Algoritmos e Estruturas de Dados

Universidade do Algarve

Néstor Cataño

nccollazos@ualg.pt

The Java Programming Language

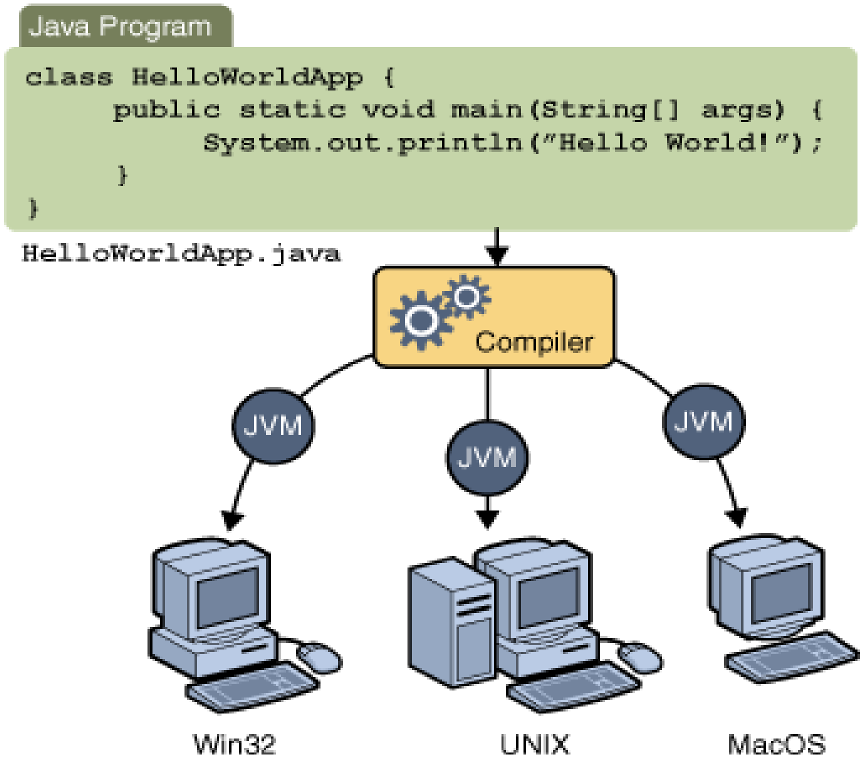
- **Java** is the ***de facto*** language for **web** applications
- **Java** is easy to learn and install, and it’s fun!
- Large number of programming libraries are available
- **Java** is available in many platforms, ***e.g.***, Windows, Linux, Mac. Etc.

Java Technology

- **Java** is a programming Language:
  - **Simple**
  - **Object-Oriented**
  - **Multi-Threaded**
  - **Distributed**
  - **Dynamic**
  - **Portable**
  - **Robust**

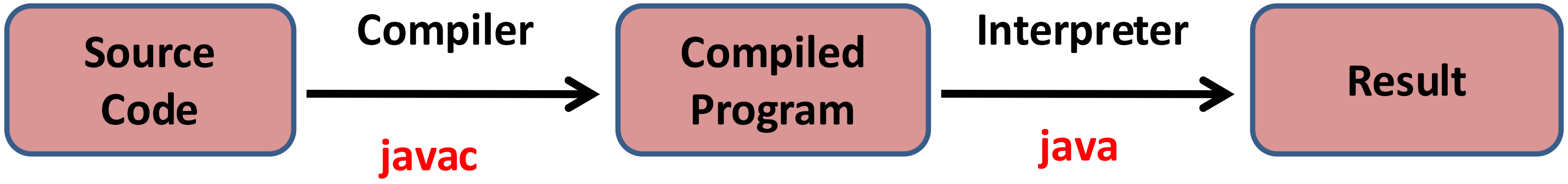
The Java Virtual Machine (JVM)

- Portability. The Java Virtual Machine is available in several platforms: Windows, Linux, Mac, Solaris
- The Java VM allows the program `Program.class` to be run on several platforms



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

Running **Java** Programs



- Programs in **Java** are both **compiled** and **interpreted**
- For example, a **Java** source program saved into a `program.java` file can be compiled by the `javac` Java compiler, which generates the `program.class` file (**Java byte-code**)
- `program.class` is not an executable program; it can be **interpreted** by the `java` command

Demo **Java** Program: “**Hello World**”

1. Getting the program text into the computer,
2. compiling the program, and
3. running the compiled program.

Demo Java Program: “Hello World”

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

Part I

1. Primitive Data Types
2. Expressions and Literals
3. Strings and Chars

Primitive Data Types in **Java**

Type	**Bits**	**Sign**	**Range**	**Default Value**
boolean	1	Signed	false, true	false
byte	8	Two’s Complement	-128…127	0
char	16	Unsigned	‘\u0000’…’\uffff’	‘\u0000’
short	16	Two’s complement	-32768…32767	0
int	32	Two’s complement	2^31…2^31-1	0
long	64	Two’s complement	2^63…2^63-1	0
float	32	32-bit IEEE 754	±3.4E+38 … 1.4E-45	0.0
double	64	64-bit IEEE 754	1.8E+308…5E-324	0.0

Expressions

- An **expression** is a piece of program code that **represents** or **computes** a **value**
- An **expression** can be a **literal**, a **variable**, a **function call**, the value of a **constant**, or several of these combined with operators such as **+**, **>**, etc.
- The **value** of an **expression** can be **assigned to a variable**, used as **parameter** of a function call, or can even be **ignored**

Literal Values

- A **literal** is a name for a constant value
  - `“Hello World”` is a literal of type `String`
  - `‘a’` and `‘b’` are literals of type `char`
  - `315` is a constant of type `int`
  - `315.46` is a constant of type `double`

Integer Literals

- There are three ways to represent **integer** numbers in the Java language
  - **Decimal** (base 10)
  - **Octal** (base 8)
  - **Hexadecimal** (base 16).

