

Object Oriented Programming in Java

Er.Sital Prasad Mandal

BCA- 2nd sem

Mechi Campus

Bhadrapur, Jhapa, Nepal

(Email : info.sitalmandal@gmail.com)

<https://ctaljava.blogspot.com/>



Text Book

1. Deitel & Dietel. -Java: How to-program-. 9th Edition. TearsorrEducation. 2011, ISBN: 9780273759168
2. Herbert Schildt. "Java: The CoriviaeReferi4.ic e 61 Seventh Edition. McGraw -Hill 2006, ISBN; 0072263857

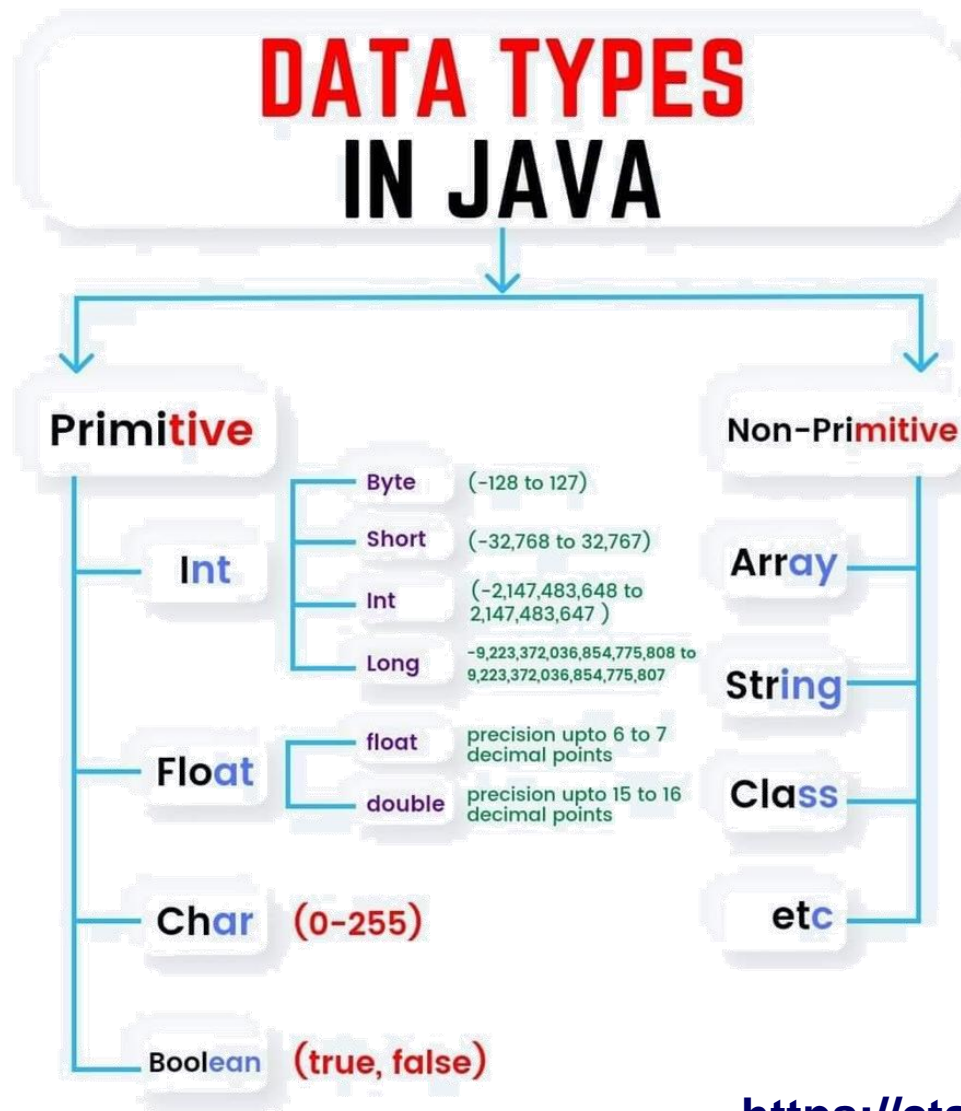
2. Tokens, Expressions and Control Structures

2. Tokens, Expressions and Control Structures

1. Primitive Data Types: Integers, Floating-Point types, Characters, Booleans;
2. User Defined Data Types, Declarations, Constants, Identifiers, Literals, Type Conversion and Casting
3. Variables: Variable Definition and Assignment, Default Variable Initializations
4. Command-Line Arguments
5. Arrays of Primitive Data Types
6. Comment Syntax
7. Garbage Collection
8. Expressions
9. Using Operators: Arithmetic, Bitwise, Relational, Logical, Assignment, Conditional, Shift, Ternary, Auto-increment and Auto-decrement
10. Using Control Statements (Branching: if, switch Looping: while, do-while, for; Jumping statements: break, continue and return)

