JAVA FULL STACK PROGRAM

Java Standard Edition Basic

Outline

- Object vs Class
- Data Types (Primitive Types vs References), Wrapper Class
- Memory (Stack vs Heap)
- String & String Pool
- Comment
- Operator
- Flow Control
- Class
- Modifier
- Deep Copy vs Shallow Copy

Object vs Class

A class is a blueprint or template for creating objects

An object is an instance of a class that has state and behavior

- An object is a combination of data and procedures working on the available data
- An object has a state and behavior. The state of an object is stored in fields (variables), while methods (functions) display the object's behavior

Data Types

A variable is a data container that saves the data values during Java program execution. A variable is assigned with a data type.

Java distinguishes two kinds of entities

1. Primitive Types:
   Data stored directly in primitive-type variables.
2. Non-Primitive Types (References):
   Reference variables store the memory address of an object.

Data Types - Primitive Types

Data Type	Size	Description
byte	8 bits	Stores whole numbers from -2^7 to 2^7 -1
short	16 bits	Stores whole numbers from -2^15 to 2^15 -1
int	32 bits	Stores whole numbers from -2^31 to 2^31 -1
long	64 bits	Stores whole numbers from -2^63 to 2^63 -1

> An int is 32 bits. 2^32 is 4,294,967,296. Divide that by 2 and you get 2,147,483,648. Then subtract 1 as we did with bytes and you get 2,147,483,647, which is 2^32/2 - 1.

Data Types - Primitive Types (Continued)

Data Type	Size	Description
float	32 bits	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	64 bits	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	16 bits	Stores a single character/letter or ASCII values

Data Types - Declaring and Initializing Primitive Types

```java
// Declaring a variable
int a, b, c;

// Declaring and initializing variables
int a = 10, b = 20, c = 30;

// Declaring an approximation of pi
double pi = 3.14159;

// Declaring 3 variables, initialize one of them
int a, b, c = 30;

// Declaring a variable, initializing it later
int a;
a = 10;
```

Data Types - Declaring and Initializing Primitive Types (Advanced)

```
// Declaring different data types in one statement
int a = 10, double b = 20.5; // Not compile

// Declaring same data types in one statement
int a = 10, int b = 20, c = 30; // Not compile

// Declaring different data types in one line
int a = 10; double b = 20.5; // Compile, but not recommended
```

Data Types - Primitive Types Conversion vs Casting

Conversion

- Automatic conversion takes place when two types are compatible and the target type is larger than the source type.
- For example, when we assign an int value to a long variable, the `int` is automatically converted to `long`.

Casting

- Casting is required when we assign a larger type to a smaller type.
- For example, when we assign a `double` value to an `int` variable, we need to cast double to int. `int a = (int) 3.14;`
- Casting will lose precision

Data Types - Primitive Types Conversion vs Casting (Examples)

```java
// Automatic conversion
long a = 2; // a = 2 is automatically converted to a = 2L

// Casting
int a = (int) 3.14; // a = 3 (3.14 is truncated)

// Casting will lose precision
double n = (int) (5.0/2.0); // n = 2.0 (5.0/2.0 = 2.5, 0.5 is truncated)

// Casting need to be explicit
float f = 3.14; // Not compile (3.14 is double, need to cast to float)
```

