Basic Elements Of Java

Data types, operators and statements

Primitive data types

- 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`**

Ranges of Primitive data types

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)		

Why 16 bits for char?

Unicode Character

- 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 !
- For example, Character 'A' represented in unicode as '\u0041' – which is the number 65 in base 10.
- The Unicode Standard encodes characters in the rang U+0000..U+10FFFF

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

Why Unicode?

- Limitation of ASCII/Extended ASCII
  - Limited number of bits therefore does not have capability to represent multi-lingual characters

- Unicode
  - Has 16 bits, therefore has ability to represent multi-lingual characters
  - Industry standard
  - Internationalization

What is UTF 8 and UTF 16 then?

More on Unicode

Definition from http://www.unicode.org/glossary/#unicode_encoding_form

- *Unicode transformation format* is character encoding form that assigns each Unicode scalar value to a unique code unit sequence. The Unicode Standard defines three Unicode encoding forms:
  - UTF 8
  - UTF 16
  - UTF 32
- Java 6 char supports UTF 16.
- Example: Printing char in Hindi
  - `System.out.println("\u0905”);`

Variables

- Variable name must begin with
  - a letter (A-Z, a-z, or any other language letters supported by UFT 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.

Keywords/ Reserved words

abstract	continue	for	new	switch
assert	default	goto	package	synchronize d
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

More on Keywords

- `goto` and `const` are unused keywords. They are reserved for future use.
- `null`, `true` and `false` are not keywords. However, they cannot be used as identifiers.
- Java classes are not keywords.

Constants

- Value of a constant cannot be changed once a value is assigned.
- **`final`** keyword in front of the variable denotes constant.
- **`final double PI=3.14;`**
- Constant Naming Convention
  - All rules that apply for a variable also applies to constants except the one below
  - Constants must be all in UPPER CASE. In cases where you have more than one word making up constant, words must be separated by underscore.

Literals

- A literal is the value assigned to a variable. For example 7 is a literal.
- The literals available in java:
  - Integer Literal
  - Floating Point Literal
  - Character Literal
  - Boolean Literal
  - String Literal

Integer Literals

- Integer literals are by default **`int`** (32 bits).
- There are four ways to represent integer numbers in the Java language:
  1. decimal (base 10): **`10, -1;`**
  2. octal (base 8) : **`07,010;`**
  3. hexadecimal (base 16): **`0xf,0xF,0x11`**
  4. long literals: **`4000000000 or x = 4000000001`**

- **`int i=0xFF;`**
- **`final long k=35L;`**

Floating-Point Literals

- Floating-Point literals are by default double (64 bits).
- They can be represented as

1. **`32.4, .24`**
2. **`3E1, 3e-1`**
3. **`32.4D , 3E1D, 3e-1D 32.4d , 3E1d`**, A suffix of D or d also could be added to improve readability for double literals.
4. **`32.1F , 32.1f`**

- **`final double PI=3.14;`**
- **`float f=2.1e10f;`**

Character Literals

1. Character Literal:

   `'a`, or '@′`

2. Unicode Character Literal:

   `'\u0041'; // The letter A`

3. Escape Sequences Literals:

   `‘\n′ (enter key), ‘\t′ (tab key)`

4. Octal character literals:

   `'\101′ (A)`

- `char c=‘z′;`
- `final char m=‘\101′;`

Escape Sequence

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)

Boolean literals

- `true`
- `false`
- `boolean b=true;`
- `final boolean b=false;`

- Unlike C/C++, **`boolean`** literals are not mapped to 0 or non-zero number.
- The **`boolean`** value is type-safe. They are not convertible to anything.

