

# Basics of Java programming

# Objectives

- Write Java program using basic elements like variables, constants, operators and loops
- Apply conversions

# Table of Content

Variable Naming Convention	Shift operators
Reading input from console	Conversions
Keywords/ Reserved words	Common Errors
Arithmetic Operators	for loop omitting options
Conditional Logical Operators	Labeled continue/break statements

- Primitive data types are basic data types.
  - Integer type: **byte**, **short**, **int**, **long**
  - Floating point types: **float**, **double**
  - Character data types : **char**
  - Boolean data type: **boolean**

Type	Size in Bytes	Min Range	Max Range
byte	1	$-2^7$	$2^7-1$
short	2	$-2^{15}$	$2^{15}-1$
int	4	$-2^{31}$	$2^{31}-1$
long	8	$-2^{63}$	$2^{63}-1$
char	2	0	$2^{16}-1$
float	4		
double	8		
boolean	JVM Specific (typically 1)		

- UNICODE is a 16 bit character.
- They are generally represented in hexadecimal format.
- ‘\u’ in beginning of the character is used to represent hexadecimal character.
- The characters represented include all basic English letters, numbers, special characters and characters from other languages also !
- ‘A’ → ‘\u0041’
- The Unicode Standard encodes characters in the range U+0000..U+10FFFF

 *What is Unicode for F?*

You could write the entire java code as  
**\u0069\u006e\u0074 \u0061;**  
This above code represents  
**int a;**



- Variable name must begin with
  - a letter (A-Z, a-z, or any other language letters supported by UTF 16)
  - An underscore (\_)
  - A dollar (\$)after which it can be sequence of letters/digits.
- Digits: 0-9 or any Unicode that represents digit.
- Length of the variable name is unlimited
- Java reserved words should not be used as variable names.

# Variable Naming Convention

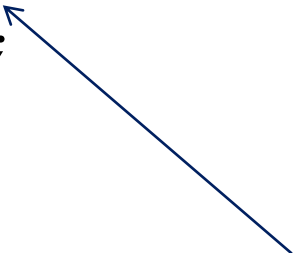
- Must begin with lower case
- Must always begin your variable names with a letter, not "\$" or "\_".
- Avoid abbreviations, use meaningful names.
- If a variable consists of two or more words, then the second and subsequent words should start with upper case.
- For example, rowHeight.



# Reading input from console

- To read input from the console, JDK provides a friendly class called **java.util.Scanner**.

```
import java.util.Scanner; →Step 1
class A{
public static void main(String[] args) {
Scanner sc = new Scanner(System.in); →Step 2
int i = sc.nextInt(); →Step 3
System.out.println(i);
}
}
```



To read any primitive types, replace **XXX** in **nextXXX()** with the primitive type.

To read a string use **next()**

# Keywords/ Reserved words

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


Note that `null`, `true` and `false` are not keywords

- **final double PI=3.14;**
- Constant Naming Convention
  - All the rules that apply for a variable, applies to constants as well except for the one given below.
  - Convention for constants is that they must all be in UPPER CASE. In cases where you have more than one word making up constant, words must be separated by underscore.

- Integer Literal

- Integer literals are by default **int** (32 bits).
- They can be decimal, octal, hexadecimal or long

- `10, -1, 010, 0xF, 400000000001, 40000000000L`



- Floating Point Literal

- Floating-Point literals are by default **double** (64 bits).
- `32.4, .24, 3E1, 3e-1, 32.4d, 3E1D, 3e-1, 32.1F`

- Character Literal

- They can be characters inside single quote, unicode char or octal character or escape sequence (next slide).
- `'a', '\u0041', '\101', '\n'`

- Boolean Literal : **true, false**

- String Literal : `"Thank You", "\u0041\u0069 " (Ai), "Fly\n"`

Escape Sequence	Character
\n	newline
\t	tab
\b	backspace
\f	form feed
\r	return
\"	" (double quote)
\'	' (single quote)
\\	\ (back slash)
\uDDDD	character from the Unicode character set (DDDD is four hex digits)
\DDD	Character from the Unicode characters set (DDD is octal digits)

- Local declarations
- Class declarations
  - Instance declaration
  - Static declaration



More on this in classes session

# Example: Variable Declarations

```
public class Student {  
    public String name;  
    public int rollno;  
    public void display() {  
        String title="ABC" ;  
        System.out.print(title) ;  
        System.out.println(name) ;  
        title="Roll No.:" ;  
        System.out.print(title) ;  
        System.out.println(rollno) ;  
    }  
}
```

Or String name="ab" ;

Class declarations

Local declaration

Must be initialized otherwise you get a compilation error !