2. Tokens, Expressions and Control Structures

Data Types

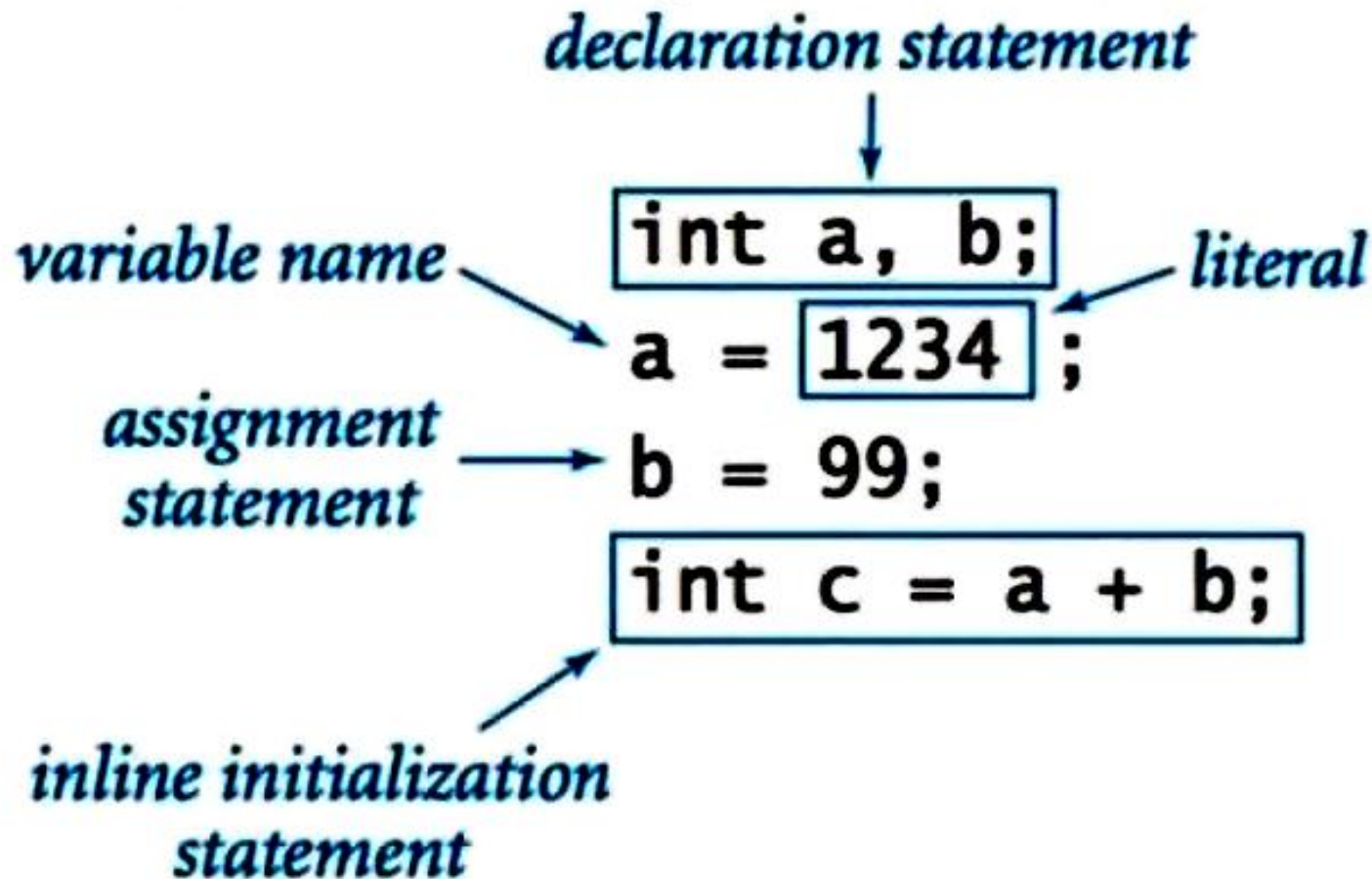


2. Tokens, Expressions and Control Structures

Built-in Data types

<i>type</i>	<i>set of values</i>	<i>common operators</i>	<i>sample literal values</i>
int	integers	+ - * / %	99 12 2147483647
double	floating-point numbers	+ - * /	3.14 2.5 6.022e23
boolean	boolean values	&& !	true false
char	characters		'A' '1' '%' '\n'
String	sequences of characters	+	"AB" "Hello" "2.5"

Declaration & Assignment



2. Tokens, Expressions and Control Structures

Integers

Integers.

<i>values</i>	integers between -2^{31} and $+2^{31}-1$					
<i>typical literals</i>	1234 99 0 1000000					
<i>operations</i>	<i>sign</i>	<i>add</i>	<i>subtract</i>	<i>multiply</i>	<i>divide</i>	<i>remainder</i>
<i>operators</i>	+ -	+	-	*	/	%



Integers

<i>expression</i>	<i>value</i>	<i>comment</i>
99	99	<i>integer literal</i>
+99	99	<i>positive sign</i>
-99	-99	<i>negative sign</i>
5 + 3	8	<i>addition</i>
5 - 3	2	<i>subtraction</i>
5 * 3	15	<i>multiplication</i>
5 / 3	1	<i>no fractional part</i>
5 % 3	2	<i>remainder</i>
1 / 0		<i>run-time error</i>
3 * 5 - 2	13	<i>* has precedence</i>
3 + 5 / 2	5	<i>/ has precedence</i>
3 - 5 - 2	-4	<i>left associative</i>
(3 - 5) - 2	-4	<i>better style</i>
3 - (5 - 2)	0	<i>unambiguous</i>

2. Tokens, Expressions and Control Structures

Primitive Data Types

- The Predefined data types provided by Java are known as it.
- It also known as in-built data types.
- The eight primitives defined in Java are int, byte, short, long, float, double, boolean and char.
- These aren't considered objects and represent raw values.
- They're stored directly on the stack.

2. Tokens, Expressions and Control Structures

Primitive Data Types

Type	Size (bits)	Example
<i>byte</i>	8	<i>byte b = 100;</i>
<i>short</i>	16	<i>short s = 30_000;</i>
<i>int</i>	32	<i>int i = 100_000_000;</i>
<i>long</i>	64	<i>long l = 100_000_000_000_000;</i>
<i>float</i>	32	<i>float f = 1.456f;</i>
<i>double</i>	64	<i>double f = 1.456789012345678;</i>
<i>char</i>	16	<i>char c = 'c';</i>
<i>boolean</i>	1	<i>boolean b = true;</i>

2. Tokens, Expressions and Control Structures

Primitive Data Types



2.1. *int*

- The first primitive data type we're going to cover is *int*. Also known as an integer, *int* type holds a wide range of non-fractional number values.
- Specifically, **Java stores it using 32 bits of memory**. In other words, it can represent values from -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31}-1$).
- In Java 8, it's possible to store an unsigned integer value up to 4,294,967,295 ($2^{32}-1$).
- We can simply declare an *int*:

```
int x = 424_242;  
int y;
```

The default value of an *int* declared without an assignment is 0.

If the variable is defined in a method, we must assign a value before we can use it.

We can perform all standard arithmetic operations on *ints*. Just be aware that **decimal values will be chopped off** when performing these on integers.

2. Tokens, Expressions and Control Structures



Primitive Data Types

2.2. *byte*

byte is a primitive data type similar to *int*, except it **only takes up 8 bits of memory**. This is why we call it a byte. Because the memory size is so small, *byte* can only hold the values from -128 (-2^7) to 127 ($2^7 - 1$).

```
byte b = 100;  
byte empty;
```

2.3. *short*

At 16 bits of memory, it's half the size of *int* and twice the size of *byte*.

```
short s = 20_020;  
short s;
```

2.4. *long*

Our last primitive data type related to integers is *long*.

long is the big brother of *int*. **It's stored in 64 bits of memory**, so it can hold a significantly larger set of possible values.

```
long l = 1_234_567_890;  
long l;
```

2. Tokens, Expressions and Control Structures

Primitive Data Types



2.5. *float*

This type is stored in 32 bits of memory just like *int*.

```
float f = 3.145f;  
float f;
```

And the default value is 0.0 instead of 0.