~~**`boolean b1=1;`**~~

~~**`boolean b2=10;`**~~

~~**`boolean b3=-1;`**~~

All the above statements will give compilation error

String Literals

- `“Bill Gates”`
- `“Thank You”`
- `"\u0041\u0069 “ (Ai)`

- String is a type defined by java standard library.
- `String s= “Welcome”;`
- `final String=“Fly\n”;`

Variable Declarations

- Local declarations
  - Declarations made inside a method
  - A local variable **must always be initialized** to a value before it can be used in calculations or display.
  - If a variable is used without initialization, compiler will flag an error.

- Class declarations
  - Declarations made outside a method
  - A class variable is **automatically assigned a default value** if it is not initialized.
  - There are two types of class declarations
    - Instance declaration
    - Static declaration

More on this in classes session

Example

```
public class Student {
public String name;
public int rollno;
public void display(){
String title ="Name:"    ;
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 compilation error !

Default Values

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

Arithmetic Unary Operators

+ - ++ --

Examples: **-5, +5**

Pre-increment: **y=++x; //(x=x+1; and y=x;)**

Post-increment: **y=x++; //(y=x; and x=x+1;)**

Examples

```
int k=10;
k++; //value becomes 11
char ch='X';
ch++;  // (ch = 'Y')
```

```
int i=-10;
int k=i++;
```

What will be the value of k?

```
float d=+3.4f;
```

Will the following code compile?

```
double f= d++;
```

Arithmetic Binary Operator

```
+    -    *     /     %
```

Example:

```
int a = 5;
int b = 2;
int c = a + b;
System.out.println(5%2); // output is 1
```

remainder operator

Relational Operators

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

Example:

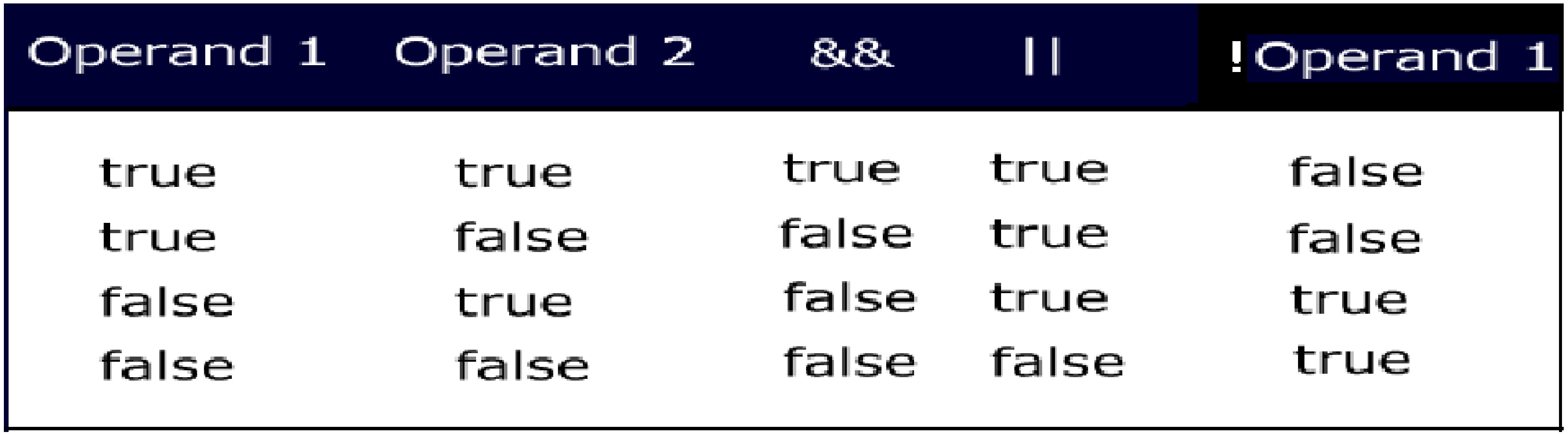
```
int i=10;
int j=20;
System.out.println( i>j );
                              // output is false
System.out.println(i==10);
                              // output is true
```

What will happen when you do this?

