FUNDAMENTALS OF JAVA

OBJECTIVE

- Data types
- Variables
- Type conversion
- Operators
- Decision constructs
- Loops
- Arrays
- Comments

DATA TYPES IN JAVA

- Primitive Data types
- Reference/Composite Data types

PRIMITIVE DATA TYPES

Group	Data Type	#bits	Range	Default value
Integer	byte	8	-128 to 127	0
	short	16	-32768 to 32767	0
	int	32	-2,147,483,648 to 2,147,483,647	0
	long	64	-9,223,372,036,854, 775,808 to 9,223,372, 036, 854,775,807	0

PRIMITIVE DATA TYPES

Group	Data Type	#bits	Range	Default value
Floating point	float	32	3.4e-038 to 3.4e+038	0.0
	double	64	1.7e-308 to 1.7e+308	0.0
Boolean	boolean	1	true or false	false
Character	char	16	A single character	Unicode character

VARIABLE NAMING CONVENTION

- A storage location for a value is called a variable
- Rules for naming variables are:
  - the name of a variable should be meaningful and short
  - variable names should not have any embedded space or symbols like - ? ! @ # % & ( ) { } [ ] . , ; : ‘ “ / and \
  - variable names must begin with a letter, a dollar symbol ($) or an underscore (_) followed by numbers or characters
  - keywords cannot be used for naming variables

For example:

- employeeName
- address1

INITIALIZING VARIABLES

- Variables cannot be used without initializing
- Java doesnot support Garbage Values
- Values can be assigned to variables in two ways:
  - at the time of declaration

```
int salary = 5000;
```

  - in the program after the declaration and before use

```
int salary;
salary = 5000;
```

TYPE CONVERSION

- Refers to the process of converting one data type into another
- Implicit Conversion

  byte -> short -> int -> long -> float -> double

  char -> int -> long -> float -> double

- Any other conversion needs to be explicitilty typecasted

EXPLICIT TYPECASTING

- Explicit conversion: Casting

  (<type>) <expression>

- Example

```
double f;
    int i = 3;

f=(double) i;
```

TYPE CONVERSION

- Type conversion may result in loss of data
- A boolean variable cannot be typecasted

OPERATORS

- Arithmetic operators

  `+ , - , * , / , %`

- Assignment Operators

  `= , += , -= , *= , /= , %=`

- Unary Operators

  `++ , --`

- Comparison Operators

  `== , != , > , < , >= , <=`

- Logical Operators

  `&& , ||`

UNARY OPERATORS

Operator	Implications
x++	Post-increment : add 1 to the value. x = 1; y = x++;
x--	Post-decrement : subtract 1 from the value. x = 1; y = x--;
++x	Pre-increment : add 1 to the value. x = 1; y = ++x;
--x	Pre-decrement : subtract 1 from the value. x = 1; y = --x;

DECISION CONSTRUCTS

- If - else
- Switch - case

THE IF CONSTRUCT

- makes a decision on the basis of a logical expression
- compares the data and decides the action on the basis of the comparison
- Syntax
  - If (Boolean_expr) { statements }
  - If (Boolean_expr) { statements } else { statements }
- Example
  - if (age > 18) { System.out.println(“Eligible to Vote”); }
  - if (age > 18) { System.out.println(“Eligible to Vote”); }
    else {System.out.println(“Not Eligible to Vote”); }

IF - STATEMENT

NESTED – IF

```
If (Boolean_expr) {
        if (Boolean_expr){
                statements;
        }
}
```

IF – ELSE IF - ELSE

```
If (Boolean_expr){
        statements;
} else if (Boolean_expr){
        statements;
}
else {  statements;
}
```

SWITCH CONSTRUCT

- Multiple values are checked with the same variable, when a match is found and the statements associated with that `case` constant are executed
- Syntax:

```
switch ( <expr.> )  {
        case <const. expr. 1> : <statement 1> break;
        case < const. expr. 2> : < statement 2> break;
                :
        case < const. expr. n> : < statement n> break;
        default : < statement>
}
```

LOOPING CONSTRUCTS

- while
- do – while
- for
- for .. each

WHILE LOOP

- causes a section of a program to repeat a certain number of times based on a condition specified
- ends when the condition becomes false
- variable that is checked in the boolean expression is called the *loop control variable*

