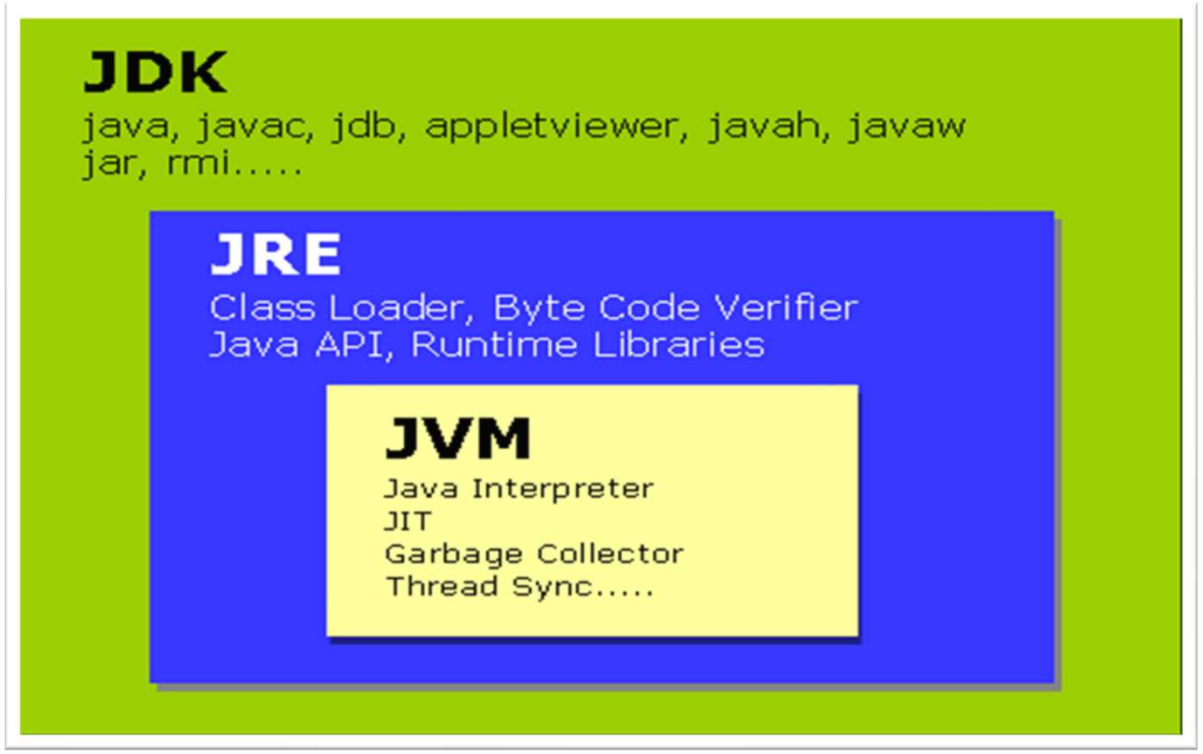
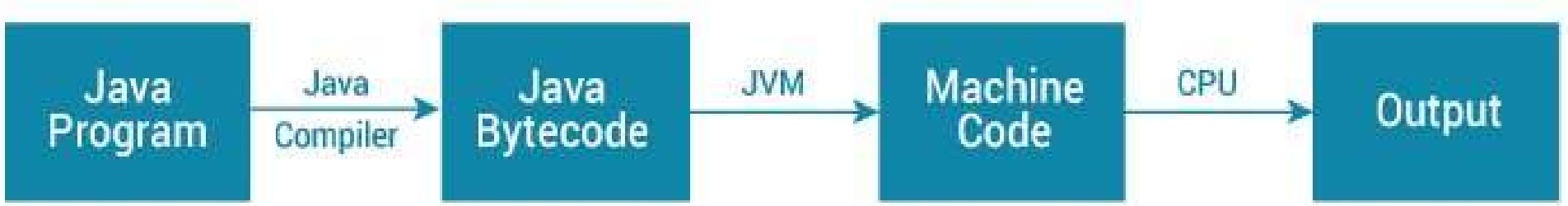
JDK is a software development kit whereas **JRE** is a software bundle that allows Java program to run, whereas **JVM** is an environment for executing bytecode.