Wrapper Class

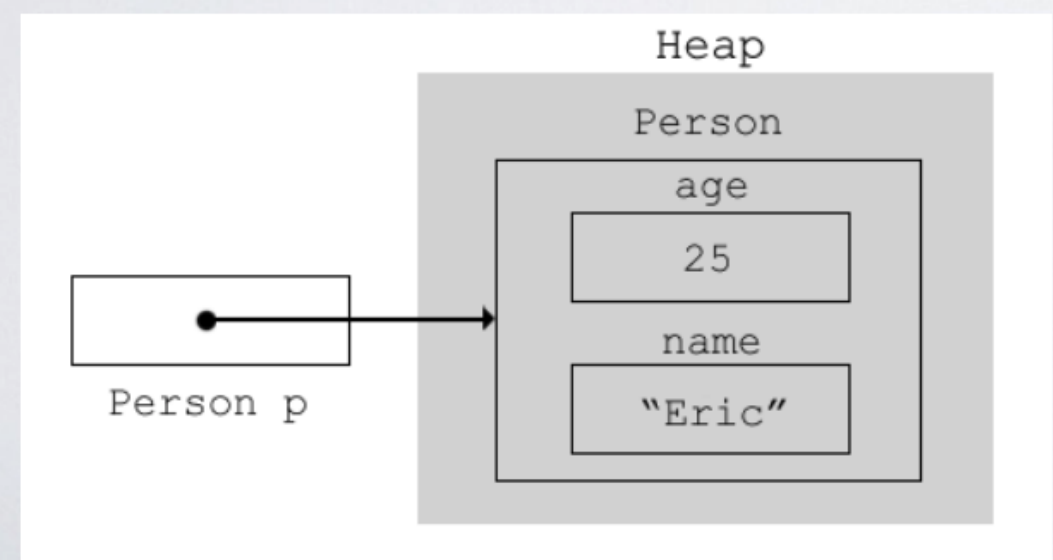
- Primitive data types are not objects. They do not belong to any class.
- Wrapper classes are used to convert primitive data types into objects, like `int` to `Integer`, `double` to `Double`, etc. Converting primitive data types into object is called boxing, and this is taken care by the compiler.
- The wrapper classes are part of the `java.lang` package, which is imported by default into all Java programs.
- All wrapper classes are immutable, i.e., they cannot be changed once they have been created.
- Used for collections, generics, and reflection.

There are 8 wrapper classes: `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`.

Data Types - Non-Primitive Types (References)