Octal Literals

```
class Octal {
 public static void main(String[] args) {
   int six = 06; // it’s equal to decimal 6
   int seven = 07; // it’s equal to decimal 7
   int eight = 010; // it’s equal to decimal 8
   int nine = 011; // it’s equal to decimal 9
   System.out.println("Octal 010 = " + eight);
 }
}
```

Hexadecimal Literals

- The hexadecimal constants are

  `0 1 2 3 4 5 6 7 8 9 a b c d e f`

Hexadecimal Literals

```
class HexTest {
 public static void main(String[] args) {
  int x = 0x0001;
  int y = 0x7fffffff;
  int z = 0xDeadCafe;
  System.out.println("x = " + x + " y = " + y
                   + " z = " + z);
 }
}
```

Literals of Type `char`

- `‘a’` and `‘b’` are **literals** of type `char`
- **Special** char symbols are formed with the aid of the back slash symbol `‘\’`
  - **Back slash** symbol: `‘\\’`
  - **Tab** symbol: `‘\t’`
  - **Carriage return**: `‘\r’`
  - **Line feed**: `‘\n’`

Numerical Constants

- `317` is a constant of type `int`
- `317.65` is a constant of type `double`
- You can make a constant be of type `long` by adding it `L` as a suffix: `317L`
- You can make a constant be of type `float` by adding it `F` as a suffix: `317.65F`
- Floating-point numbers can be expressed in exponential form: `1.5e12`

List of Java **Keywords**

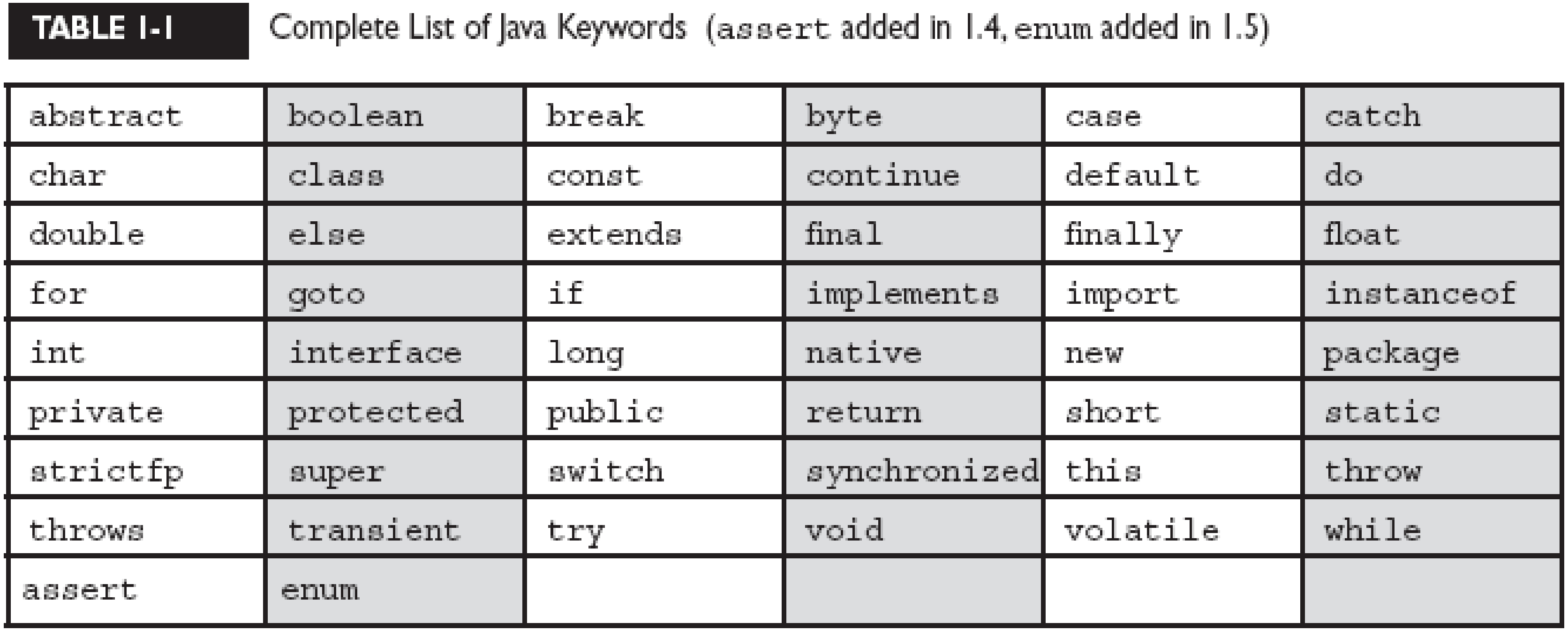


TABLE 1-1 Complete List of Java Keywords (assert added in 1.4, enum added in 1.5)

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

Assignments

- **<variable> = <expression>;**

```
rate = 0.07;
interest = rate * 5;
```

Part I

1. **Primitive Data Types**
2. **Expressions and Literals**
3. **Strings and Chars**

Strings

- **`String x = “Hello World”;`**
  - **`x`** is a literal of type **`String`**

-
  ```
  String x = “Hello”,
         y = “World”,
         z = x + y;
  ```
  - Concatenation of two constants of type **`String`**

Operations on Strings

- `s1.equals(s2)`: returns `true` if `s1` consists of exactly the same sequence of characters as `s2`, and returns `false` otherwise.
- `s1.equalsIgnoreCase(s2)`: checks whether `s1` is the same string as `s2`, but upper and lower case letters are considered to be equivalent.
  - If `s1` is `“cat”`, then `s1.equals("Cat")` is `false`
  - If `s1` is `“cat”`, then `s1.equalsIgnoreCase("Cat")` is `true`

Operations on Strings

- `s1.length()` : returns the number of characters in `s1`.
  - If `s1` is “`hello`” then `s1.length` is 5
- `s1.charAt(n)` : returns the n-th character in the string `s1`
  - If `s1` is “`hello`” then `s1.charAt(0)` is ‘h’ and `s1.chartAt(4)` is `o`
  - an error occurs if `n` less than zero or greater than `s1.length()-1`.