2.6. *double*

It's stored in 64 bits of memory.

```
double d = 3.13457599923384753929348D;  
double d;
```

2.7. *boolean*

The simplest primitive data type is *boolean*. It can contain only two values: *true* or *false*. **It stores its value in a single bit.**

However, for convenience, Java pads the value and stores it in a single byte.

```
boolean b = true;  
boolean b;
```

2. Tokens, Expressions and Control Structures

Primitive Data Types



2.8. *char*

The final primitive data type to look at is *char*.

Also called a character, *char* is a 16-bit integer representing a Unicode-encoded character. Its range is from 0 to 65,535. In Unicode, this represents `'\u0000'` to `'\uffff'`.

```
char c = 'a';
```

```
char c = 65;
```

```
char c;
```

2.9. Overflow

We run into a situation called **overflow**.

```
int i = Integer.MAX_VALUE;
```

```
int j = i + 1; // j will roll over to -2_147_483_648
```

```
double d = Double.MAX_VALUE;
```

```
double o = d + 1; // o will be Infinity
```

Underflow is the same issue except it involves storing a value smaller than the minimum value. When the numbers underflow, they return **0.0**.

2. Tokens, Expressions and Control Structures

Primitive Data Types



2.10. Autoboxing

Each primitive data type also has a full Java class implementation that can wrap it. For instance, the *Integer* class can wrap an *int*. There is sometimes a need to convert from the primitive type to its object wrapper (e.g., using them with generics).

Luckily, Java can perform this conversion for us automatically, a process called *Autoboxing*:

```
Character c = 'c';  
Integer i = 1;
```

2. Tokens, Expressions and Control Structures



User Defined Data Types

- User defined data types are those which are developed by programmers by making use of appropriate features of the language.
- User defined data types related variables allows us to store multiple values either of same type or different type or both.
- This is a data type whose variable can hold more than one value of dissimilar type, in java it is achieved using class concept.

Note: In java both derived and user defined data type combined name as reference data type.

- In C language, user defined data types can be developed by using struct, union, etc.
- In java programming user defined datatype can be developed by using the features of classes and interfaces.

2. Tokens, Expressions and Control Structures



User Defined Data Types

Example

Student s = new Student();

In java we have eight data type which are organized in four groups. They are

1. Integer category data types
2. Character category data types
3. Float category data types
4. Boolean category data types

2. Tokens, Expressions and Control Structures



User Defined Data Types

1. Integer category data types

Data Type	Size (byte)	Range
Byte	1	+ 127 to -128
Short	2	+ 32767 to -32768
Int	4	+ x to - (x+1)

2. *Character category data types*

A character is an identifier which is enclosed within single quotes. In java to represent character data, we use a data type called char. This data type takes two byte since it follows Unicode character set.

Data Type	Size(Byte)	Range
Char	2	232767 to -32768

2. Tokens, Expressions and Control Structures



User Defined Data Types

3. Float category data types

Float category data type are used for representing float values. This category contains two data types, they are in the given table

Data Type	Size	Range	Number of decimal places
Float	4 byte	+2147483647 to -2147483648	8
Double	8 byte	+ 9.223*1018	16

4. Boolean category data types

Boolean category data type is used for representing or storing logical values is true or false. In java programming to represent Boolean values or logical values, we use a data type called Boolean.

2. Tokens, Expressions and Control Structures



Assignment

Why Java take 2 byte of memory for store character ?

Java support more than 18 international languages so java take 2 byte for characters, because for 18 international language 1 byte of memory is not sufficient for storing all characters and symbols present in 18 languages. Java supports Unicode but c support ascii code. In ascii code only English language are present, so for storing all English latter and symbols 1 byte is sufficient. Unicode character set is one which contains all the characters which are available in 18 international languages and it contains 65536 characters.

Why Boolean data types take zero byte of memory ?

Boolean data type takes zero bytes of main memory space because Boolean data type of java implemented by Sun Micro System with a concept of flip - flop. A flip - flop is a general purpose register which stores one bit of information (one true and zero false).

Note: In C, C++ (Turbo) Boolean data type is not available for representing true false values but a true value can be treated as non-zero value and false values can be represented by zero.

2. Tokens, Expressions and Control Structures



Identifier

- An Identifier is needed to name a variable.
- An identifier is sequence of character, of any length, comprising uppercase and lowercase letters (**a-z, A-Z**), digits (**0-9**), underscore **_** and dollar sign **\$**.

Java imposes the following rules on identifiers

1. White space (**blank, tab, newline**) and other special characters (such as **+, -, *, /, @, &, commas**, etc.) are not allowed.
2. An identifier must begin with letter (**a-z, A-Z**) or underscore (**_**).
3. It cannot begin with digits (**0-9**).
4. Identifiers begin with dollar sign **\$** are reserved for system-generated entities. So, you can not use it.
5. An identifier cannot be a reserved keyword or a reserved literals (e.g. **class, int, double, if, else, for, true, false, null**).
6. Identifiers are case sensitive. A **SPM.com.np** is not same as **spm.com.np**.

2. Tokens, Expressions and Control Structures



Literals

- A literal is a specific constant value or raw data, such as **123**, **-456**, **3.14**, **'a'**, **"hello"**, that is used in the program source.
- They are called literals because they *literally* and *explicitly* identify their values.
- It can be of any Java data types.

2. Tokens, Expressions and Control Structures



Constants

Constants are declared with keyword **final**. Their values cannot be changed during program execution.