- Non-primitive data types are called reference types because they refer to objects.
- Reference types are created using the keyword `new` followed by a call to a constructor.
- A constructor executes when a new object is created

```
Person p = new Person(25, "Eric");
```




```
public class Person {
    1 usage
    int age;
    1 usage
    String name;

    public Person(int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

Stack vs Heap

Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer's RAM.

Stack:

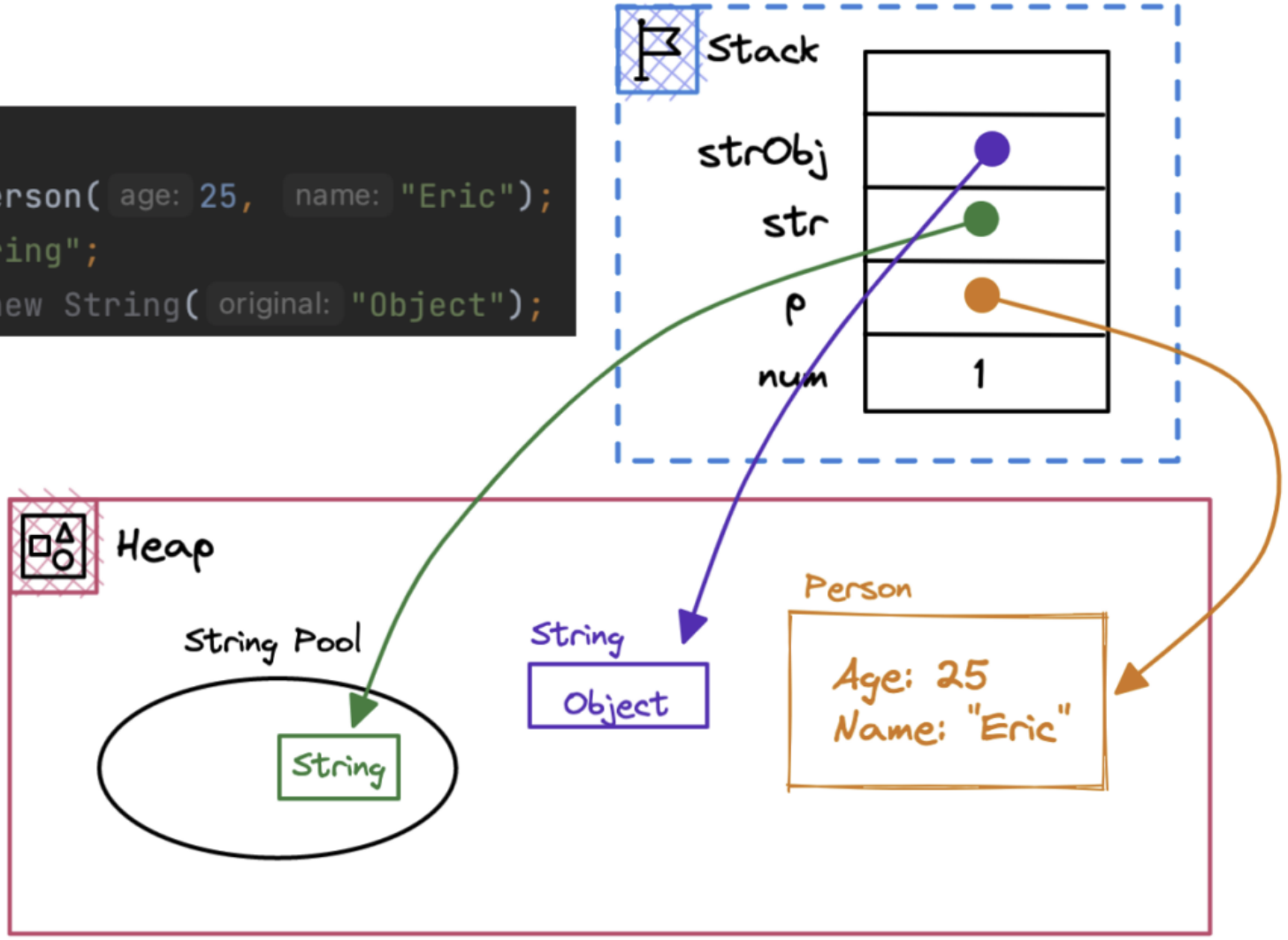
- Contains primitive values and references to objects
- Each thread has its own stack
- Stores data in a last-in-first-out (LIFO) manner
- Faster access

Heap:

- Contains objects and arrays
- Objects created on the heap are globally accessible