Operations on Strings

- **`s1.substring(n, p)`**: returns a **`String`** consisting of the characters in **`s1`** in positions **`n,n+1,...,p-1`**
  - If s1 is "**`hello`**" then **`s1.substring(1,5`**) returns "**`ello`**"

Operations on Strings

- **`s1.indexOf(s2)`**:
  - If **`s2`** is a substring of **`s1`**, returns an integer that is the starting position of that substring
  - Otherwise, the returned value is -1.
  - If s1 is **`“hello”`** then **`s1.indexof(“ello”)`** is 1

Operations on Strings

- `s1.compareTo(s2)`: compares the strings `s1` and `s2`
  - If `s1` and `s1` are equal, the value returned is zero
  - If `s1` is less than `s2`, the value returned is a number less than zero,
  - If `s1` is greater than `s2`, the value returned is some number greater than zero
  - less than and greater than refer to alphabetical order

Operations on Strings

- **`s1.toUpperCase()`**: returns a new string that is equal to **`s1`**, except that any lower case letters in **`s1`** is converted to upper case
  - **`“Cat”.toUpperCase()`** is the string **`“CAT”`**
  - There is also a function **`s1.toLowerCase()`**

Operations on Strings

- `s1.trim()`: returns a new string that is equal to `s1` except that any **non-printing** characters such as **spaces** and **tabs** are trimmed from the beginning and end of `s1`
- if `s1` has the value “`fred  `”, then `s1.trim()` is the string “`fred`”

String Initialization

```
String s = "Dog";
String s1 = "White " + "Dog";
```

String Initialization

```
String s = "Dog";
String s1 = "White " + "Dog";
System.out.println(s1);
//It prints White Dog
```

char

```
Character.isLetter('d');
// true

Character.isLowerCase('d');
// true

Character.isUpperCase('d');
// false
```

char

```
Character.toLowerCase('D');
//d

Character.toUperCase('d');
//D

Character.isWhitespace('d');
//false

Character.isWhitespace(' ');
//true
```

Part II

1. **Java Operators**
2. **Control Structures**
3. **Arrays**

Arithmetic Operators

Operator	Usage	Description
`+`	`a + b`	Addition between `a` and `b`
`-`	`a - b`	Substraction
`-`	`- a`	Negation of `a`
`*`	`a * b`	`a` times `b`
`/`	`a / b`	`a` divided by `b`
`%`	`a % b`	Module of `a` by `b`

Arithmetic Operators

```
class ArithmeticDemo {
  public static void main (String[] args){
   int result = 3 + 5; // result is now 8
   System.out.println(result);
   result = result * 3; // result is now 24
   System.out.println(result);
   result = result / 2; // result is now 12
   System.out.println(result);
   result = result % 5; // result is now 2
   System.out.println(result);
 }
}
```

Assignment Operators

Operator	Usage	Description
`=`	`a = b`	assigns `b` to `a`
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`

Assignment Operators

```
class ArithmeticDemo {
  public static void main (String[] args){
   int result = 3 + 5; // result is now 8
   System.out.println(result);
   result = result * 3; // result is now 24
   System.out.println(result);
   result = result / 2; // result is now 12
   System.out.println(result);
   result = result % 5; // result is now 2
   System.out.println(result);
 }
}
```

Assignment Operators

```java
class AssignmentDemo {
  public static void main (String[] args){
   int result = 3 + 5; // result is now 8
   System.out.println(result);
   result *= 3; // result is now 24
   System.out.println(result);
   result /= 2; // result is now 12
   System.out.println(result);
   result %= 5; // result is now 2
   System.out.println(result);
 }
}
```

Relational Operators

Operator	Usage	Description
`==`	`a == b`	**a** is equals to **b**
`!=`	`a != b`	**a** is different to **b**
`>`	`a > b`	**a** is greater than **b**
`>=`	`a >= b`	**a** is greater than or equal to **b**
`<`	`a < b`	**a** is less than **b**
`<=`	`a <= b`	**a** is less than or equal to **b**

Relational Operators

```java
class RelationalDemo {
 public static void main(String[] args){
  int value1 = 1; int value2 = 2;
  if(value1 == value2)
   System.out.println("value1 == value2");
  if(value1 != value2)
   System.out.println("value1 != value2");
  if(value1 > value2)
   System.out.println("value1 > value2");
  if(value1 < value2)
   System.out.println("value1 < value2");
  if(value1 <= value2)
   System.out.println("value1 <= value2");
 }
}
```

Logical Operators

Operator	Usage	Description
&&	a && b	Returns **true** if **both** a and b are **true**. If a is **false** or b is **false** returns **false**. If a is **false** a&&b **does not** evaluate b.
&	a & b	Returns **true** if **both** a and b are **true**. If a is **false** or b is **false** returns **false**. If a is **false** a&&b **does** evaluate b.
||	a || b	Returns **true** if a or b are **true**. If a is **true**, it **does not** evaluate b.
|	a | b	Returns **true** if a or b are **true**. If a is **true**, it **does** evaluate b.
!	!a	Negation. Returns **false** when a is **true**, and **true** when a is **false**
^	a ^ b	Exclusive Or. Returns **false** when a and b have the same boolean value, and returns **true** otherwise

Logical Operators

```java
public class OrDemo {
  public static void main(String[] args) {
    int a = 10;
    boolean b = false;

    if( a==10 | (b=true) ) {
     if(b)
       System.out.println("Dog ...");
     else
       System.out.println("Cat ...");
    }
  }
}
```

Logical Operators

```
public class OrDemo {
  public static void main(String[] args) {
    int a = 10;
    boolean b = false;

    if( a==10 | (b=true) ) {
     if(b)
       System.out.println("Dog ...");
     else
       System.out.println("Cat ...");
    }
  }
} ➔ Dog
```

Logical Operators

```java
public class OrDemo {
  public static void main(String[] args) {
    int a = 10;
    boolean b = false;

    if( a==10 || (b=true) ) {
     if(b)
       System.out.println("Dog ...");
     else
       System.out.println("Cat ...");
    }
  }
}
```

Logical Operators

```
public class OrDemo {
  public static void main(String[] args) {
    int a = 10;
    boolean b = false;

    if( a==10 || (b=true) ) {
     if(b)
       System.out.println("Dog ...");
     else
       System.out.println("Cat ...");
    }
  }
} ➔ Cat
```