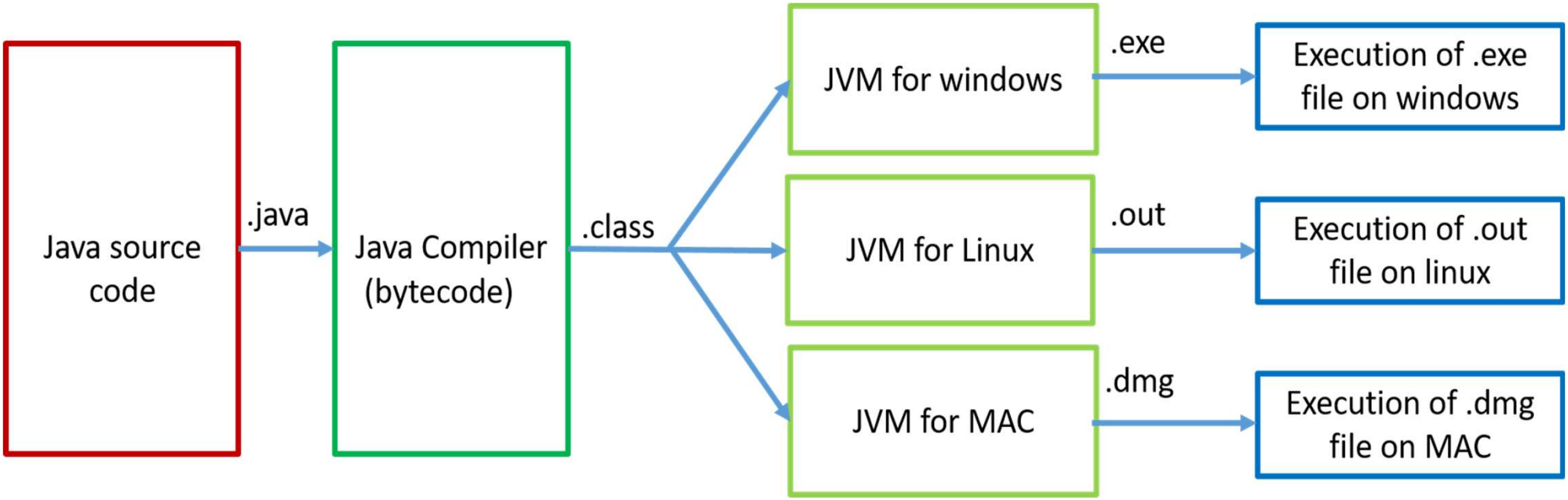


How Java Program works?



1. *We write a program in java and save the file with an extension .java called as source code*
2. *Then, the javac compiles the source code **and generates he bytecode** (.class file)*
3. *Then, JVM interprets the bytecode and generates a machine.*
4. *Thus, program runs and gives the desired output.*

Platform Independent/Portable



- *When we compile a Java file, output is not an .exe but it's a .class file. This .class is known as Java byte code; it has the Java byte codes which can be understood by JVM.*

Note - Every system has its own JVM which gets installed automatically when the jdk software is installed. For every operating system separate JVM is available which is capable to read the .class file or byte code.

Java Development Kit - JDK

- *JDK contains everything that JRE has along with development tools for developing, debugging, and monitoring Java applications.*
- *JDK used to develop Java applications.*
- *JDK contains compiler (javac.exe), Java application launcher (java.exe), Applet viewer, etc.,*
    - *javac - Java compiler translates java source code into byte code.*

Java Runtime Environment - JRE

- *JRE is a software package which bundles the libraries (jars) and the JVM, and other components to run applications written in the Java.*
- *To execute any Java application, you need JRE installed in the machine. It's the minimum requirement to run Java applications on any computer.*