Stack vs Heap

```
int num = 1;
Person p = new Person( age: 25,  name: "Eric");
String str = "String";
String strObj = new String( original: "Object");
```



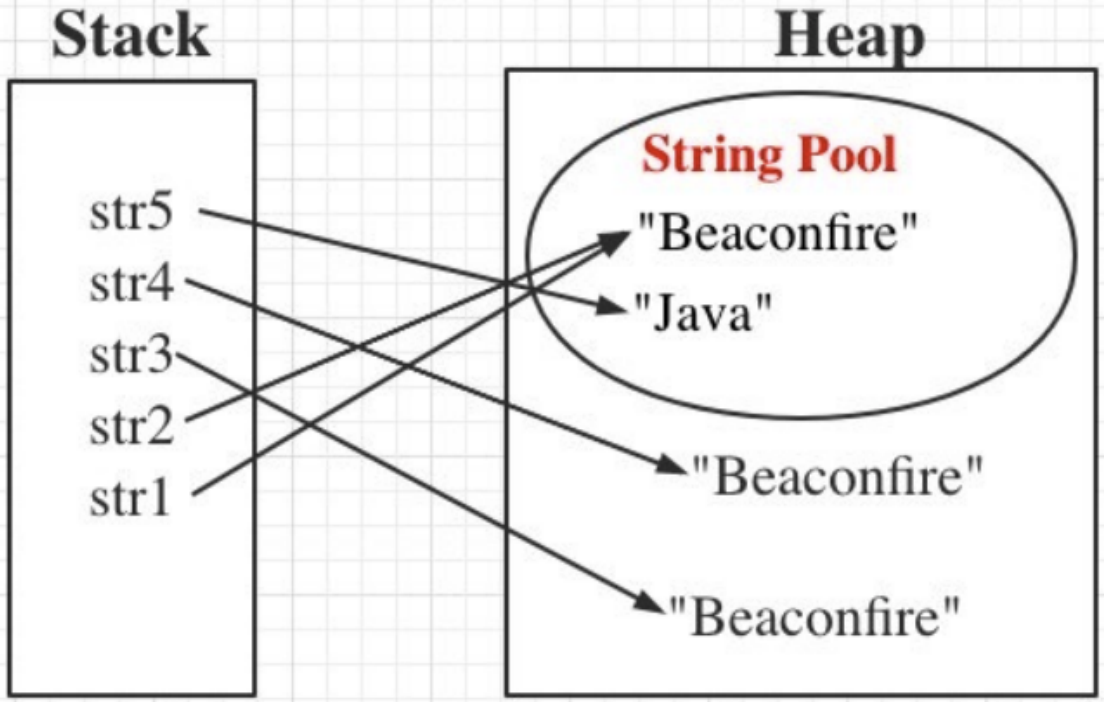
String and String Pool

- A String is a sequence of characters. In java, String objects are **immutable**, which means that once they are created, they cannot be changed.
- String objects are stored in a special memory area known as the **String Pool**.
- The String Pool is located in the heap memory area.
- When a String object is created, it is placed in the String Pool. If another String object is created with the same value, a reference to the existing String object will be returned, instead of creating a new object.

String and String Pool (Examples)