Variable Type	Default Value
byte, short, int, long	0
float, double	0.0
char	'\u0000'
boolean	false
String/ any other Object	null



# Arithmetic Operators

**Unary:** +   -   ++   --


Examples: -5, +5

```
char ch = 'X' ;
```

```
ch++; // (ch = 'Y')
```

**Binary :** +   -   \*   /   %

Examples: `int i = j+5;`

 `int i = -10;`  
`int k = i++;`  
*What will the value of k be?*

 `System.out.println(.4%.2);`  
*What will the code print?*

< > >= <= == !=

Returns **true** or **false**

Example:

```
int i=10;
int j=20;
System.out.println( i>j );
                        // output is false
System.out.println(i==10);
                        // output is true
```

# Exercise

*Write a program to do the following,*

- a) Get two numbers as input from the user through console and swap the values of two numbers without using a temporary variable and display the same. (15 Mins)*
- b) Determine whether the given year is leap year or not (Read the input through console ). (15 Mins)*

~   &   |   ^

Operand 1	Operand 2	&		^	~ Operand 1
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

- They convert the integral data types into their binary form and then perform operations according the table.

# Example : Integer Bitwise Operators

```
public class AndOrNotEx{  
    public static void main(String str[]){  
        byte x=1;  
        byte y=3;  
        System.out.println(x&y); // prints 1  
        System.out.println(x|y); // prints 3  
        System.out.println(x^y); // prints 2  
        System.out.println(~x); // prints -2  
    }  
}
```



*Can you arrive at all of these without executing the code?*

*What will happen if you change byte to double?*

&   |   ^

Operand 1	Operand 2	&		^
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

- ⑦ *What will happen when you compile the following statement?*  
`if (~ (1>2) )`  
`System.out.println("OK") ;`

# Conditional Logical Operators

**&&    ||    !    ? :**

Operand 1	Operand 2	&&		!Operand 1
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Logical operators are binary operators that require **boolean** values as operands.

**&&** and **||** are also called short circuit operators because they are optimized.

# Activity

- Type this code and find out the difference between && and & operators?

```
public class Example{  
    public static void main(String args[]){  
        int i=0;  
        int j=2;  
        boolean b= (i>j) && (j++>i);  
        System.out.println(j);  
        b= (i>j) & (j++>i);  
        System.out.println(j);  
    }  
}
```



Syntax of ?:

```
<variable> = (boolean expression) ? <value to assign  
if true> : <value to assign if false>
```

Example :-

```
int i=10;
```

```
double j=10.1;
```

```
int k=(i>j)?10:20;
```

```
System.out.println(k); // outputs 20
```



*What is the problem with the code below assuming i, j and k are declared as in the example above?*

```
int k=(j=i)?10:20;
```

`+=   -=   *=   /=   %=   &=   |=   ^=`

Example:

```
int a = 10;
```

```
int b=2;
```

```
a+=b; // means a=(int) (a+b) ;
```

```
double d=45.3;
```

```
d/=1.2;
```

# Shift operators

- `>>` the SHIFT RIGHT operator (Arithmetic Shift)
- `<<` the SHIFT LEFT operator
- `>>>` the UNSIGNED SHIFT RIGHT operator (Logical Shift)

```
System.out.println(2<<1); // prints 4
```

```
System.out.println(-1>>2); // prints -  
1
```

```
System.out.println(-1>>>2); // prints  
1073741823
```

# Activity



- Find out what happens if you try to shift more than the number of bits in an int ?

```
System.out.println(2<<32) ;  
System.out.println(2>>32) ;  
System.out.println(2>>>32) ;  
System.out.println(2<<33) ;  
System.out.println(2>>33) ;  
System.out.println(2>>>33) ;
```

## Operators

## Associativity

-----	
[] . () methodCall()	left to right
! ~ ++ -- - + new (cast)	right to left
* / %	left to right
+ -	left to right
>> >>> <<	left to right
< > <= >= instanceof	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	left to right
= += -= *= /= >>= <<= >>>= &= ^=  =	left to right

# Conversions

- A. Widening Conversions (achieved by automatic or implicit conversion)



Overall magnitude not lost

- B. Narrowing Conversions (achieved by casting or explicit conversion)

All primitives are convertible to each other except for `boolean`.