For example:

```
final double PI = 3.1415926; // Need to initialize
```

Note:

*A final local variable that has been declared but not yet initialized is called a blank **final**.*

2. Tokens, Expressions and Control Structures



Variables

- A variable is used to store a piece of data for processing. It is called variable because you can change the value stored.
- A variable is a named storage location, which stores a value of a particular data type. In other words, a variable has a name, a type and stores a value of that type.
- A variable has a type. A variable can store a value of that particular type.

```
int x = 100;  
short y = 5;  
float z = 365.24;
```


2. Tokens, Expressions and Control Structures



Type Conversion and Casting

There are two kind of type conversion in Java:

1. **Implicit type-conversion**
2. **Explicit type-casting**

2. Tokens, Expressions and Control Structures



Type Conversion and Casting

1. Implicit type-conversion:

Implicit type-conversion performed by the *compiler automatically*, if there will be no loss of precision.

Example:

```
int i = 3;  
double f;  
f = i; // it is ok, no explicit type casing required.  
// here, f = 3.0
```

Widening Conversion:

The rule is to promote the smaller type to bigger type to prevent loss of precision, known as **Widening Conversion** (as above example).

2. Tokens, Expressions and Control Structures



Type Conversion and Casting

2. Explicit type-casting:

- Explicit type-casting performed via a type-casting operator in the prefix form of (*new-type*) operand.
- Type-casting forces an explicit conversion of type of a value. Type casting is an operation which takes one operand, operates on it and returns an equivalent value in the specified type.

Syntax:

newValue = (typecast)value;

Example:

```
double f = 3.5; int i; i = (int)f; // it cast double value 3.5 to int 3.
```

Narrowing Casting:

Explicit type cast is requires to Narrowing conversion to inform the compiler that you are aware of the possible loss of precision (as above example).

<https://ctaljava.blogspot.com/>

2. Tokens, Expressions and Control Structures



Type Conversion and Casting

2. Explicit type-casting:

- Explicit type-casting performed via a type-casting operator in the prefix form of (*new-type*) operand.
- Type-casting forces an explicit conversion of type of a value. Type casting is an operation which takes one operand, operates on it and returns an equivalent value in the specified type.

Syntax:

newValue = (typecast)value;

Example:

```
double f = 3.5; int i; i = (int)f; // it cast double value 3.5 to int 3.
```

Narrowing Casting:

Explicit type cast is requires to Narrowing conversion to inform the compiler that you are aware of the possible loss of precision (as above example).

<https://ctaljava.blogspot.com/>

2. Tokens, Expressions and Control Structures



Assignment

1. Scope and Default Values of Variables.
2. What is wrapper class?

2. Tokens, Expressions and Control Structures

How to get input from user in Java

Java Scanner Class

- Java **Scanner class** allows the user to take input from the console.
- It belongs to **java.util** package.
- It is used to read the input of primitive types like int, double, long, short, float, and byte.
- It is the easiest way to read input in Java program.

Syntax

Scanner sc=**new** Scanner(System.in);

Method	Description
int nextInt()	It is used to scan the next token of the input as an integer.
String nextLine()	Advances this scanner past the current line.
double nextDouble()	It is used to scan the next token of the input as a double.

2. Tokens, Expressions and Control Structures

How to get input from user in Java

Example of integer input from user

```
1.import java.util.*;
2.class UserInputDemo
3.{
4.public static void main(String[] args)
5.{
6.Scanner sc= new Scanner(System.in); //System.in is a standard input stream
7.System.out.print("Enter first number- ");
8.int a= sc.nextInt();
9.System.out.print("Enter second number- ");
10.int b= sc.nextInt();
11.System.out.print("Enter third number- ");
12.int c= sc.nextInt();
13.int d=a+b+c;
14.System.out.println("Total= " +d);
15.}
16.}
```

Output:



```
C:\demo>javac UserInputDemo.java
C:\demo>java UserInputDemo
Enter first number- 6
Enter second number- 44
Enter third third- 23
Total= 73
C:\demo>
```

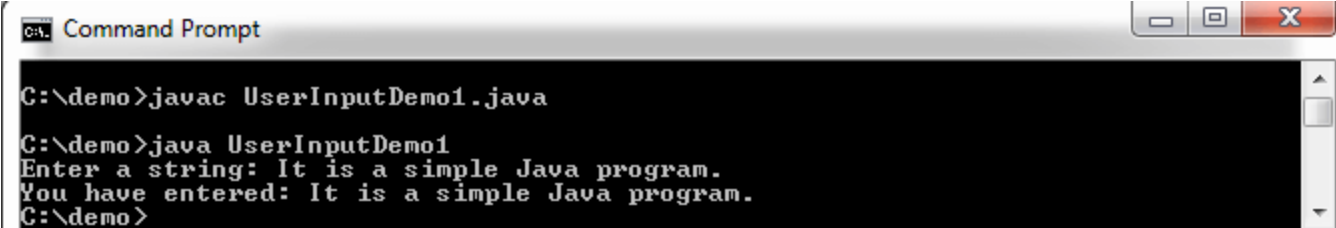
2. Tokens, Expressions and Control Structures

How to get input from user in Java

Example of String input from user

```
import java.util.*;
class UserInputDemo1
{
    public static void main(String[] args)
    {
        Scanner sc= new Scanner(System.in); //System.in is a standard input stream
        System.out.print("Enter a string: ");
        String str= sc.nextLine();           //reads string
        System.out.print("You have entered: "+str);
    }
}
```

Output:



```
cmd: Command Prompt
C:\demo>javac UserInputDemo1.java
C:\demo>java UserInputDemo1
Enter a string: It is a simple Java program.
You have entered: It is a simple Java program.
C:\demo>
```


2. Tokens, Expressions and Control Structures



Comment Syntax

Comments are used to document and explain your codes and program logic. Comments are not programming statements and are ignored by the compiler, but they VERY IMPORTANT for providing documentation and explanation for others to understand your program.

The syntax of the java programming language supports two types of commenting illustrated in the following table.

1. Single-line Comments

- Single-line comments start with two forward slashes (//).
- Any text between // and the end of the line is ignored by Java (will not be executed).

2. Java Multi-line Comments

- Multi-line comments start with /* and ends with */.
- Any text between /* and */ will be ignored by Java.

Ex 1. `Int x; // a comment`

Ex2. `/* the variable x is an integer */`

- ***Comments can be used to explain Java code, and to make it more readable.***
- ***It can also be used to prevent execution when testing alternative code.***

2. Tokens, Expressions and Control Structures

Arrays of Primitive Data Types

- Arrays are objects that hold multiple values of the **same type**.
- Each data value stored in an array is called an *element*.
- Each element is accessed using an integer *index* or *subscript*.
 - As with strings, the first subscript is 0.
- Organizing data in an array allows programs to access huge amount of data multiple times and in any order!