```
String str1 = "Beaconfire";
String str2 = "Beaconfire";
String str3 = new String("Beaconfire");
String str4 = new String("Beaconfire");
String str5 = "Java";

str1==str2; //true
str1==str3;//false
str3==str4;//false
```



String Pool is used for optimization purposes. It saves a lot of memory by sharing one String object among several references.

Comments

Java supports three types of comments:

1. Single-line comments

   The compiler ignores everything from `//` to the end of the line

```
// This is a single-line comment
```

2. Multi-line comments

   The compiler ignores everything from `/*` to `*/`

```
/*
This is a multi-line comment
*/
```

3. Documentation comments

   This is a documentation comment and in general its called doc comment. The JDK javadoc tool uses doc comments when preparing automatically generated documentation.

```
/**
 * This is a documentation comment
 * @param args
 * @return
 * @throws Exception
 */
```

Operators

- Operators are used to perform operations on variables and values.
- Java divides the operators into the following groups:
    - Arithmetic operators
    - Assignment operators
    - Comparison operators
    - Logical operators
    - Bitwise operators

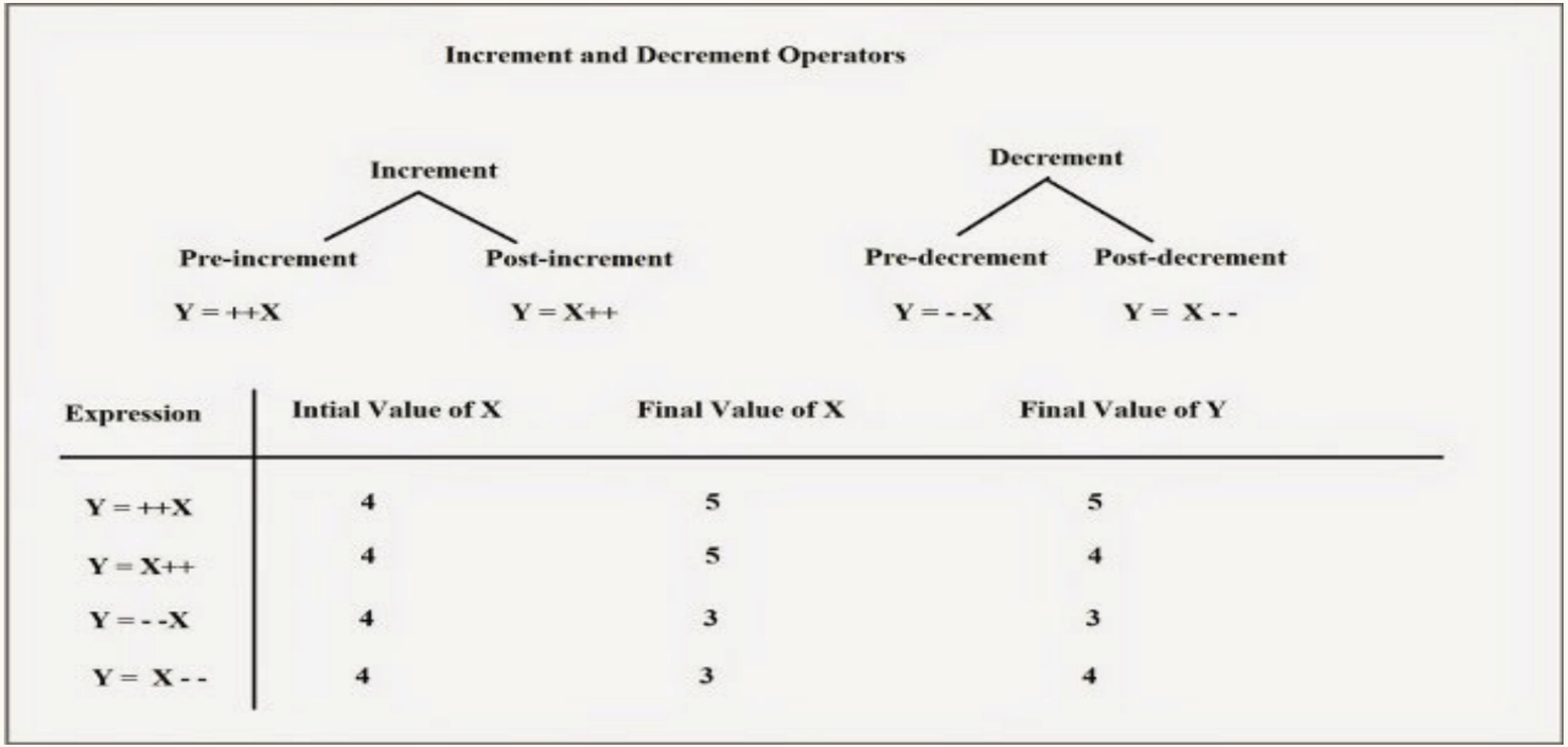
Order of operator precedence

Operator	Precedence
Post-unary operators	`expr++` , `expr--`
Pre-unary operators	`++expr` , `--expr`
Other unary operators	`+` , `-` , `!`
Multiplication/Division/Modulus	`*` , `/` , `%`
Addition/Subtraction	`+` , `-`

Order of operator precedence (cont.)

Operator	Precedence
Shift operators	`<<` , `>>` , `>>>`
Relational operators	`<` , `<=` , `>` , `>=`
Equal to/not equal to	`==` , `!=`
Logical operators	`&` , `^` , `` ` ``
Short-circuit logical operators	`&&` , `||`
Ternary operators	`? :`
Assignment operators	`=` , `+=` , `-=` , `*=` , `/=` , `%=` , `&=` , `^=` , `` ` ``

Increment and Decrement Operators



Expression	Intial Value of X	Final Value of X	Final Value of Y
Y = ++X	4	5	5
Y = X++	4	5	4
Y = - -X	4	3	3
Y = X - -	4	3	4

LOGICAL OPERATOR

Operator	Name	Type	Description
!	Not	Unary	Returns true if the operand to the right evaluates to false. Returns false if the operand to the right is true.
&&	Conditional And	Binary	If the operand on the left returns false, it returns false without evaluating the operand on the right.
||	Conditional Or	Binary	If the operand on the left returns true, it returns true without evaluating the operand on the right.

RELATIONAL OPERATORS

Operator	**Result**
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

FLOW CONTROL

- Selection Statements
    - if and switch
- Iteration Statements
    - while, do-while, for, and nested loops
- Jump Statements
    - break, continue, and return

Selection Statements

IF