```
System.out.println(i=10);
```

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.
- Does not work if the operands are **`int`**

Short circuit operators

- **`&&`** and **`||`** are also called ***short circuit operators*** because they are optimized.
- **`&&`** checks if the first condition is false. If it is so then it doesn't evaluate the second condition.
- **`||`** checks if the first condition is true. If it is so then it doesn't evaluate the second condition.

Example :-

```
int i=0;
int j=10;
System.out.println( i>j || j>i ); // prints true
```

Example for **!** :-

```
int i=0;
int j=10;
System.out.println(! (i>j)); // prints true
```

Guess what the result of this code is?

```
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);
 }
}
```

Ternary operator

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;
```

Integer Bitwise Operators

~ & | ^

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; // 0000 0001
byte y=3; // 0000 0011
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?
(for ~x explanation is in the next slide)

What will happen if you change byte to double?

More bitwise

- **`byte x=1;`** `// 0000 0001`

  `System.out.println(~x); // prints -2`

Explanation:

`~x is` `1111 1110.`

This is a –ve number. To get the value of –ve number, find its 2's compliment.

2's compliment is 1's compliment +1

1's compliment of `1111 1110` **`is`** `0000 0001`

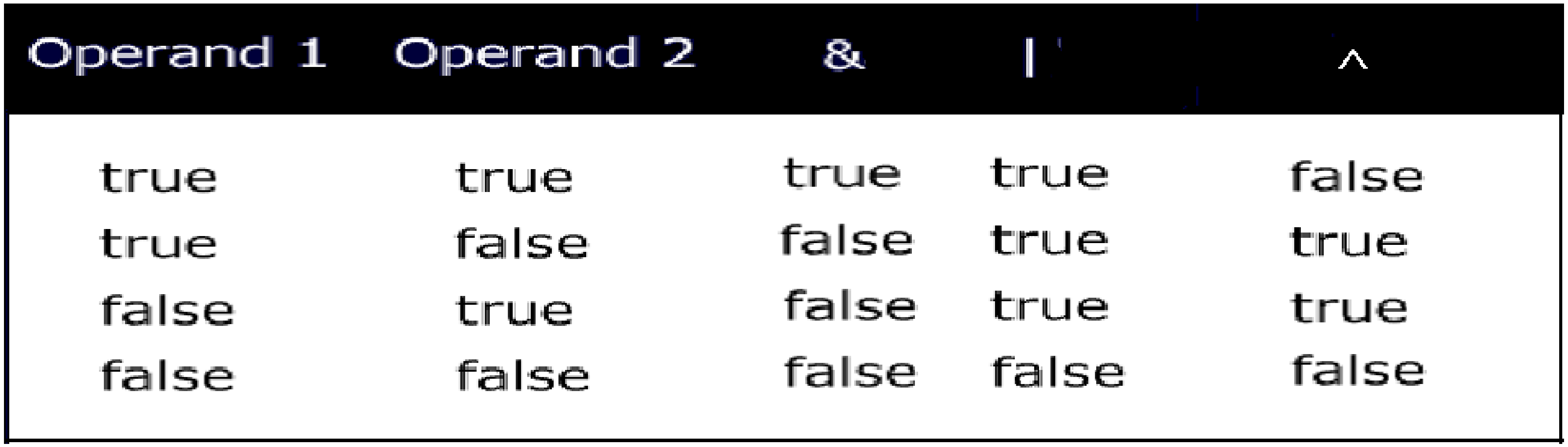
```
+1                                   1
                            ------------
                             0000 0010 → 2
```

Since the number is –ve the result is `-2`

What is the value of ~ -2?