Syntax

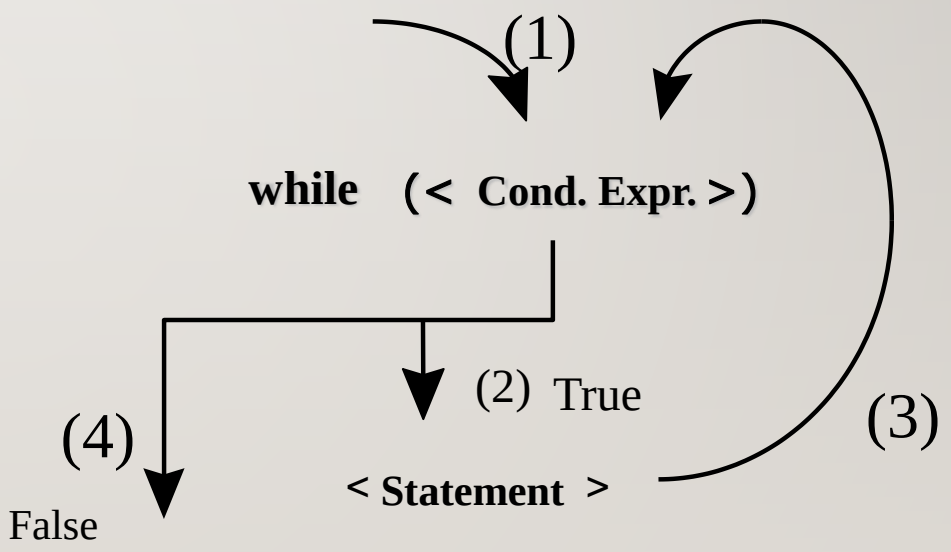
```
while (boolean_expr)
{
    statements;
}
```

WHILE LOOP

EXAMPLE

```
i = 1;  s = 0;
while (i <= N) {   // summation from 1 to N
      s += i;  ++i;
}
```

ORDER OF EXECUTION



FOR LOOP

- is like a `while` loop
- is used when the number of iterations are known in advance
- contains three parts with the keyword `for`:
  - initialization expression
  - test expression
  - increment/decrement expression
- Syntax

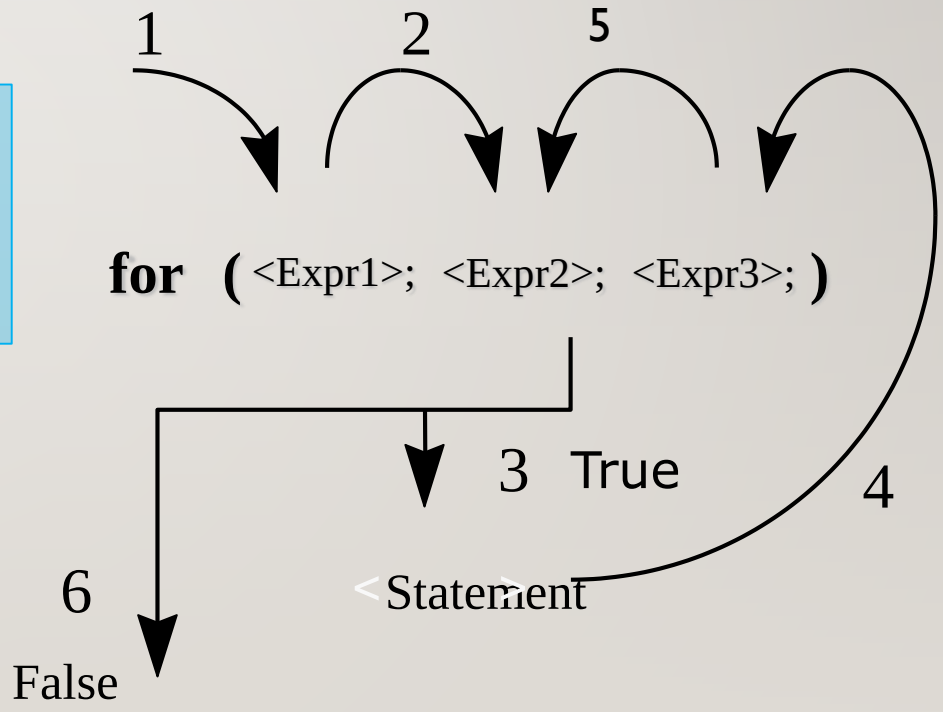
```
for ( initialization_expr ; test_expr ; increment/decrement_expr){
        //statements
}
```