The Conditional Operator ? :

<Condition> ? <Value 1> : <Value2>

returns <Value 1> if <Condition> is true,
otherwise returns <Value2>

The Conditional Operator ? :

```
class ConditionalDemo {
  public static void main(String[] args){
    int result = (6 == 3*2) ? 5 : 7;
    System.out.println(result);
    //It prints 5
  }
}
```

Increment and Decrement Operators

Operator	Usage	Description

`++`	`x = ++3;`
	`x = 3++;`
		increment (**prefix** and **postfix**) operator

`--`	`x = --5;`
	`x = 5--;`
		decrement (**prefix** and **postfix**) operator

Increment and Decrement Operators

```
class MathDemo {
 public static void main(String[] args) {
  int players = 0;
  System.out.println("players online: "
                    + players++);
  System.out.println("The value of players is "
                   + players);
  System.out.println("The value of players is now "
                   + ++players);
 }
}
```

Increment and Decrement Operators

```
players online: 0
The value of players is 1
The value of players is now 2
```

Precedence Rules

Operator Type	Operator
Unary operators	++, --, !, unary - and +, type-cast
Multiplication and division	*, /, %
Addition and subtraction	+, -
Relational operators	<, >, <=, >=
Equality and inequality	==, !=
Boolean and	&&, &
Boolean or	||, |
Conditional operator	?
Assignment operators	=, +=, -=, *=, /=, %=

Precedence Rules

```
class PrecedenceDemo {
 public static void main(String[] args) {
  int x = 10;
  x *= 2 + 5;
  System.out.println(x);
}
```

Precedence Rules

```
class PrecedenceDemo {
 public static void main(String[] args) {
  int x = 10;
  x *= 2 + 5;
  System.out.println(x);
} ➔ 70
```

Cast

- **Cast** allows you **convert** values from one value to another
- **Casts** can be **implicit** or **explicit**

Implicit **Cast**

```
int a = 100;
long b = a;
// Implicit cast, an int value
// always fits in a long value
```

Explicit **Cast**

```
float a = 100.001f;
int b = (int)a;
System.out.println(b);
```

Explicit **Cast**

```
float a = 100.001f;
int b = (int)a;
System.out.println(b);
// It prints 100
// Explicit cast,
// the float could loose info
```

What Does This Program Print?

```
public class AssgDemo{
 public static void main(String[] args)
  {
    boolean x = false, y = true;

    if (x = y)
     System.out.println("Same");
    else
      System.out.println("Different");
 }
}
```

Part II

1. Java Operators
2. Control Structures
3. Arrays

Control Structures

1. conditional: `if(...){...}else{...}`
2. “for” loop: `for(...){...}`
3. “while” loop: `while(...){...}`
4. “do-while” loop: `do{...}while(...)`
5. “switch” statement: `switch(...){...}`

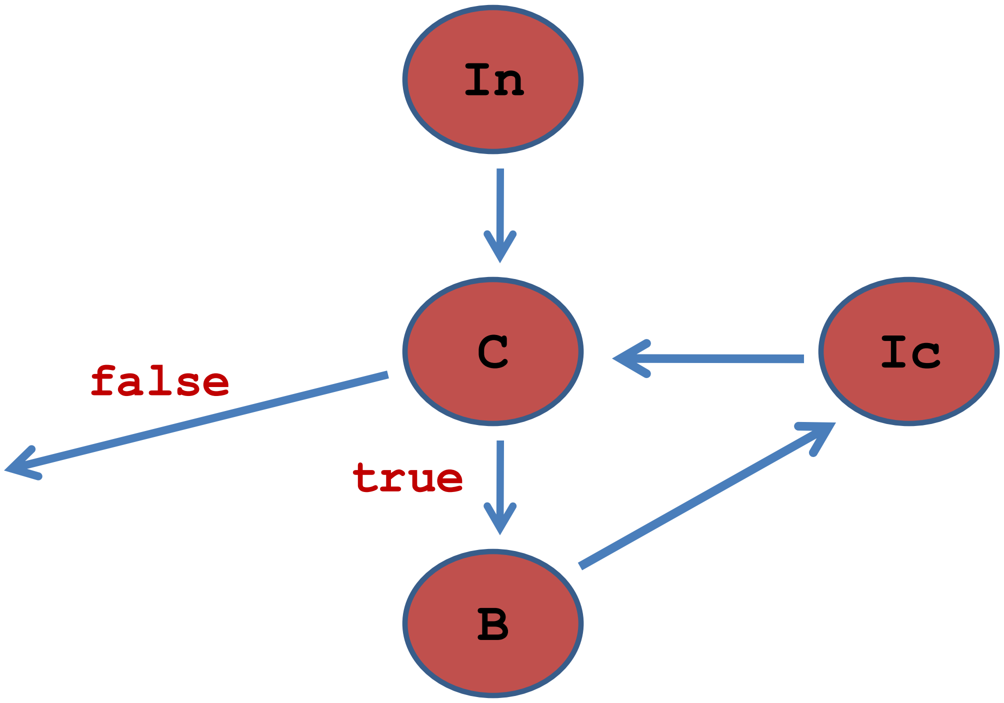
if(...){...}else{...}

```
class IfDemo {
  public static void main(String[] args) {
     char c  = 'i';

     if(c == 'a') { System.out.println("1"); }
     else if(c == 'e') { System.out.println("2");}
     else if(c == 'i') { System.out.println("3");}
     else if(c == 'o') { System.out.println("4"); }
     else if(c == 'u') { System.out.println("5"); }
     else { System.out.println(”6");}
  }
}
```

for(...){...}

```
class ForDemo {
  public static void main(String[] args){
     for(int i=1; i<=10; i++) {
        System.out.println(i);
     }
  }
}
```