Java Virtual Machine - JVM

- *JVM interprets the byte code into the machine code and execute it.*

Java In Time – JIT

- *JIT compiler is part of the JVM that helps to speed up the execution time*
- *Parts of the byte code that have similar functionality at the same time complied by JIT and saves time needed for compilation. JIT compiles only the reusable byte code to machine code*

Now, let's install JDK and write our first Java Program

- *https://phoenixnap.com/kb/install-java-windows*

Variables and Identifiers

- used for storing data/values
- To create a variable, you must specify the type and assign it a value
    - *type variable_name = values*

- All variables in java identified by a unique names called as identifiers
- Rules for naming a variables
    - Name should start the letter
    - Names can contain letters, digits, underscores, and dollar signs
    - Names can also begin with $ and _
    - Names are case sensitive ("myVar" and "myvar" are different variables)
    - Reserved word like Java Keywords can’t be used

Keywords: Java has a set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers

abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto*	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

Datatypes

- Specify
  - how the values of that data type are stored in memory
  - what operations can be performed on the data.

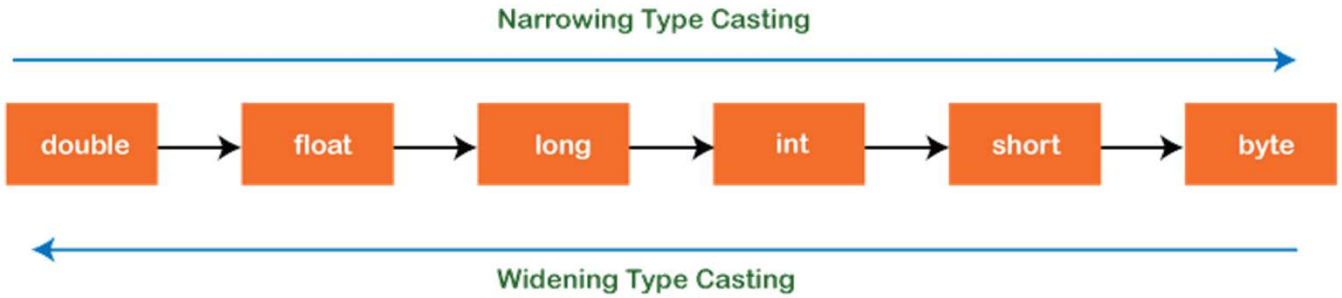
- Data types are divided into
  1. Primitive data types
  2. Non-primitive data types (or Reference Type)

Primitive Data Type

- Specify the size and type of variable values.
- Built in Java
- 8 types
- Numbers, decimal, characters, boolean

Keyword	Type	Range	Example
byte	8-bit Integral value	From -128 to 127	byte a = 57, byte b = -99;
short	16-bit Integral value	From -32768 to 32767	short a = 90, short b = -500;
int	32-bit Integral value	From 2147483648 (-2^31) to 2147483647 (2^31 -1)	int a = 678361, int b = -3470;
long	64-bit Integral value	From -92233720368547758O8(-2^63) to 9223372036854775807(2^63 -1)	long a = 300000L, long b = -57563526919L;
float	32-bit floating-point value	can have a 7-digit decimal precision and stores fractional numbers ranging from 3.4e-038 to 3.4e+038	float f1 = 47.85f;
double	64-bit floating-point value	can have a 15-digit decimal precision and stores fractional numbers ranging from 1.7e-308 to 1.7e+308	double d1 = 179.600985;
char	16-bit Unicode value	stores a single character	char char1='a'; char char2='A';
boolean	Stores true or false	true or false	boolean val = true;

```
double myDouble = 9.78d;
int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78
System.out.println(myInt); // Outputs 9
```



Type Casting in Java

Explicit Conversion
Narrowing Type casting
(double -> byte)

Automatic Type Conversion
Implicit Conversion
Widening Type Casting
(byte -> double)