2. Tokens, Expressions and Control Structures

Arrays of Primitive Data Types

- Arrays are objects that hold multiple values of the **same type**.
- Each data value stored in an array is called an *element*.
- Each element is accessed using an integer *index* or *subscript*.
 - As with strings, the first subscript is 0.
- Organizing data in an array allows programs to access huge amount of data multiple times and in any order!

2. Tokens, Expressions and Control Structures



Arrays of Primitive Data Types

Array Declaration

Syntax :

`datatype[] identifier; or datatype identifier[];`

Example :

```
int[] arr;  
char[] arr;  
short[] arr;  
long[] arr;  
int[][] arr; //two dimensional array.
```

Initialization of Array

```
int[] arr = new int[10]; //10 is the size of array.  
or  
int[] arr = {10,20,30,40,50};
```

2. Tokens, Expressions and Control Structures

Arrays of Primitive Data Types

Accessing array element

As mention ealier array index starts from 0. To access nth element of an array.

Syntax

arrayname[n-1];

Example : To access 4th element of a given array

```
int[] arr={10,20,30,40};
```

```
System.out.println("Element at 4th place"+arr[3]);
```

The above code will print the 4th element of array arr on console.



Garbage Collection

Before Java

(Why so much hype about GC ?)

Before java “you” had to allocate and de-allocate memory.
How cool that would be? Not very cool.

malloc() / **realloc()** / **calloc()**

free()