```
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 20) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

Selection Statements

SWITCH

```
char c = 'A';
switch (c) {
case 'A': {
    System.out.println("A");
}
case 'B':{
    System.out.println("B");
}
default:
    System.out.println("Default");
}
```

A
B
Default

```
switch (c) {
case 'A': {
    System.out.println("A");
    break;
}
case 'B':{
    System.out.println("B");
    break;
}
default:
    System.out.println("Default");
}
```

A

Selection Statements

TERNARY EXPRESSION

- Variable = (condition) ? expressionTrue : expressionFalse

```
boolean valid = true;
int i;
if(valid) {
     i=1;
}else {
     i=0;
}
```

```
i = valid ? 1 : 0;
```

Iteration Statements

WHILE

```
while(condition)
  {
  // statements to keep executing while condition is true
  ..
  ..
  }
```

Example

```
  //Increment n by 1 until n is greater than 100
  while (n > 100) {
  n = n + 1;
  }
```

Iteration Statements

DO WHILE

```
Do {
   // statements to keep executing while condition is true
} while(condition)
```

It will first executes the statement and then evaluates the condition.

Example

```
int n = 5;
Do {
System.out.println(" n = " + n);
N--;
    } while(n > 0);
```

Iteration Statements

FOR

```
for(initializer; condition; incrementer)
{
// statements to keep executing while condition is true
}
```

Example

```
int i;
int length = 10;
for (i = 0; i < length; i++) {
. . .
// do something to the (up to 9 )
. . .
}
```

Jump Statements

Break

```
public static void main(String args[])
{
    // Initially loop is set to run from 0-9
    for (int i = 0; i < 10; i++)
    {
        // terminate loop when i is 5.
        if (i == 5)
            break;

        System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
}
```

Jump Statements

Continue

```
public static void main(String args[])
{
    for (int i = 0; i < 10; i++)
    {
        // If the number is even
        // skip and continue
        if (i%2 == 0)
            continue;

        // If number is odd, print it
        System.out.print(i + " ");
    }
}
```

Jump Statements

Return

```
public static void main(String args[])
{
    boolean t = true;
    System.out.println("Before the return.");

    if (t)
        return;

    // Compiler will bypass every statement
    // after return
    System.out.println("This won't execute.");
}
```

CLASS

- A class is a container that contains the block of code that includes fields, methods, constructors, getters and setters.

```
class Person{
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public void hi() {
        System.out.println("Hi, My name is "+name);
    }
}
```

Modifier

- Access Modifier
  - - specify accessibility of field, method, constructor or class
  - public, private, protected, default
- Non Access Modifier
  - - do not control access level, but provides other functionality
  - static, final, abstract, transient, synchronized

ACCESS MODIFIER SCOPE

Access Modifier	**Class or member can be referenced by…**
public	methods of the same class, and methods of other classes
private	methods of the same class only
protected	methods of the same class, methods of subclasses, and methods of classes in the same package
No access modifier (package access)	methods in the same package only

NON-ACCESS MODIFIER

STATIC

- Static:
    - Class level
    - initialized at compile time or early binding
- Non-Static
    - Object level
    - initialized at runtime time or dynamic binding

NON-ACCESS MODIFIER - Static

CLASS VARIABLE
INSTANCE VARIABLE

- What is the difference between following statements?
- public int x; // instance variable
- public static int x; // class (static) varibale

NON-ACCESS MODIFIER - Static

CLASSVARIABLE

INSTANCEVARIABLE

```java
public class VariableDemo {
    static int staticVariable=0;
    int instanceVariable=0;

    public static void main(String[] args) {

        System.out.println(staticVariable); //0
        //instance variable can only be accessed through Object reference
        System.out.println(instanceVariable);

        VariableDemo object1 = new VariableDemo();
        System.out.println(object1.instanceVariable); //object reference

        object1.staticVariable = 1;
        object1.instanceVariable = 1;

        VariableDemo object2 = new VariableDemo();

        //each object has its own copy of instance variable
        System.out.println(object2.instanceVariable);

        //common to all object of a class
        System.out.println(object2.staticVariable);
```