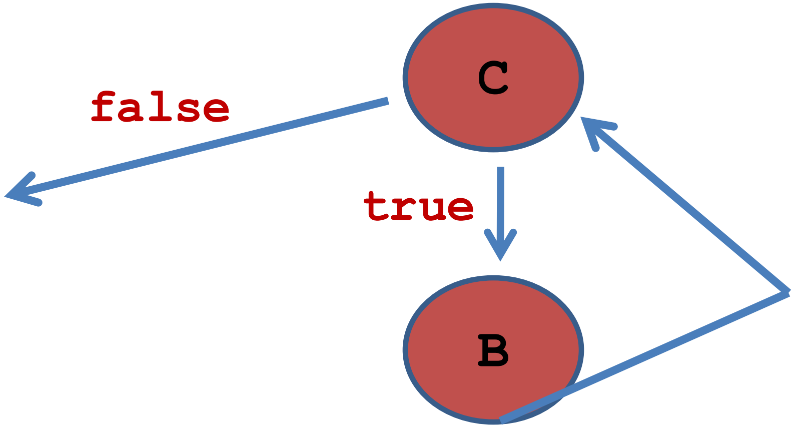
`for(In;C;Ic){B}`



while(...){...}

```
class WhileDemo {
  public static void main(String[] args){
     int i = 1;
      while (i <= 10) {
        System.out.println(i);
        i++;
      }
  }
}
```

while(C){B}



while(C){B}

```
int x = 7;

while(x) { System.out.println("hi"); }
```

while(C){B}

```
int x = 7;

while(x) { System.out.println("hi"); }
//won’t compile; x is not a boolean
```

while(C){B}

```
int x = 7;

while(x = 5){ System.out.println("hi"); }
```

while(C){B}

```
int x = 7;

while(x = 5){ System.out.println("hi"); }
//won’t compile; resolves to 5
```

while(C){B}

```
int x = 7;

while(x == 5){ System.out.println("hi"); }
```

while(C){B}

```
int x = 7;

while(x == 5){ System.out.println("hi"); }
//Legal, equality test
```

while(C){B}

```
int x = 7;

while(true) { System.out.println("hi"); }
```

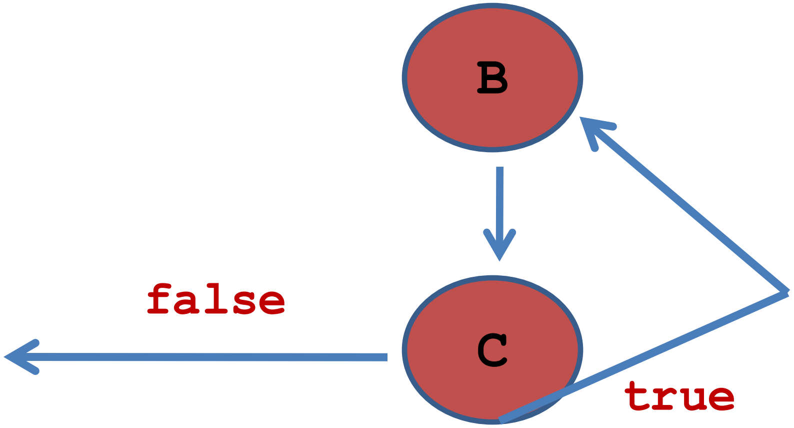
while(C){B}

```
int x = 7;

while(true) { System.out.println("hi"); }
//Legal: infinite loop
```

do{...}while(...)

```
class DoWhileDemo {
  public static void main(String[] args){
     int i = 1;
      do {
        System.out.println(i);
        i++;
      } while (i <= 10);
  }
}
```

`do{...}while(...)`



break

```
class DoWhileDemo {
  public static void main(String[] args){
     int i = 1;
      do {
        if (i==5) break;
        System.out.println(i);
        i++;
      } while (i <= 10);
  }
} ➔
```

break

```
class DoWhileDemo {
  public static void main(String[] args){
      int i = 1;
       do {
         if (i==5) break;
         System.out.println(i);
         i++;
       } while (i <= 10);
   }
} ➔ 1 … 2 … 3 … 4
```

break

```
class WhileDemo {
  public static void main(String[] args){
     int i = 1;
      while (i <= 10) {
        if (i==5) break;
        System.out.println(i);
        i++;
      }
  }
} ➔
```

break

```
class WhileDemo {
  public static void main(String[] args){
     int i = 1;
      while (i <= 10) {
        if (i==5) break;
        System.out.println(i);
        i++;
      }
  }
} ➔ 1 … 2 … 3  … 4
```

break

```
class ForDemo {
  public static void main(String[] args){
     for(int i=1; i<=10; i++) {
        if (i==5) break;
        System.out.println(i);
     }
  }
}
```

switch(...){...}

```
class SwitchDemo {
  public static void main(String[] args) {
     char c  = 'i';

     switch(c) {
       case 'a': System.out.println("1");
       case 'e': System.out.println("2");
       case 'i': System.out.println("3");
       case 'o': System.out.println("4");
       case 'u': System.out.println("5");
       default: System.out.println(”wrong value");
     }
  }
}
```

switch(...){...}

```
class SwitchDemo {
  public static void main(String[] args) {
    char c  = 'i';

    switch(c) {
      case 'a': System.out.println("1");
      case 'e': System.out.println("2");
      case 'i': System.out.println("3");
      case 'o': System.out.println("4");
      case 'u': System.out.println("5");
      default: System.out.println(”wrong value");
    }
  }
} ➔ 3 … 4 … 5 … wrong value
```

switch(...){...}

```
class SwitchDemo {
  public static void main(String[] args) {
     char c  = 'i';

     switch(c) {
       case 'a': System.out.println("1"); break;
       case 'e': System.out.println("2"); break;
       case 'i': System.out.println("3"); break;
       case 'o': System.out.println("4"); break;
       case 'u': System.out.println("5"); break;
       default: System.out.println(”wrong value");
     }
  }
}
```

Example: Prime Numbers

1. To write **prime numbers** up to some **limit**
2. A number is prime if it has exactly **two distinct** natural numbers **divisors**: 1 and itself
   - Primer numbers are: 2, 3, 5, 7, 11, 13, ...