• **new** and **destructors**

2. Tokens, Expressions and Control Structures

Garbage Collection

RAM



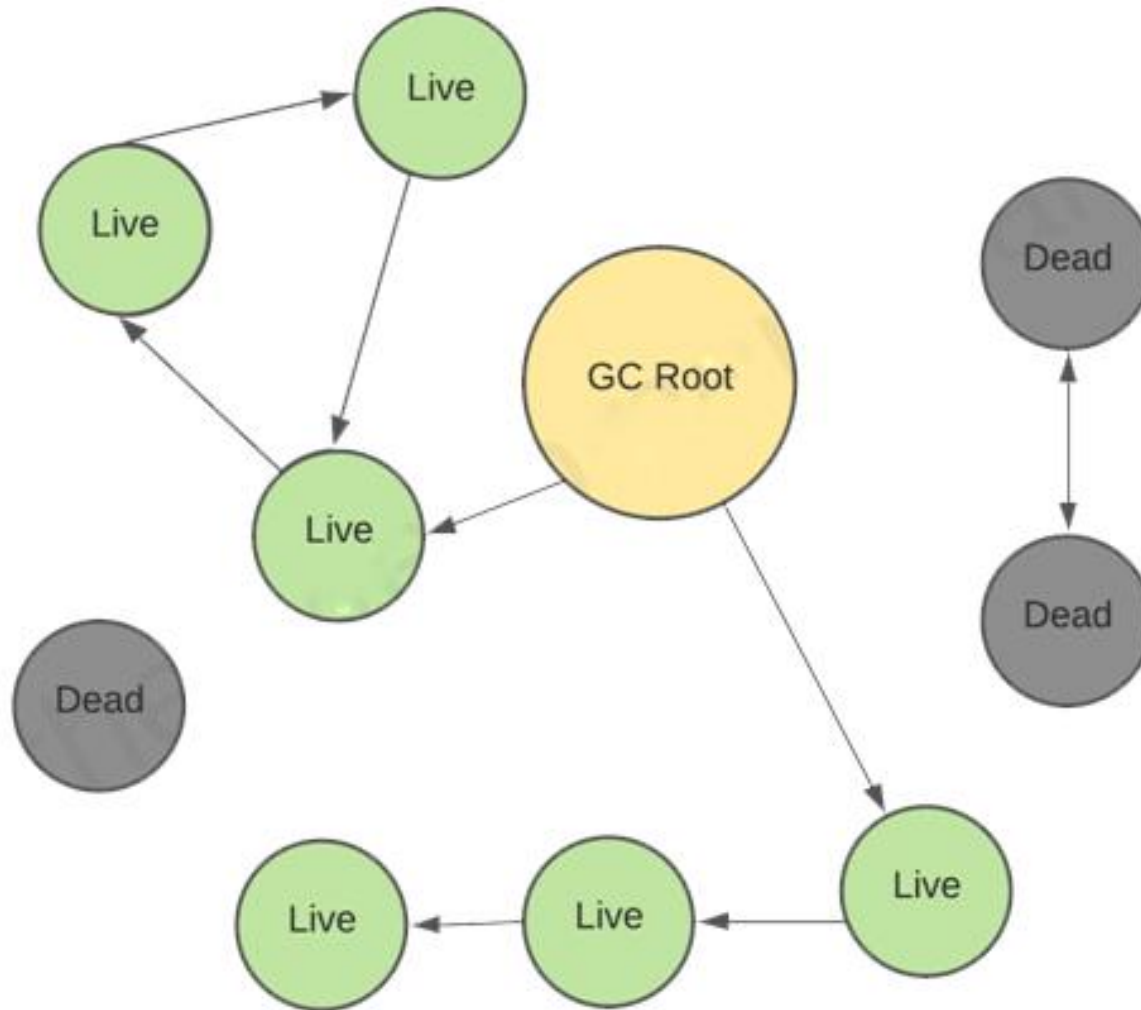
Stack vs Heap Memory

<https://www.youtube.com/watch?v=3onXUw1303g>

<https://ctaljava.blogspot.com/>

2. Tokens, Expressions and Control Structures

Garbage Collection

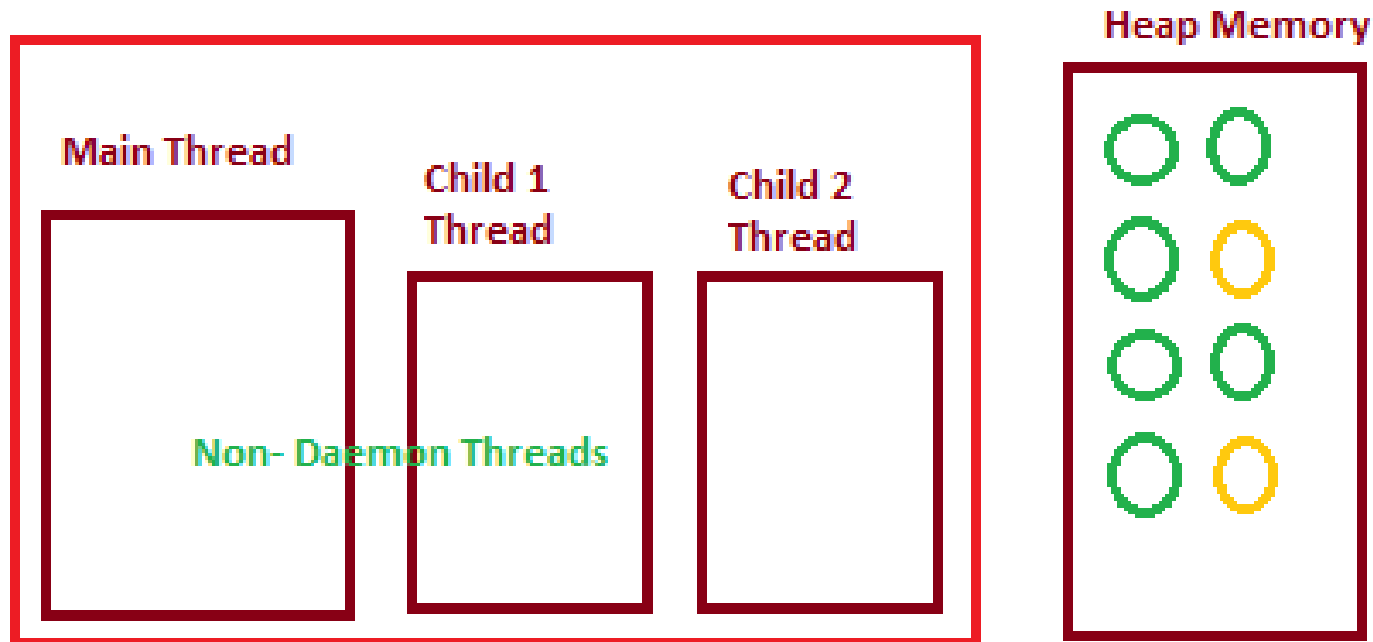


2. Tokens, Expressions and Control Structures



Garbage Collection

Garbage Collection (GC) is an automated dynamic memory management That identifies dead memory blocks and reallocates storage for reuse.



VisualGC

<https://plugins.jetbrains.com/plugin/14557-visualgc>

<https://ctaljava.blogspot.com/>

2. Tokens, Expressions and Control Structures



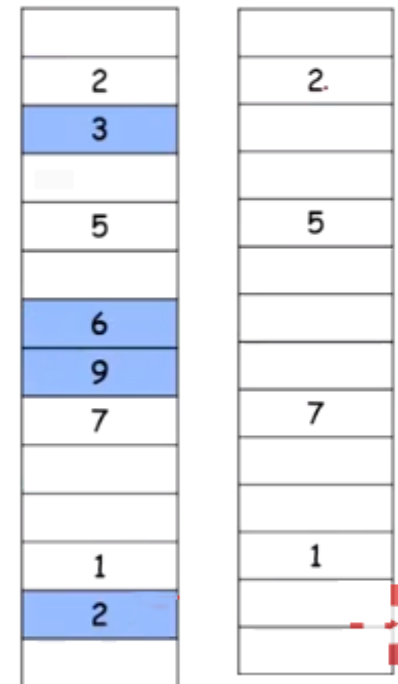
Garbage Collection

When to run GC?

1. No free storage left
2. Periodically (time interval)
3. Free storage reach a threshold (i.e. 100 units total → run in 20 units free)
4. System is idle

Two Steps Process:

1. GC will sequentially visit all nodes in memory, and mark all nodes which are being used in programs.
2. It will collect all unmarked nodes and place them in free storage area. .