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9
System.out.println(myDouble); // Outputs 9.0
```

Non-Primitive Data Type

- Reference Type
- Non-primitive data types refer to objects.
- They cannot store the value of a variable directly in memory. They store a memory address of the variable.
- Eg: Class and String

Points to be Noted:

Packages

- *Package is a collection of related classes. Assume package as a folder or a directory that is used to store similar files.*
- *Java uses package to group related **classes, interfaces and sub-packages** in any Java project.*
- *Package statement must be **first statement in the program** even before the import statement.*
- ***Types***
  - *In built - math, util, lang, io etc. are the example of built-in packages.*
  - *User defined – Created by user*

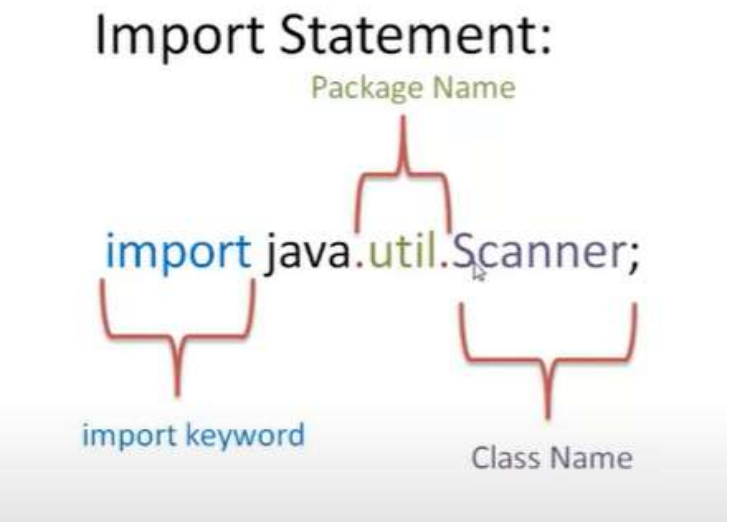
Import Statement

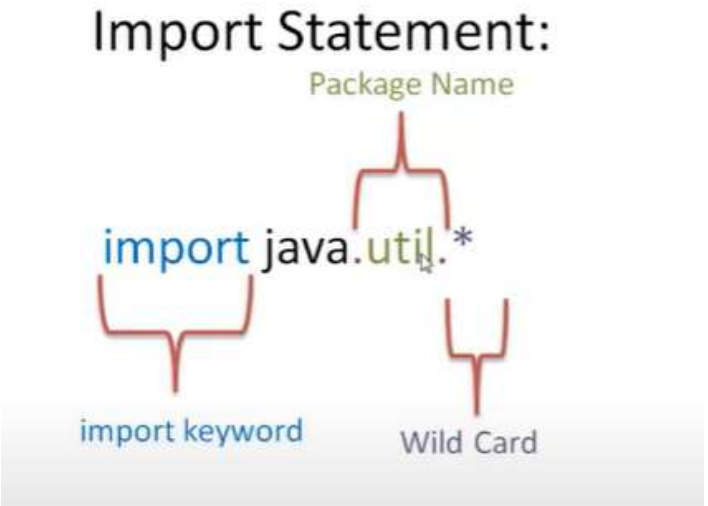
- *To import java package into a class, we need to use java **import** keyword which is used to access package and its classes into the java program.*
- *You can import the specific class or all the classes from the package*

java.lang package

- *Provides classes that are **fundamental** to the design of the Java programming language.*
- *Classes in that packages are **Object, System, String, all wrapper classes**, etc.,*
- ***Java compiler** imports java.lang package internally by default*.

Packages and Imports






