); // OUTPUT: 2 3 4 5  
  
    for (int i = 0; i < x.length; i++) {  
        System.out.println(x[i]);  
        // OUTPUT: 2 3 4 5  
    }  
} //end of main  
//end of class
```

Array of objects

- When we say array of objects it is not the object itself that is stored in the array but the reference of the object.

- Syntax:

```
ClassName arr_obj[ ]= new ClassName[Array_Length];
```

- And once an array of objects is instantiated like this, then the individual elements of the array of objects needs to be created using the new keyword.

- i.e.

```
arr_obj[0] = new ClassName();
```

```
...
```

```
...
```

String

- Java's string type, called **String**, is not a primitive type. Nor is it simply an array of characters.
- Rather, String defines an object.
- The **String** type is used to declare string variables. You can also declare arrays of strings.
- A double quoted string constant can be assigned to a String variable.
- A variable of type String can be assigned to another variable of type String.

How to create a String object??

1. By string literal

- e.g. `String s = "DDIT";` or `String s; s="DDIT";`
- `String t = "DDIT";`
- Here new object for `t` is not created. Then what happens?
- Every time a string literal (here `DDIT` is a string literal) is created, JVM checks in String constant pool.
 - If the string already exists in the string constant pool, then a reference to the pooled instance is returned.
 - If the string does not exist in the pool, then new String object is instantiated and placed in the pool.

How to create a String object??

2. By new keyword

```
String s = new String ("DDIT");
```

Or

```
String mystring="DDIT";
```

```
String s=new String (mystring);
```

Or

```
char[] ch = {'D','D','I','T'};
```

```
String s = new String (ch);
```

In Java, String objects are immutable.

- ***Immutable*** simply means un-modifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

```
String text = "Java";
```

```
text.concat("Language");
```

```
System.out.println(text);           //Java
```

```
String text1 = text.concat("Lanaguage");
```

```
System.out.println(text);           //Java
```

```
System.out.println(tex1);           //JavaLanguage
```

String comparison

1. equals()

- returns boolean
- It compares values.
- E.g. `s1.equals(s2)`

2. == operator

- returns boolean
- It compares references not values.
- E.g. `s1 == s2`

3. compareTo()

- returns integer
- If `s1==s2` then zero; if `s1>s2` then +ve value; if `s1<s2` then –ve value
- E.g. `s1.compareTo(s2)`

Output??

```
String s1 = "Hello";  
String s2 = new String("Hello");  
String s3 = "Hello";  
String s4 = new String("Hello");  
System.out.println(s1 == s2);  
System.out.println(s1 == s3);  
System.out.println(s2 == s4);
```

Output

```
String s1 = "Hello";  
String s2 = new String("Hello");  
String s3 = "Hello";  
String s4 = new String("Hello");  
System.out.println(s1 == s2);  
System.out.println(s1 == s3);  
System.out.println(s2 == s4);
```