Logical Operators

& | ^



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

What will happen when you compile the following statement:

```
if(~(1>2))
System.out.println(“OK");
```

Example: Logical 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); //prints 3
 }
}
```

Compare this with short circuit operator &&.

Compound Operators

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

Example:

```
int a = 10;
int b=2;
a+=b; //  means a=(int)(a+b);

double d=45.3;
d/=1.2;
```

Shift operators

- In case of right shift there is always a debate whether the left bits that get vacant during the shift must retain the sign of the original number or not.Some people believe that the sign must be retained → Arithmetic Shift.Others believe it must not be retained → Logical Shift
- That is the reason why Java came up with two shift operators >> for Arithmetic Shift and >>> for Logical Shift. (*Recall the non-portability feature of Java)*
- **>>** the SHIFT RIGHT operator
- **<<** the SHIFT LEFT operator
- **>>>** the UNSIGNED SHIFT RIGHT operator

```
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);`

Precedence

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)

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

$\downarrow$

Overall magnitude not lost

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

Automatic or implicit conversion

- The conversion in the direction indicated happens automatically.

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float} \arrow[r] & \texttt{double} \\
 & \texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

- When an arithmetic operation happens between different numeric types, the result of the operation is always of the higher numeric data type.
- 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`**.

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`

- Integer literals are **`int`**
- When the integer literals are within the range of the specifies integer data types, the conversion automatically happen. In case the literal is beyond range then compiler flags an error.
- This again is one if the strict type-checking that compiler does.

Assignment conversions- double

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

Guess what happens if you compile this statement?

```
float f = 32.3;
```

- Floating point literals are double.
- So double value cannot be assigned to float.

What should be done to correct this?

Arithmetic operators conversions

1. **`byte b1=20, b2=30;`**

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

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

   Same is the case with **`short`**

- Since arithmetic operations between integer types results in int (except when it involves long), b1+b2 yields error (possible loss of precision)

2. **`byte b=b+1; // error`**

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

   because compiler converts this as **`byte=(byte)(b+1);`**

3. **`byte c=10; // ok`**

   `byte c1=-c; // error`

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

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

- Compound assignment operator and pre/post increment/ decrement operator does the conversions required.

char conversions

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
```

Though short and char are 16 bits! Why should compiler not allow this?

Explicit conversion or casting

- 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.

Flow control statements

- Conditional Statements
  - **`if`** statement
  - **`switch`** statement
- Loops
  - **`for`** statement
  - enhanced **`for`** statement $\longrightarrow$ *Coming up later*
  - **`while`** statement
  - **`do-while`** statement
- Loops can have **`break or continue.`**
- General Syntax Notation
  - Anything inside square brackets **`[… ]`** means it is optional
  - ***`statement(s)`*** could be either a single statement or a block of statement enclosed in with curly brackets "**`{…}`**"

`if` statement

- Syntax:

```
if (condition) statement(s)
[else statement(s)] ;
```

- **`if`** statement are same as that in C except that the condition must always evaluate to a **`boolean`** value.

`if` example

- `if(x==y) x++;`
- `if (x>y) y++;`

  `else x++;`
- `if(true) {`

  `y++;`

  `System.out.println(“OK”);`

  `}`

Will this work?

```
if(1) System.out.println(“OK”);
```

Activity : `if` statement

Tax slabs for general	Income tax slabs 2011-2012 for Women
0 to 1,80,000 No tax	0 to 1,90,000 No tax
1,80,001 to 5,00,000 10%	1,90,001 to 5,00,000 10%
5,00,001 to 8,00,000 20%	5,00,001 to 8,00,000 20%
Above 8,00,000 30%	Above 8,00,000 30%

Write **`if`** statements to achieve this.

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

`switch` Statement

- **`Syntax:`**

```
switch (expression){
   [case expression: statement(s)]
  …
  [default: statement(s)]
}
```

So when do you use switch statement instead of if statement?

- **`switch`** statement is also very similar to the one in C
- **`switch`** expression must be either integer value (not long) or char. In Java SE and later, **`String`** can also be used in **`switch`** statement's expression
- **`case`** expression must evaluate to a constant/final value.
- **`break`** statement after every set of case statements will prevent fall-through.

switch Example 1