Example: Prime Numbers

```
class PrimeNumbers {
 public static void main(String[] args) {
  ...
 }
}
```

Example: Prime Numbers

```
class PrimeNumbers {
 public static void main(String[] args) {
  int limit = 20;

  for(int i=2; i<=limit; i++) {
    ...
  }
 }
}
```

Example: Prime Numbers

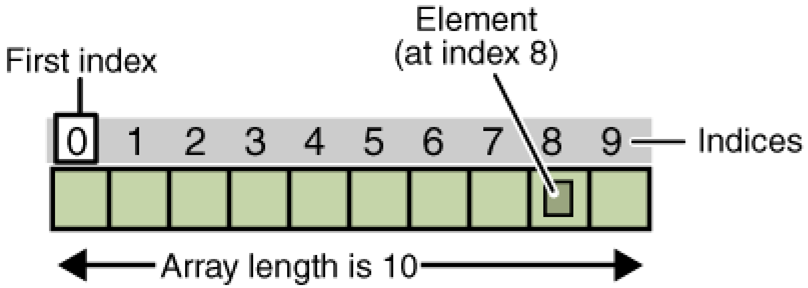
```
class PrimeNumbers {
 public static void main(String[] args) {
  int limit = 20;

  for(int i=2; i<=limit; i++) {
   int k = 2;

   while(i%k != 0)
    k++;
   if(k == i)
    System.out.println(i);
  }
 }
}
```

Part II

1. Java Operators
2. Control Structures
3. **Arrays**

Arrays

- An **array** stores elements of the **same type**
- The elements of an **array** are referenced using a numeric index
- Indexes range from `0` to `(size – 1)`
- Given an array `a`, the elements of the array are `a[0], a[1], a[2],…, a[a.length – 1]`



Arrays

```
class MyClass {
  public static void main(String[] args) {
    char[] a;
    a = new char[11];
    a[0] = 'h';
    a[1] = 'e';
    a[2] = 'l';
    a[3] = 'l';
    a[4] = 'o';
    a[5] = ' ';
    a[6] = 'w';
    a[7] = 'o';
    a[8] = 'r';
    a[9] = 'l';
    a[10] = 'd';
```

```
System.out.print(a[0]);
System.out.print(a[1]);
System.out.print(a[2]);
System.out.print(a[3]);
System.out.print(a[4]);
System.out.print(a[5]);
System.out.print(a[6]);
System.out.print(a[7]);
System.out.print(a[8]);
System.out.print(a[9]);
System.out.print(a[10];
  }
}
```

Arrays

```
class MyClass {
  public static void main(String[] args) {
    char[] a;
    a = new char[11];

    a[0] = 'h';
     ...
    a[10] = 'd';

   for(int i=0; i < 11; i++)
    System.out.print(a[i]);
  }
}
```

Arrays: length

```
class MyClass {
  public static void main(String[] args) {
    char[] a;
    a = new char[11];

    a[0] = 'h';
     ...
    a[10] = 'd';

   for(int i=0; i < a.length; i++)
    System.out.print(a[i]);
  }
}
```

Arrays: Initialization

```
class MyClass {
  public static void main(String[] args) {
    char[] a;
    a = new char[11];

    a[0] = 'h';
     ...
    a[10] = 'd';

   for(int i=0; i < a.length; i++)
    System.out.print(a[i]);
  }
}
```

Arrays: Initialization

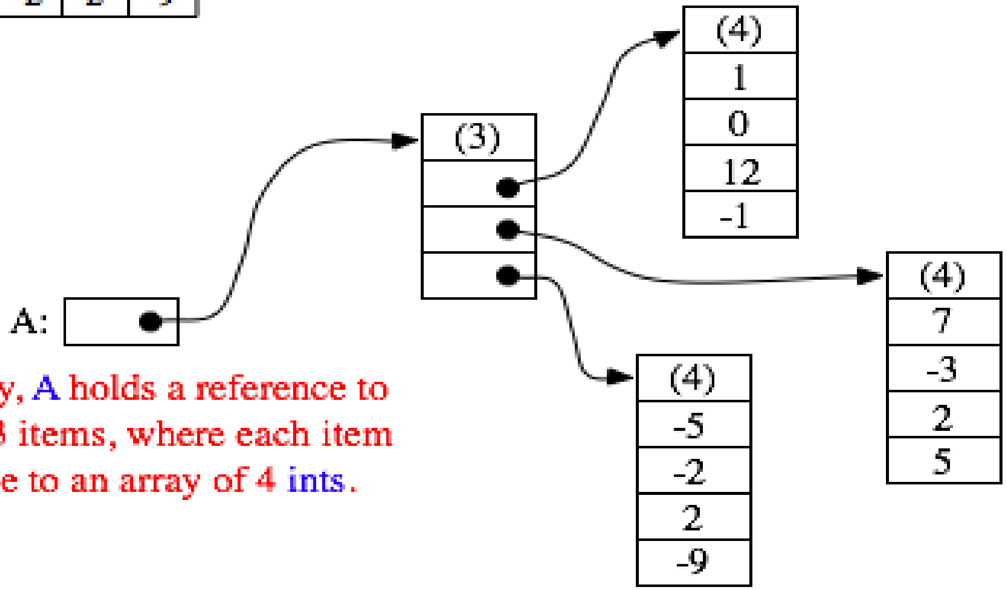
```
class MyClass {
  public static void main(String[] args) {
   char[] a = {'h', 'e', 'l', 'l', 'o', ' ',
               'w', 'o', 'r', 'l', 'd'};

   for(int i=0; i < a.length; i++)
    System.out.print(a[i]);
  }
}
```

Two-Dimensional **Arrays**

If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns.