```
false  
true  
false
```

String concatenation

1. Using + operator

```
String fn = "N", mn = "J", ln = "B";
```

```
String name = fn + mn + ln;
```

```
System.out.println(name); //NJB
```

```
String text = "MyText";
```

```
String newtext = text + 2;
```

```
System.out.println(newtext); //MyText2
```

```
String newtext1 = text + 2 + 2;
```

```
System.out.println(newtext1); //MyText22
```

```
String newtext2 = text + (2 + 2);
```

```
System.out.println(newtext2); //MyText4
```

2. Using concat() method

Some useful public methods

1. `char charAt(int where)` `startIndex`
2. `char[] toCharArray()`
3. `boolean startsWith(String str)`
4. `boolean endsWith(String str)`
5. `int indexOf(int ch)`
6. `int lastIndexOf(int ch)`
7. `int indexOf(String str)`
8. `int lastIndexOf(String str)`
9. `String trim()`
10. `String substring(int`
11. `String substring(int`
`startIndex, int endIndex)`
12. `String replace(char original,`
`char replacement)`
13. `String replace(CharSequence`
`original, CharSequence`
`replacement)`
14. `String replaceAll(String`
`regExp, String newStr)`

Some useful public static methods

- 15. static String valueOf(boolean b) true or false
- 16. static String valueOf(char *ch*)
- 17. static String valueOf(char *chars*[])
- 18. static String valueOf(int *num*)
- 19. static String valueOf(double *num*)
- 20. static String valueOf(float *num*)
- 21. static String valueOf(long *num*)
- 22. static String valueOf(Object *ob*)
- 23. String toLowerCase()
- 24. String toUpperCase()
- 25. static String join(CharSequence delim, CharSequence . . . strs)

Questions

1. If I need to store names of all students, can I create an array of Strings? How?
2. Suppose I want to know the name of the 5th student, how do I access it?

Solution

1. `String []name = new String[50];`

Or

`String[]name={"Bhairavi","Malhar","Ragini","Aarohi","Sargam"};`

2. `System.out.println(name[4]); //Sargam`

StringBuilder-StringBuffer

- **StringBuilder** and **StringBuffer** classes can be used to create **mutable** string.
- Methods like append, insert, reverse, delete, replace returns reference to StringBuffer object which may or may not be assigned to a new variable.
- Similarly, StringBuilder class can be used.

append()

- Example:

```
String s;
```

```
int a = 42;
```

```
StringBuffer sb = new StringBuffer(40);
```

```
s = sb.append("a = ").append(a).append("!).toString();
```

```
System.out.println(s);
```

- Output: **a = 42!**

insert()

- Example:

```
StringBuffer sb = new StringBuffer("I Java!");  
sb.insert(2, "like "); //1st argument is index  
System.out.println(sb);
```

- Output: **I like Java!**

reverse()

- Example:

```
StringBuffer s = new StringBuffer("abcdef");
```

```
System.out.println(s);
```

```
s.reverse();
```

```
System.out.println(s);
```

- Output: **abcdef**
 fedcba

delete()

- Example:

```
StringBuffer sb = new StringBuffer("This is a test.");
```

```
sb.delete(4, 7);
```

```
System.out.println("After delete: " + sb);
```

```
sb.deleteCharAt(0);
```

```
System.out.println("After deleteCharAt: " + sb);
```

- Output: **After delete: This a test.**
 After deleteCharAt: his a test.

replace()

- Example:

```
StringBuffer sb = new StringBuffer("This is a test.");  
sb.replace(5, 7, "was");  
System.out.println("After replace: " + sb);
```

- Output: **After replace: This was a test.**

Some more methods...

- `String substring(int startIndex)`
- `String substring(int startIndex, int endIndex)`
- `int indexOf(String str), int indexOf(String str, int startIndex)`
- `int lastIndexOf(String str), int lastIndexOf(String str, int startIndex)`

String vs. StringBuilder vs. StringBuffer

- If a string is going to *remain constant* throughout the program, then use **String** class object because a String object is immutable.
- If a string can change (example: lots of logic and operations in the construction of the string) and will only be accessed from a *single thread*, using a **StringBuilder** is good enough.
- If a string can change, and will be accessed from *multiple threads*, use a **StringBuffer** because StringBuffer is synchronous so you have thread-safety.

StringTokenizer

- The **StringTokenizer** class allows an application to break a string into tokens.
- The tokenization method is much simpler than the one used by the StreamTokenizer class.
- StringTokenizer implements **Enumeration<> interface**.
- So, methods `hasMoreElements()` and `nextElement()` can be used.
- Also, it has methods like `hasMoreTokens()`, `nextToken()`, `countTokens()`,

Example

```
String s = "this is a test.";
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

OUTPUT:

this

is

a

test.

split() of String class

```
String s = "this is a test.";
String [] tokens=s.split(" ");
for(String t:tokens){
    System.out.println(t);
}
```