UnUsed
Memory
Data

Used
Memory

2. Tokens, Expressions and Control Structures

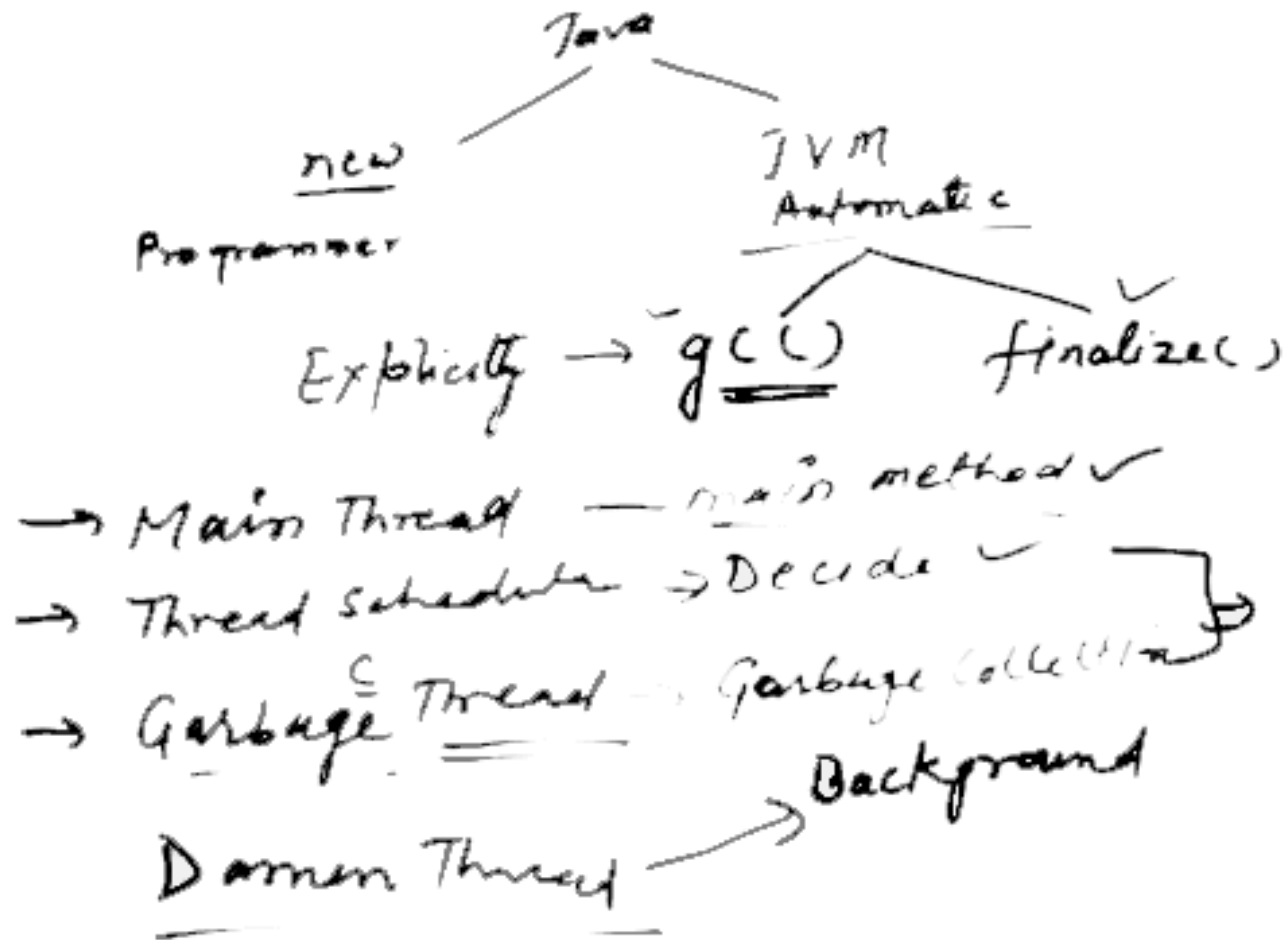
Garbage Collection

JVM threads

- ☐ Whenever you run a java program, JVM creates three threads.
 - main thread
 - Thread Scheduler
 - Garbage Collector Thread.
- ☐ In these three threads, main thread is a user thread and remaining two are daemon threads which run in background.

2. Tokens, Expressions and Control Structures

Garbage Collection



2. Tokens, Expressions and Control Structures



Assignment

Is garbage collection in Java good or bad?

Definitely good. But, as the short goes, too much of anything is a bad thing. So, you need to make sure Java heap memory is properly configured and managed so that the GC activity is optimized.

When is Java GC needed?

It is needed when there are unreferenced objects to be cleared out. Since it is not a manual activity, the JVM will automatically take care of this for you. From all the information above, you would have learned why GC is needed and when.

```
public class TestGarbage1{
    public void finalize()
    {
        System.out.println("object is garbage collected");
    }
    public static void main(String args[]){
        TestGarbage1 s1=new TestGarbage1();
        TestGarbage1 s2=new TestGarbage1();
        s1=null;
        s2=null;
        System.gc();
    }
}
```

2. Tokens, Expressions and Control Structures

Assignment

Java Garbage Collection

↳ means unreferenced objects.

It is the process of reclaiming the runtime unused memory automatically.

Advantages

- i) Makes Java Memory Efficient
- ii) Automatically done. (JVM)

What is the meaning of Unreferenced object?

- i) By nulling the reference
- ii) By assigning ref. to another
- iii) By anonymous object.

```
→ A a = new A();  
   a = null;  
→ A b = new A();  
   a = b;  
→ new A();
```

Methods.

object → finalize()
before object is garbage collected.

System and Runtime. → gc()
↳ invoke garbage collⁿ.

Example

```
class C1 { b.s.v.m(String[] ar) {
```

```
C1 a = new C1();
```

```
C1 b = new C1();
```

```
a = null; b = null;
```

```
System.gc();
```

O/P:

G.C

G.C

```
public void finalize() {
```

```
{ S.O.P("G.C");
```

```
}
```

2. Tokens, Expressions and Control Structures



Expressions

Expressions in Java are used to fetch, compute, and store values.

To fetch a value, you use a type of expression called a primary expression.
To compute and store values, we use the various operators.

A Java expression consists of variables, operators, literals, and method calls.

For example,

```
int score;  
score = 90;
```

Here, score = 90 is an expression that returns an int.

Consider another example,

```
Double a = 2.2, b = 3.4, result;  
result = a + b - 3.4;
```

Here, a + b - 3.4 is an expression.

Consider another example,

```
if (number1 >= number2)  
    System.out.println("Number 1 is larger than number 2");
```

Here, number1 >= number2 is an expression that returns a boolean value.
Similarly, "Number 1 is larger than number 2" is a string expression.

```
// statement  
    int score = 9*5;  
// expression 9 * 5.  
In Java, expressions are  
part of statements.
```

2. Tokens, Expressions and Control Structures



Using Operators

1. Arithmetic
2. Bitwise
3. Relational
4. Logical
5. Assignment
6. Conditional
7. Shift
8. Ternary
9. Auto-increment and Auto-decrement

2. Tokens, Expressions and Control Structures

Using Operators



	Operator Type	Category	Precedence
1	Unary	postfix	<i>expr++ expr--</i>
		prefix	<i>++expr --expr +expr -expr</i>
2	Arithmetic	multiplicative	<i>* / %</i>
		additive	<i>+ -</i>
3	Shift	shift	<i><< >> >>></i>
4	Relational	comparison	<i>< > <= >=</i>
		equality	<i>== !=</i>
5	Bitwise	bitwise AND	<i>&</i>
		bitwise exclusive OR	<i>^</i>
		bitwise inclusive OR	<i> </i>
6	Logical	logical AND	<i>&&</i>
		logical OR	<i> </i>
7	Ternary	ternary	<i>? :</i>
8	Assignment	assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>

2. Tokens, Expressions and Control Structures



Using Operators

Arithmetic Operators

Java supports following arithmetic operator.