A:

1	0	12	-1
7	-3	2	5
-5	-2	2	-9



But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

```
int[][] a = { {  1,  0, 12, -1 },
              {  7, -3,  2,  5 },
              { -5, -2,  2, -9 } };
```

Two-Dimensional **Arrays**

```
class Array2D {
  public static void main(String [] args) {
    int[][]  a  =  {  {  1,  0, 12, -1 },
                      {  7, -3,  2,  5 },
                      { -5, -2,  2, -9 } };
    for(int fila=0; fila<3; fila++) {
     for(int coluna=0; coluna<4; coluna++)
        System.out.print(a[fila][coluna] +" ");
     System.out.print("\n");
    }
  }
}
```

Two-Dimensional **Arrays**

```
class Array2D {
  public static void main(String [] args) {
    int[][]  a  =  {  {  1,  0, 12, -1 },
                      {  7, -3,  2,  5 },
                      { -5, -2,  2, -9 } };
    for(int fila=0; fila<3; fila++) {
     for(int coluna=0; coluna<4; coluna++)
        System.out.print(a[fila][coluna] +" ");
     System.out.print("\n");
    }
  }
} // Error ...
```

Two-Dimensional **Arrays**

```
class Array2D {
  public static void main(String [] args) {
    int[][]  a  =  {  {  1,  0, 12, -1 },
                      {  7, -3,  2,  5 },
                      { -5, -2,  2, -9 } };
    for(int fila=0; fila<a.length; fila++) {
     for(int coluna=0; coluna<a[fila].length; coluna++)
        System.out.print(a[fila][coluna] +" ");
     System.out.print("\n");
    }
  }
} /O.K. ...
```

Plan 3

1. Program Execution
2. Ordered Strings
3. Multiplying Matrices
4. Palindrome
5. Operations on Bits

Program Execution

Programs are executed from the `main` method

```
class ProgramExec {
 public static void main(String[] args) {
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

Program Execution

```
class ProgramExec {
 public static void main(Strin
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

Local variable declaration

Program Execution

```
class ProgramExec {
 public static void main(Strin
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

Method Call: *Suma5* **is a method, and k is an actual parameter**

Program Execution

```
class ProgramExec {
 public static void main(Strin
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

Method Call: ***Suma5*** **is a method, and `k` is an actual parameter with value 7**

The result of `Suma5(k)`'s call is assigned to `res`

Program Execution

```
class ProgramExec {
 public static void main(String[] args) {
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

j is a formal parameter.
Inside Suma5, k is named j

Program Execution

```
class ProgramExec {
 public static void main(String[] args) {
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

This method returns an integer value

Program Execution

```
class ProgramExec {
 public static void main(String[] args) {
  int k = 7;
  int res = Suma5(k);

  System.out.println(res);
 }

 public static int Suma5(int j) {
  int temp = j + 5;
  return temp;
 }
}
```

Returns temp and exits the method call

Part III

1. **Program Execution**
2. **Ordered Strings**
3. **Multiplying Matrices**
4. **Palindrome**
5. **Operations on Bits**

Ordered Strings

1. To write a program that prints a list of `String`s orderly
2. Use an **Array** of `String` to store your strings

Ordered Strings

```
public class OrderedString {
 public static void main(String[] args) {
  String[] words = {"pato", "gato", "cachorro",
                    "formiga", "cavalo"};
  ...
 }
}
```

```java
public class OrderedString {
 public static void main(String[] args) {
  String[] words = {"pato", "gato", "cachorro",
                    "formiga", "cavalo"};
  boolean swapped;
  do {
   swapped = false;

   // swap elements in array words

  }
  while(swapped);
 }
}
```

```java
public class OrderedString {
 public static void main(String[] args) {
  String[] words = {"pato", "gato", "cachorro",
                    "formiga", "cavalo"};
  boolean swapped;
  do {
   swapped = false;
   for(int i=0; i < words.length-1; i++) {
    if(words[i].compareTo(words[i+1]) > 0) {
     // swap words[i] and words[i+1]
     swapped = true;
    }
   }
  }
  while(swapped);
 }
}
```

```java
public class OrderedString {
 public static void main(String[] args) {
  String[] words = {"pato", "gato", "cachorro",
                    "formiga", "cavalo"};
  boolean swapped;
  do {
   swapped = false;
   for(int i=0; i < words.length-1; i++) {
    if(words[i].compareTo(words[i+1]) > 0) {
     String tmp = words[i];
     words[i] = words[i+1];
     words[i+1] = tmp;
     swapped = true;
    }
   }
  }
  while(swapped);
 }
}
```

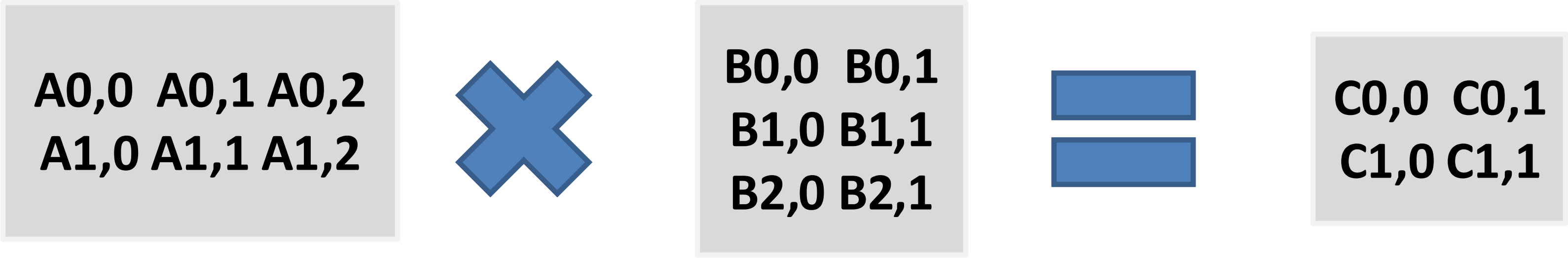
Part III

1. Program Execution
2. Ordered Strings
3. Multiplying Matrices
4. Palindrome
5. Operations on Bits

Multiplying Matrices

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \times 3 + 0 \times 2 + 2 \times 1 & 1 \times 1 + 0 \times 1 + 2 \times 0 \\ -1 \times 3 + 3 \times 2 + 1 \times 1 & -1 \times 1 + 3 \times 1 + 1 \times 0 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

Multiplying Matrices



A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

C0,0 C0,1
C1,0 C1,1

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

```
C0,0  = A0,0 * B0,0 +
        A0,1 * B1,0 +
        A0,2 * B2,0
```