# Automatic or implicit conversion

- The conversion in the direction indicated happens automatically.

**byte**→**short**→**int**→**long**→**float**→**double**  
                    ↙  
                    **char**

- When an arithmetic operation happens between different numeric types, the result of the operation is always of the higher numeric data type.
- When the integer literals are within the range of the specified integer data types, the conversion automatically happens. In case the literal is beyond range, the compiler flags an error.
- If an arithmetic operation happens between any integer type except **long**, the result is an **int**.
- Similarly when arithmetic operation happens between floating points, the result is **double**.
- Compound assignment operator and pre/post increment/ decrement operator does the conversions required.

# Loss of information

- There could be loss of some information during conversion of the following:

a) `int` → `float`

b) `long` → `float`

c) `long` → `double`

Example

```
int i=76543210;
```

```
float f= i; → 7.6543208E7
```



# Assignment conversions- integers

1. `int i=10;`

`int b=10;`

`int k=i+b; // ok`

2. `byte b1=20; // ok`

`short s=10; // ok`

But `byte b=39078; // error`

3. `int b=10;`

`byte b1=b; //error`

But `final int b=10;`

`byte b1=b; //ok`

4. `byte b1=20, b2=30;`

`int i=b1+b2; // ok`

But `byte b=b1+b2; // error`

Same is the case with `short`

5. `byte b=b+1; // error`

But `byte b+=1; // ok`

6. `byte c=10; // ok`

`byte c1=-c; // error`

7. `byte c2=c1+1; // Error : possible  
loss of precision`

But `byte c2=++c1; // ok`

# Assignment conversions- double

```
int f = 32;  
float t=f; //ok  
int k=t; //error
```

① `float f = 32.3;` is compilation error, Why?  
*How can you correct this?*

① `float d=3.4f;`  
*Will the following code compile?*  
`double f= d++;`

1. `char c='A' ;`

`int i=c;`

Assigns unicode value of A to `int`

2. `char c=65; // ok`

65 is within the range of char

3. `char c=-65; // error`

4. `int ii=65;`

`char c=ii; //error`



*What change can you make in `int`'s declaration so that the code compiles?*

5. `char c='A' ;`

`short s=c ; // error`



*Note that even though `short` and `char` are 16 bits, an error is generated. Why should compiler not allow this?*

- Any conversion between primitives (excluding **boolean**) that is not possible implicitly can be done explicitly.
- Conversions like (a) **double to long**, (b) **char to byte** etc.
- This is done through what is called casting.

Example:

```
int k=10;  
char b=k; // error
```


Casting makes the error disappear:

```
char b=(char)k;
```



```
byte b=(byte)128;
```

*What will be the value when you print b? Compute and arrive at this by yourself.*

- Conditional Statements
  - **if** statement
  - **switch** statement
- Loops
  - **for** statement
  - enhanced **for** statement  *Coming up later*
  - **while** statement
  - **do-while** statement
- Loops can have **break** or **continue**.
- All of these statements except (enhanced for statement) are same as that in C (in terms of the syntax and way they work).
- But note that for Java conditions must always evaluate to **boolean** value.
- The **switch** expression should be integer value (not long) or char and case expression must evaluate to a constant/final value

# Common Errors

② Can you figure out why the following statements are erroneous?

- `if(1) System.out.println("OK");`
- `while(1.1) { // some statements;  
}`
- `switch(myString) {  
case "Sun" :  
case "Sat": System.out.println("Holiday");  
break;  
default: System.out.println("Holiday");  
}`
- `for(int i=1; (i<5); i++)  
{ System.out.println(i+j); }  
i=12;`

# for loop omitting options

- `int j=10;`  
  `for(; j>0 ;j--) {`  
    `System.out.println(j);}`
- `int j=10;`  
  `for(;j>0;) {j--; System.out.println(j);}`
- `int j=10;`  
  `for(;;j--) {`  
    `if(j<0) break;`  
    `System.out.println(j);}`
- `for(;;) {...}`

What is the purpose of the statement above?