FOR LOOP

EXAMPLE

```
s = 0;
  for (i=1; i<=N; ++i)    // sum from 1 to N : i increment
        s += i;
```

ORDER OF EXECUTION



DO…WHILE LOOP

- After executing the repeating statements, then check the conditional expression
- Form of do-while statement

```
do
    <statement>
while (<conditional expression>);
```

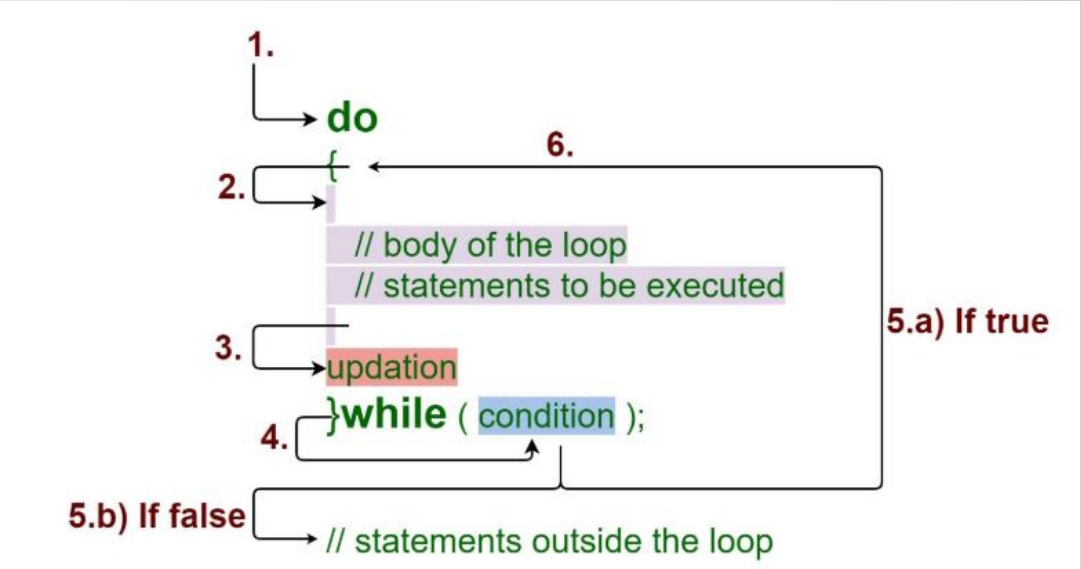
- Even if the conditional statement is false, the statements will be executed atleast once.

DO… WHILE LOOP

EXAMPLE

```
i = 1;  s = 0;
do {
    s += i;  ++i; // summation from 1 to N
 } while (i <= N);
```

ORDER OF EXECUTION



BRANCH STATEMENT - BREAK

- To move control to the out of the block
- Form of break statement

```
int i = 1;
while (true) {
        if (i == 3)
                break;
        System.out.println("i have visited " + i + " time/s");
        ++i;
}
```

BRANCH STATEMENT - CONTINUE

- To move control to the start of next repetition
- Form of continue statement

  continue [Label] ;

- When used in for statement

```
for (i=0; i<=5; ++i) {
        if (i % 2 == 0)
                continue;
        System.out.println("i have visited " + i + " time/s");
}
```

CONTINUE IN WHILE LOOP

```
i = 0;
while (i <= 5)  {
        ++i;
        if (i % 2) == 0)
                continue;
        System.out.println("This is a odd iteration - " + i);
}
```

LABELLING OF LOOP

```
labelName :
    Rep. St. 1  {
        Rep. St. 2  {
            // . . .
            break;
            // . . .
            break   labelName;
        }
        // . . .
    }
```

```
labelName:
    Rep. St. 1 {
        Rep. St. 2 {
            // ...
            continue;
            // ...
            continue  labelName;
        }
    }
```

ARRAYS

COMMENTS

- Java supports three styles of comments
  - Multiple-line comments

```
/* multiple line
comments */
```

  - Single-line comment entries

```
// single line comment
```

  - The `javadoc` comments

```
/** this is a
javadoc comment */
```