OUTPUT:

this

is

a

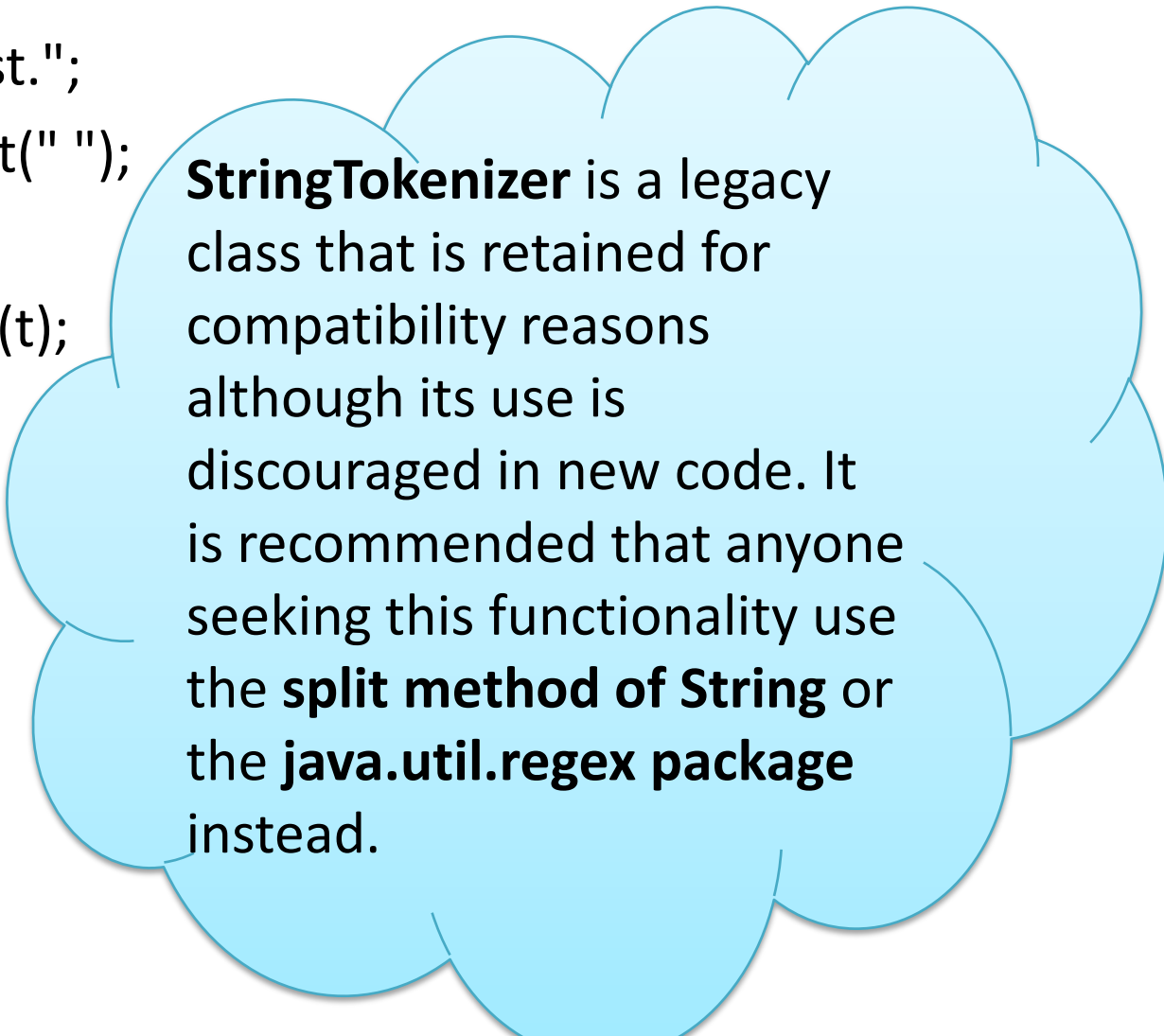
test.

split() of String class

```
String s = "this is a test.";
String [] tokens=s.split(" ");
for(String t:tokens){
    System.out.println(t);
}
```

OUTPUT:

this
is
a
test.



StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the **split method of String** or the **java.util.regex package** instead.

class Class

- Instances of the `java.lang.Class` represent classes and interfaces in a running Java application.
- It has no public constructor. Class objects are constructed automatically by the JVM.
- It is a final class, so we cannot extend it.
- The Class class methods are widely used in Reflection API.