```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
```

Data Type	Method
String	`String str = s.next();`
double	`double d = s.nextDouble();`
long	`long l = s.nextLong();`
short	`short sh = s.nextShort();`
byte	`byte b = s.nextByte();`
float	`float f = s.nextFloat();`
boolean	`boolean bo = s.nextBoolean();`

Calculate discount using primitive float

Problem

Calculate the Discount based on Bill Amount and Discount Percentage.

Discount = Bill Amount x Discount Percentage / 100

Example Calculation

Bill Amount = 1982

Discount Percentage = 18

Discount = 1982 x 18 / 100 = 356.76

Sample Output

Discounted Amount = 356.76

Exercise Problem

Calculate Fahrenheit value based on Celsius.

$Fahrenheit = \left(Celsius \times \frac{9}{5}\right) + 32$

Example Calculation

Celsius = 12

Fahrenheit = $\left(12 \times \frac{9}{5}\right) + 32 = 53.6$

Sample Output

Fahrenheit = 53.6

Arithmetic Operators

Operator	Meaning	Example	Result
+	Addition	10 + 2	12
-	Subtraction	10 – 2	8
*	Multiplication	10 * 2	20
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 2	0
++	Increment	a++ (consider a = 10)	11
--	Decrement	a-- (consider a = 10)	9
+=	Addition Assignment	a += 10 (consider a = 10)	20
-=	Subtraction assignment	a -= 10 (consider a = 10)	0
*=	Multiplication assignment	a *= 10 (consider a = 10)	100
/=	Division assignment	a /= 10 (consider a = 10)	1