```
switch (week) {
  case 1: System.out.println("Sunday");
          break;
  case 2: System.out.println("Monday");
          break;
  case 3: System.out.println("Tuesday");
          break;
  case 4: System.out.println("Wednesday");
          break;
  case 5: System.out.println("Thrusday");
          break;
  case 6: System.out.println("Friday");
          break;
  case 7: System.out.println("Saturday");
          break;
}
```

`switch` Example 2

```
switch(x){
case 'm':   ↓  Fall through
case 'M':
       System.out.print("Male");
       break;
case 'f':
case 'F':
       System.out.print("Female");
       break;
default:
      System.out.print("Male");
}
```

`for` Loop

- Syntax

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

- `for` statement (like in C) is used to iterate through a set of statements over a range of values specified and computed by the `for` loop itself.
- The ***`initialization`*** expression is
  - used to initialize variable(s)
    - more than one variable initialization is separated by commas)
    - can also include initialization with declaration
  - executed only once when the loop begins.

- When the ***`condition`*** expression
  - Must evaluate to a **`boolean`** value
  - Loop iterates till the condition is **`true`**
  - Only one condition can be specified; multiple conditions can be combined using logical operators
  - is evaluated at the beginning of each loop
- The ***`iteration`*** expression
  - is usually an increment or decrement expression of the variable initialized in the initialization expression
  - is evaluated at the beginning of each loop

`for` loop- Example 1

```
public class ________
{
/* This program checks if a ______ is a  _______. */
public static void main(String str[]){
int num = 5;
boolean flag=true;
for (int i=2;i<=num/2;i++){
 if(num%i==0){
   flag=false;
   System.out.println("_______");
   return;
   }
}
if (flag) System.out.println("______");
}}
```

Notice the way i is initialized in the for loop.
i is not available outside the for loop.

Can you guess what this code does and fill in the blanks?

for loop multiple declarations- Example 2

```
public class Test{
public static void main(String[] args)
{
for(int i=1, j=5 ; (i<5) && (j>0) ; i++ , j--) {
System.out.println(i+j);
}
~~i=12;~~ /* gives error since i and j are available
   only inside the for loop */
}
}
```

`for` loop omitting options - Example 3

-
  ```
  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?

`while` and `do-while` loop

- **`while(`*`condition`*`)`** ***`statement(s);`***
- **`do`** ***`statement(s)`*** **`while(`*`condition`*`);`**

- Like in C, **`while`** and **`do-while`** statement is used to iterate through a set of statements till the condition remains true.
- **`while`** evaluates the condition before at the beginning of each iteration whereas **`do-while`** evaluates condition only at the end of each iteration.
- Therefore, **`do-while`** guarantees that the loop statements are executed at least once.

Condition expression must result in ____________

Can you think of a situation where you would prefer **`do-while`** *instead of* **`while`***?*

`while` loop- Example

```
int j=10002;
while(true)
{
if(j/13 ==0) break;
else
j++;
}
```

What will happen if `true` is *changed* to `false`

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(f2);

do {

f2=f1+f2;

System.out.println(f2);

}while(f2<10);}

}
```

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

Test your understanding

Select the statement (s) are ideal for the scenarios.

Scenarios

- User has to enter his/her information. After he/she finishes entering information, a choice whether he/she wants to change some the details is prompted (Y/N). IF Y is entered he 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?

Reason is 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; }
```

`break` and `continue`

- `break` (as in C) is used to break out of the loop. It is also used with switch statement (which we have already seen).
- `continue` (as in C) is used to skip the rest of the statements in the loop and start with a new iteration

break and continue- Example

```
public class Prime
{
public static void main(String str[]){
int num = 5;
boolean flag=true;
for (int i=2;i<=num/2;i++){
 if(num%i!=0) continue;
  flag=false;
  System.out.println("Not Prime");
  break;
}
if (flag) System.out.println("Prime");
}}
```

Labeled `continue/break` statements

- 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.

Labeled continue/break statements

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