<https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html>

<https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html#defineClass-byte:A-int-int->

Object class

- class **Object** is the root of the class hierarchy.
- Every class has Object as a superclass.
- All objects, including arrays, implement the methods of this class.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>

Modifier and Type	Method	Description
protected Object	clone()	Creates and returns a copy of this object.
boolean	equals(Object obj)	Indicates whether some other object is "equal to" this one.
protected void	finalize()	Deprecated. The finalization mechanism is inherently problematic.
Class<?>	getClass()	Returns the runtime class of this Object.
int	hashCode()	Returns a hash code value for the object.
void	notify()	Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll()	Wakes up all threads that are waiting on this object's monitor.
String	toString()	Returns a string representation of the object.
void	wait()	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> .
void	wait(long timeoutMillis)	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed.
void	wait(long timeoutMillis, int nanos)	Causes the current thread to wait until it is awakened, typically by being <i>notified</i> or <i>interrupted</i> , or until a certain amount of real time has elapsed.

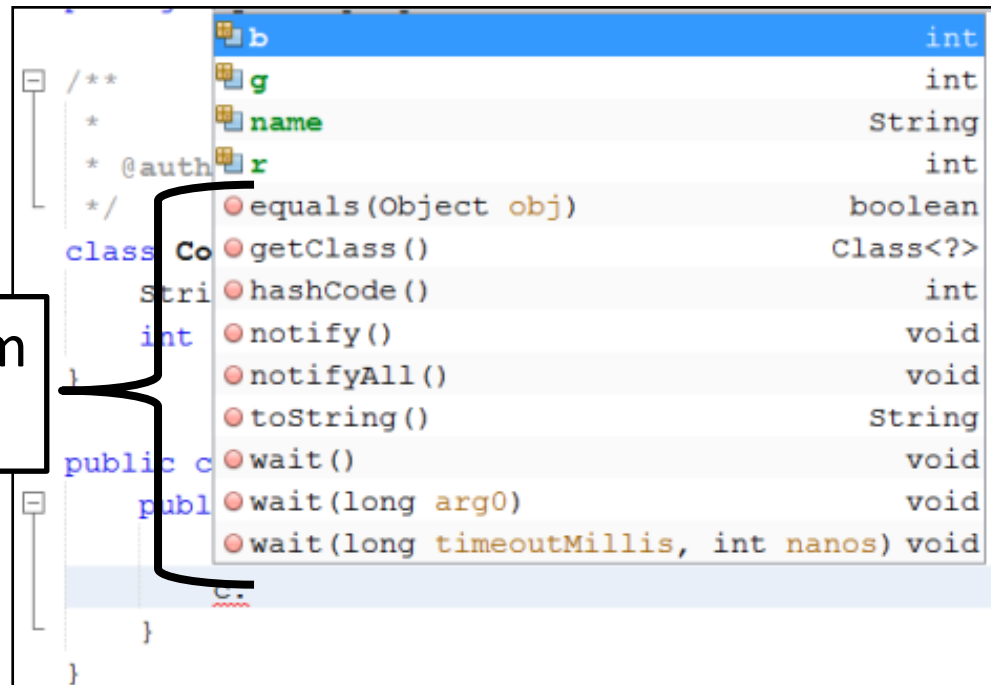
```

class Colours {
    String name;
    int r, g, b;
}

public class ObjectDemo {
    public static void main(String args[]) {
        Colours c = new Colours();
    }
}

```

inherited from
Object class.



```
Colours c = new Colours();  
Colours c1 = new Colours();  
System.out.println("c.hashCode() : " + c.hashCode());  
System.out.println("c.equals(c1) : " + c.equals(c1));  
System.out.println("c.toString() : " + c.toString());  
System.out.println("c : " + c);  
System.out.println("c.getClass().getSimpleName() : " +  
c.getClass().getSimpleName());
```

OUTPUT

```
c.hashCode() : 1562557367  
c.equals(c1) : false  
c.toString() : myfirstproject.Colours@5d22bbb7  
c : myfirstproject.Colours@5d22bbb7  
c.getClass().getSimpleName() : Colours
```

$$(1562557367)_{10} = (5d22bbb7)_{16}$$