%=	Modulus assignment	a %= 10 (consider a = 10)	0

Relational Operators

int a = 10, b = 2 for all examples below

Operator	Meaning	Example	Result
==	Equals to	a == b	false
!=	Not equal to	a != b	true
<	Less than	a < b	false
<=	Less than or equal to	a <= b	false
>	Greater than	a > b	true
>=	Greater than or equal to	a >= b	true

Logical Operators

int x = 8 for all examples

Operator	Name	Meaning	Example	Result
&&	Logical add	Returns true if both statements are true	1) x < 15 && x < 10 2) x < 5 && x <10	1) true 2) false
||	Logical or	Returns true if one of the statements is true	1) x > 15 || x > 10 2) x < 5 || x <10	1) false 2) true
!	Logical Not	Reverse the result, returns false if the result is true	1) !(x<15) 2) !(x>15)	1) false 2) true


(B)
O²
Brackets
Orders
Division
Multiplication
Addition
Subtraction
1st
Start with anything inside brackets.
2nd
Are there any powers or square roots?
Division and multiplication rank equally.
3rd
Working from left to right, calculate them in the order they appear.
Addition and subtraction also rank equally.
4th
Again, start from the left and work your way across the equation.

Simplify: $14 + (8 - 2 \times 3)$

Solution:

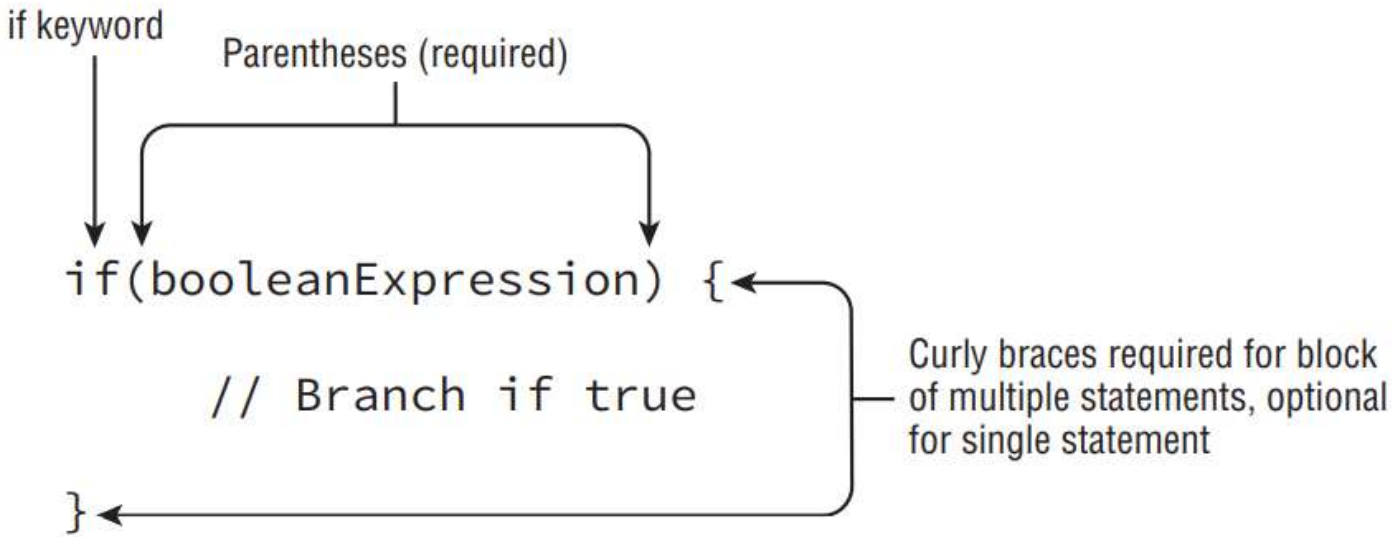
$14 + (8 - 2 \times 3)$

$= 14 + (8 - 6)$

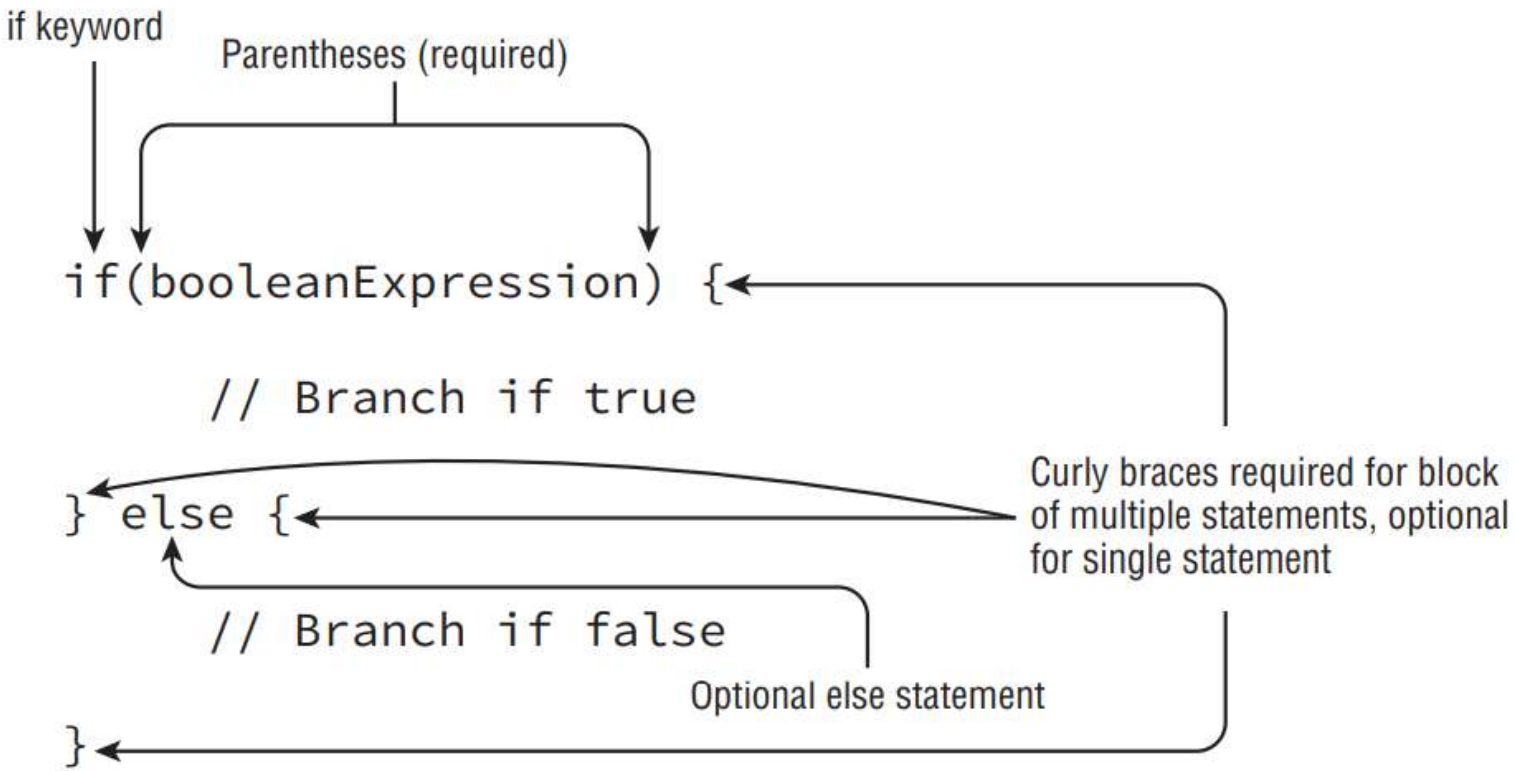
$= 14 + 2$

$= 16$

Therefore, $14 + (8 - 2 \times 3) = 16$.