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

C0,0 = A0,0 * B0,0 +
A0,1 * B1,0 +
A0,2 * B2,0

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

C0,1 = A0,0 * B0,1 +
A0,1 * B1,1 +
A0,2 * B2,1

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

```
C1,0  = A1,0 * B0,0 +
        A1,1 * B1,0 +
        A1,2 * B2,0
```

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

C1,0 = A1,0 * B0,0 +
A1,1 * B1,0 +
A1,2 * B2,0

```
C[?][?] += A[?][?] * B[?][?];
```

Multiplying Matrices

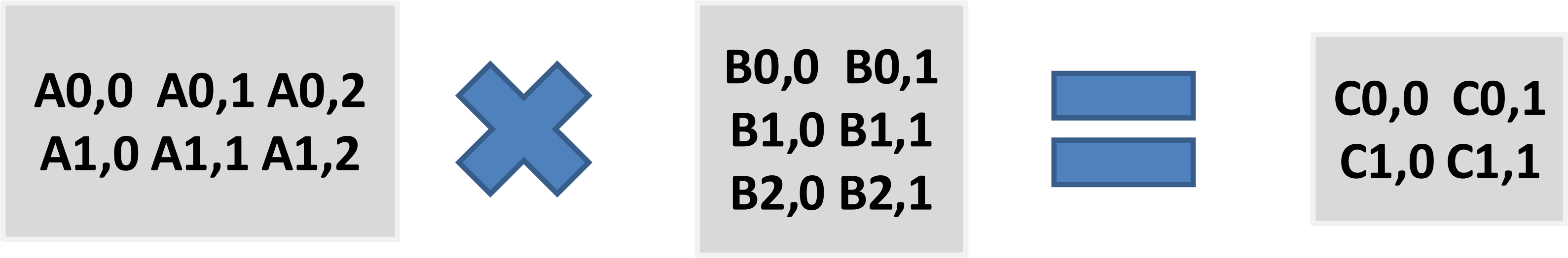
$$\begin{matrix} \text{A0,0} & \text{A0,1} & \text{A0,2} \\ \text{A1,0} & \text{A1,1} & \text{A1,2} \end{matrix} \times \begin{matrix} \text{B0,0} & \text{B0,1} \\ \text{B1,0} & \text{B1,1} \\ \text{B2,0} & \text{B2,1} \end{matrix} = \begin{matrix} \text{C0,0} & \text{C0,1} \\ \text{C1,0} & \text{C1,1} \end{matrix}$$

```
C1,0  = A1,0 * B0,0 +
        A1,1 * B1,0 +
        A1,2 * B2,0
```

```
C[i][?] += A[i][?] * B[?][?];
```

```
i < A.length
```

Multiplying Matrices



A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

C0,0 C0,1
C1,0 C1,1

C1,0 = A1,0 * B0,0 +
A1,1 * B1,0 +
A1,2 * B2,0

```
C[i][?] += A[i][j] * B[j][?];
```

```
i < A.length
j < B.length
```

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

C1,0 = A1,0 * B0,0 +
A1,1 * B1,0 +
A1,2 * B2,0

```
C[i][k] += A[i][j] * B[j][k];
```

```
  i < A.length
  j < B.length
k < B[j].length
```

Multiplying Matrices

A0,0 A0,1 A0,2
A1,0 A1,1 A1,2

×

B0,0 B0,1
B1,0 B1,1
B2,0 B2,1

=

C0,0 C0,1
C1,0 C1,1

C1,0 = A1,0 * B0,0 +
A1,1 * B1,0 +
A1,2 * B2,0

```
for(int i = 0; i < A.length; i++)
 for(int j = 0; j < B.length; j++)
  for(int k = 0; k < B[j].length; k++)
    C[i][k] += A[i][j] * B[j][k];
```

Multiplying Matrices

```
class MatrixMul {
 public static void main(String[] args) {
  int[][] A = { {1, 0, 2}, {-1, 3, 1} };
  int[][] B = {{3, 1}, {2, 1}, {1, 0} };
  int x = A.length, y = b[0].length;
  int[][] C = new int[x][y];

  for(int i = 0; i < A.length; i++)
   for(int j = 0; j < B.length; j++)
    for(int k = 0; k < y; k++)
     C[i][k] += A[i][j] * B[j][k];
 }
}
```

Plan 3

1. Program Execution
2. Ordered Strings
3. Multiplying Matrices
4. Palindrome
5. Operations on Bits

Palindrome

- Write a program in **Java** that prints **Yes** if a word taken as a parameter is a **palindrome** and **No** if it is not
- A **palindrome** is a word that reads exactly the same from the beginning to the end than from the end to the beginning
  - ***abcba*** is a **palindrome**
  - ***abcdba*** is **not** a **palindrome**

Palindrome

```
class Palindrome {
 public static void main(String[] args) {
   String s = args[0];
   int left = 0, right = s.length() – 1;
   boolean b = false;

   ...
}
```

Palindrome

```
class Palindrome {
 public static void main(String[] args) {
   String s = args[0];
   int left = 0, right = s.length() - 1;
   boolean b = false;
   while(left < right && !b) {
     if(s.charAt(left) != s.charAt(right)) {
       System.out.println("No");
       b = true;
     } else { left++; right--; }
   }
   if(!b) System.out.println("Yes");
 }
}
```

Plan 3

1. Program Execution
2. Ordered Strings
3. Multiplying Matrices
4. Palindrome
5. Operations on Bits

Decimal Numbers	Numbers over 4 bits
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Operations on Bits: Addition

```
 1101   (carry)
  0101 (5)
+ 1101 (-3)
==============
  0010 (2) correct

 0111   (carry)
  0111 (7)
+ 0011 (3)
==============
  1010 (-6) overflow

 0100   (carry)
  0111 (7)
+ 0100 (4)
==============
  1011 (-5) overflow
```

What Does This Program Print?

```
public class ByteDemo {
  public static void main(String[] args)
  {
    byte b = 126;
    b = (byte)(b + 5);
    System.out.println(b);
  }
}
```

What Does This Program Print?

```
public class ByteDemo {
  public static void main(String[] args)
  {
    byte b = 126;
    b = (byte)(b + 5);
    System.out.println(b);
  }
} ➔ -125
```