Operator	Description	Usage	Example
*	Multiplication	$a * b$	$2 * 3 \rightarrow 6$
/	Division	a / b	$10 / 5 \rightarrow 2$
%	Modulus (return Remainder of Division)	$a \% b$	$10 \% 5 \rightarrow 0$
+	Addition (or unary positive)	$a + b$	$5 + 2 \rightarrow 7$
-	Subtraction (or unary negative)	$a - b$	$5 - 2 \rightarrow 3$

2. Tokens, Expressions and Control Structures

Using Operators

Java Unary Operator Example: ++ and --

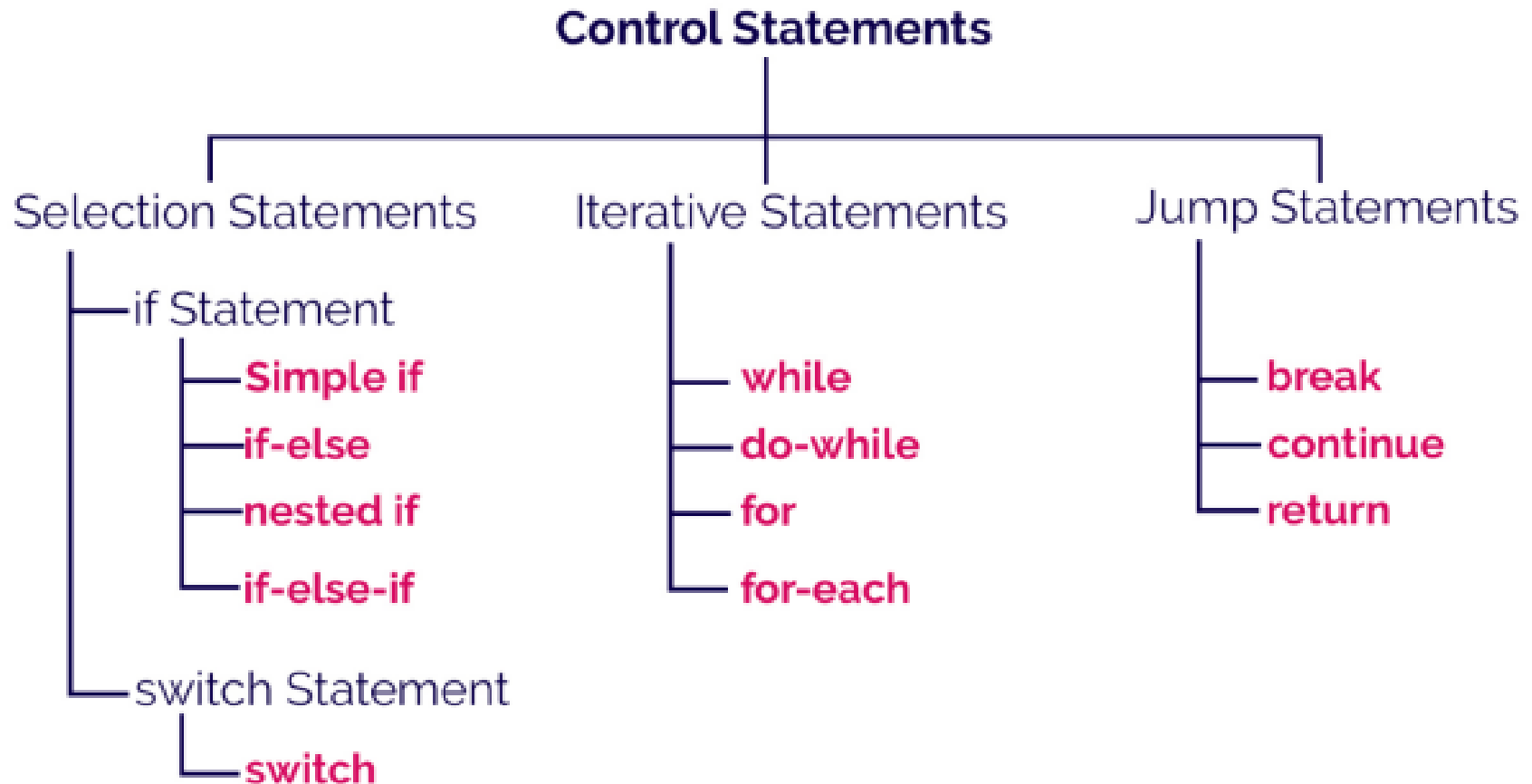
```
1. public class OperatorExample{  
2. public static void main(String args[]){  
3. int x=10;  
4. System.out.println(x++); //10 (11)  
5. System.out.println(++x); //12  
6. System.out.println(x--); //12 (11)  
7. System.out.println(--x); //10  
8. }  
9. }
```

Output:

10 12 12 10

2. Tokens, Expressions and Control Structures

Using Control Statements



2. Tokens, Expressions and Control Structures

Using Control Statements

1. (Branching)Decision Making statements
 - i. if statements
 - ii. switch statement
2. Looping statements
 - i. do while loop
 - ii. while loop
 - iii. for loop
 - iv. for-each loop
3. Jumping statements
 - i. break statement
 - ii. continue statement
 - iii. return

2. Tokens, Expressions and Control Structures



Using Control Statements

for-each loop Example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachDemo {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hockey");
        System.out.println("--Iterating by passing lambda expression--");
        // gamesList.forEach(System.out::print);
        for(String g : gamesList){
            System.out.println(g);
        }
    }
}
```

2. Tokens, Expressions and Control Structures

Using Control Statements

```
public class ReturnTypeTest1 {  
    public int add() { // without arguments  
        int x = 30;  
        int y = 70;  
        int z = x+y;  
        return z;  
    }  
    public static void main(String args[]) {  
        ReturnTypeTest1 test = new ReturnTypeTest1();  
        int add = test.add();  
        System.out.println("Sum: " + add);  
    }  
}
```

Return Example

```
public class ReturnTypeTest3 {  
    public static int add(int x, int y) {  
        int z = x+y;  
        return z;  
    }  
    public static void main(String args[]) {  
        int add = add(10, 20);  
        System.out.println("Sum: " + add);  
    }  
}
```

```
public class ReturnTypeTest2 {  
    public int add(int x, int y) { // with arguments  
        int z = x+y;  
        return z;  
    }  
    public static void main(String args[]) {  
        ReturnTypeTest2 test = new ReturnTypeTest2();  
        int add = test.add(10, 20);  
        System.out.println("Sum: " + add);  
    }  
}
```

2. Tokens, Expressions and Control Structures

Using Control Statements

ContinueExample

*//Java Program to demonstrate the use of continue statement
//inside the for loop.*

```
public class ContinueExample {  
    public static void main(String[] args) {  
        //for loop  
        for(int i=1;i<=5;i++){  
            if(i==3){  
                //using continue statement  
                continue;//it will skip the rest statement  
            }  
            System.out.println(i);  
        }  
    }  
}
```

Output:

1
2
4
5

2. Tokens, Expressions and Control Structures

Motivate

**We fall. We fail. We break.
But then, we rise. We heal.
We overcome.**