```
if(hourOfDay < 11)
      System.out.println("Good Morning");
```




```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else {
  System.out.println("Good Afternoon");
}
```

Find out eligibility to vote using 'if'

Problem

Given the age of an Indian national, find out if the person is eligible to vote or not. In India, a person who is not less than 18 years has the eligibility to vote. Get the age input using Scanner.

Sample output when age less than 18 years

```
Enter the age
10
Cannot Vote
```

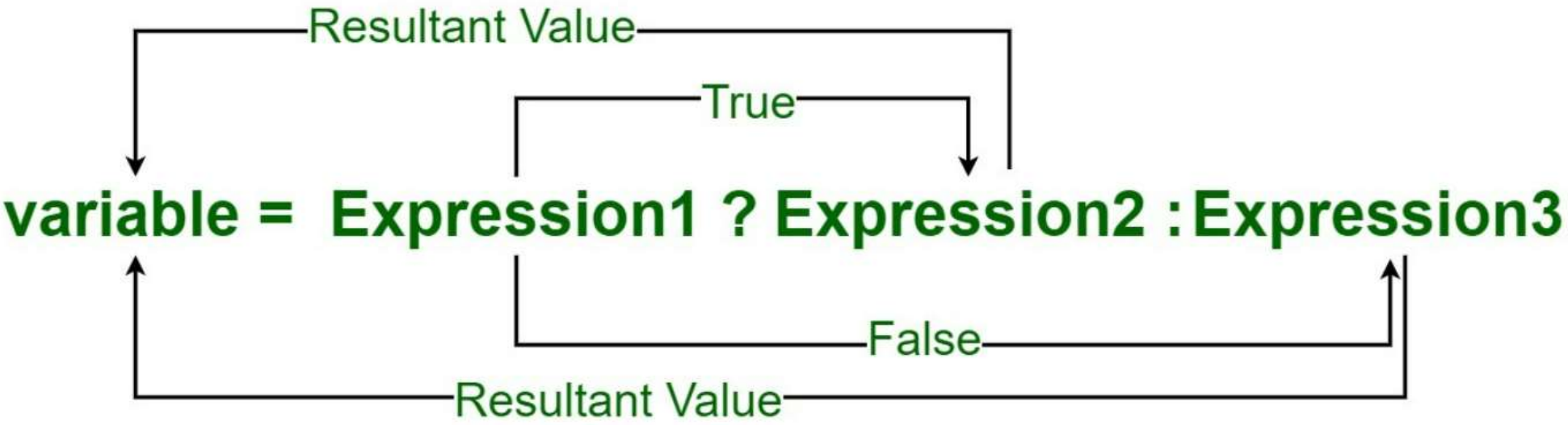
Sample output when age is 18 years

```
Enter the age
18
Can Vote
```

Sample output when age is not less than 18 years