# Exercise

*Tax slabs for general*

*0 to 1,80,000 No tax*

*1,80,001 to 5,00,000 10%*

*5,00,001 to 8,00,000 20%*

*Above 8,00,000 30%*

*Income tax slabs 2011-2012 for Women*

*0 to 1,90,000 No tax*

*1,90,001 to 5,00,000 10%*

*5,00,001 to 8,00,000 20%*

*Above 8,00,000 30%*

*Write if statements to achieve this.*

*Make sure that you indent the code well so that it is readable.*

*(20 mins)*

# Exercise

*A shopkeeper sells three products whose retail prices are as follows:*

Product No	Product Code	Retail Price
1	A	22.50
2	B	44.50
3	C	9.98

*Write an application that reads a series of pairs of numbers as follows:*

- a) Product number or code (Code is not case sensitive)*
- b) Quantity sold*

*The application should use a switch statement to determine the retail price for each product. It should calculate and display the total retail value of all products sold. (20 Mins)*

# Exercise

*Consider user has N eggs. Then display the no of eggs in gross (144 eggs make one gross) and no of eggs in dozen (12 eggs make one dozen) and the no of eggs that is left out remaining. The total no of eggs can be input through console. The program should display how many gross, how many dozen, and how many left over eggs the user has. (20 Mins)*

*For example, if the input is 1342 eggs, then the program should respond with*

➤ *Your number of eggs is 9 gross, 3 dozen, and 10*

*(20 mins)*

# Activity: do-while loop

```
public class Test1 {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int f1=0,f2=1;  
        System.out.println(f1);  
        System.out.println(1 ?  
        do {  
            f2=f1+f2;  
            System.out.println(f2);  
        }while(f2<10) ; }  
    }
```

- What is the program trying to do?
- There is a problem with this code. Can you correct this code by just inserting a line?
- What will the code display after correction?

# Activity

Select the statement (s) ideal for the scenarios.



## Scenarios

- User has to enter his/her information. After he/she finishes entering information, a prompt (Y/N) whether he/she wants to change some details is displayed. If Y is entered he/she can re-enter the details again.
- Sum of first 50 odd numbers
- Easiest way to write infinite loop.
- Any of the 5 different types of Graph is to be displayed based on the user's choice.

## Statements

- **if** statement
- **switch** statement
- **while** statement
- **do-while** statement
- **for** statement

# Tell me why?

- `while(false) { statement;}` → unreachable code error

But

`if(false) { statement;}` → unreachable code error

Why?

The reason for this is that Java does not have conditional compilation statement. The java language creators decided to leave the if statement with false condition to enable programmers to use conditional compilation kind of statements in java.

Example:

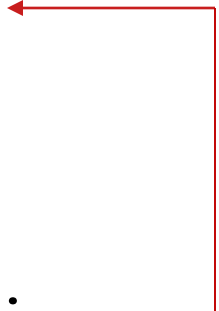
```
final boolean DEBUG = false;  
if (DEBUG) { y=10; }
```

# Labeled continue/break statements

- The normal **break** (without labels) is used to **break** out of the loop (used in switch statement also) and **continue** is used to skip the rest of the statements in the loop and start with a new iteration.
- In cases where there are multiple loops, when we **break** out or **continue** the loop, the immediate loop breaks or continues.
- **break** and **continue** statement can also be used with labels to indicate from which outer loops should it **break** out or **continue**.
- The outer loop where the **break** or **continue** must happen is labeled.
- Note that labeled break/ continue will work only if the labels are provided for the loops in which they are enclosed.

Example:

```
first: for(int i=0;i<2;i++)  
    for(int j=1;j>0;j--)  
        if(i!=j)  
            continue first;  
    else  
        System.out.println(i+j) ;  
  
//prints 2
```



When i=0 and j=1



# Exercise

*Write a program to display whether the given number is palindrome*

*or not. Also, check whether the number is a prime number. If it is a*

*prime number, display the number along with the alphabet*

*'p' appended to it.*

*(30 Mins)*

# Summary

- byte, short, int, long, float, double, char, boolean are Primitive data types.
- `java.util.Scanner` is used to read input from the console.
- Arithmetic Operators, Relational Operators, Integer Bitwise Operators, Logical Operators, Conditional Logical Operators, Ternary operator, Compound Operators, Shift operators are various operators.
- Any conversion between primitives (excluding boolean) that is not possible implicitly can be done explicitly by casting.
- Control Statements are the statements that enable us to order the sequence of flow of a program. if statement, switch statement are the Conditional Statements and while statement, for statement, do-while statement, enhanced for statement are Loops.
- Loops can have break or continue.