NON-ACCESS MODIFIER - Static

CLASSVARIABLE

INSTANCEVARIABLE

Class Variables	**Instance Variables**
Class variables are declared with keyword *static*.	Instance variables are declared without *static* keyword.
Class variables are common to all instances of a class. These variables are shared between the objects of a class.	Instance variables are not shared between the objects of a class. Each instance will have their own copy of instance variables.
As class variables are common to all objects of a class, changes made to these variables through one object will reflect in another.	As each object will have its own copy of instance variables, changes made to these variables through one object will not reflect in another object.
Class variables can be accessed using either class name or object reference.	Instance variables can be accessed only through object reference.

NON-ACCESS MODIFIER - Static

STATIC METHOD
NON-STATIC METHOD

- What is the difference between following statements?

- public int getX(); // non static method

- public static int getX(); // static method

NON-ACCESS MODIFIER - Static

STATIC METHOD
NON-STATIC METHOD

	static Method	Non-*static* Method
Access instance variables?	no	yes
Access *static* class variables?	yes	yes
Call *static* class methods?	yes	yes
Call non-*static* instance methods?	no	yes
Use the object reference *this*?	no	yes

NON-ACCESS MODIFIER

Final

- The final keyword can be used to make a class, method or variable immutable.

- Final variable:
    - Once a final variable is assigned a value, it becomes a constant and can no longer be changed.
- Final method:
    - Once a method is made final, it cannot be overridden.
- Final Class:
    - Once a class is made final, it cannot be extended.

MAIN METHOD

```
public static void main( String [] args )
{
   // application code
}
```

- main is a method
  - public – *main* can be called from outside the class
  - static – *main* can be called by the JVM without instantiating an object
  - void – main does not return a value

DEEP COPY VS SHALLOW COPY

- In OOP languages like Java, object copying is creating an exact copy of an existing object.

- **Shallow copy:**
  - If you modify the copied object, the original object will be modified as well. The shallow copy points to the reference of the original object.
  - To do shallow copy, we use the default clone() method.

- **Deep copy:**
  - If you modify the copied object, the original object will not be modified. The copy has its own reference address.
  - To do a deep copy, we use the new keyword and copy over the values one by one.

DEEP COPY VS SHALLOW COPY

- Shallow copy

```java
public class ShallowCopy {
    2 usages
    private int [] copy;

    //Shallow copying an object using the default cloning process
    1 usage
    public ShallowCopy(int[] copy) {
        this.copy = copy;
    }
    2 usages
    public void printArray(){
        System.out.println(Arrays.toString(copy));
    }
}
```

```java
//Testing shallow copy
ShallowCopy shallowCopy = new ShallowCopy(original);
shallowCopy.printArray(); //prints {1, 6, 9}
original[0] = 1;
shallowCopy.printArray(); // prints {1, 6, 9}
/*
Modifying the original array also changes the copied array,
causing unwanted side effects
 */
```

Deep copy

```java
2 usages
public class DeepCopy {
    3 usages
    private int[] copy;

    //Modified constructor to make a deep copy
    1 usage
    public DeepCopy(int[] original) {
        copy = new int[original.length];
        for(int i = 0; i < original.length; i++){
            copy[i] = original[i];
        }
    }
    2 usages
    public void printArray(){
        System.out.println(Arrays.toString(copy));
    }
}
```

```java
int [] original = {3, 6, 9};
//Testing deep copy
DeepCopy deepCopy = new DeepCopy(original);
deepCopy.printArray(); // prints {3, 6, 9}
original[0] = 1;
deepCopy.printArray(); //prints {3, 6, 9}
//Modifying the original array will not change the copied array
```

ANY QUESTIONS?