```
Enter the age
25
Can Vote
```

Conditional or Ternary Operator (?:) in Java



1st Condition is true

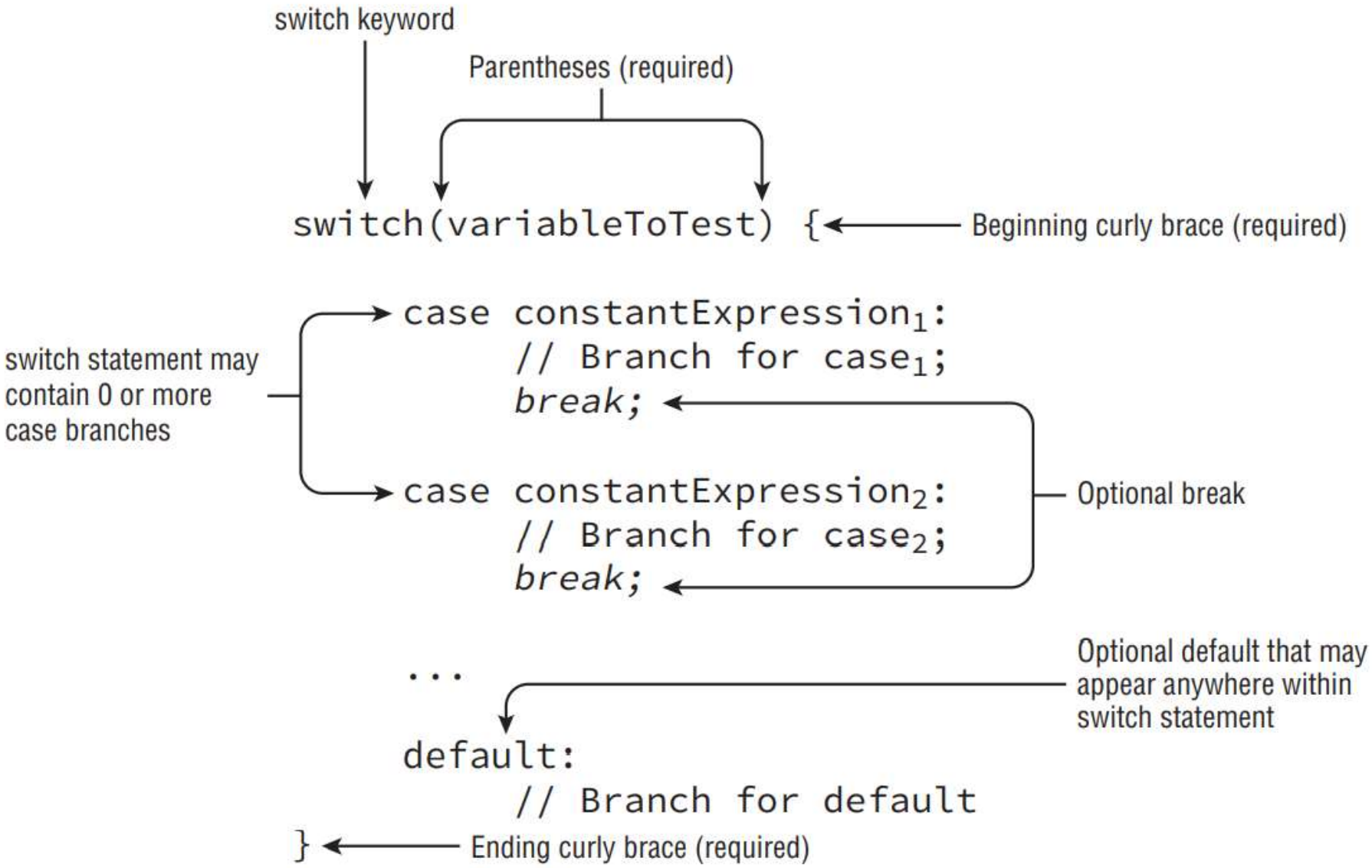
```
int number = 2;
if (number > 0) {
  // code
}
else if (number == 0){
  // code
}
else {
  //code
}

//code after if
```

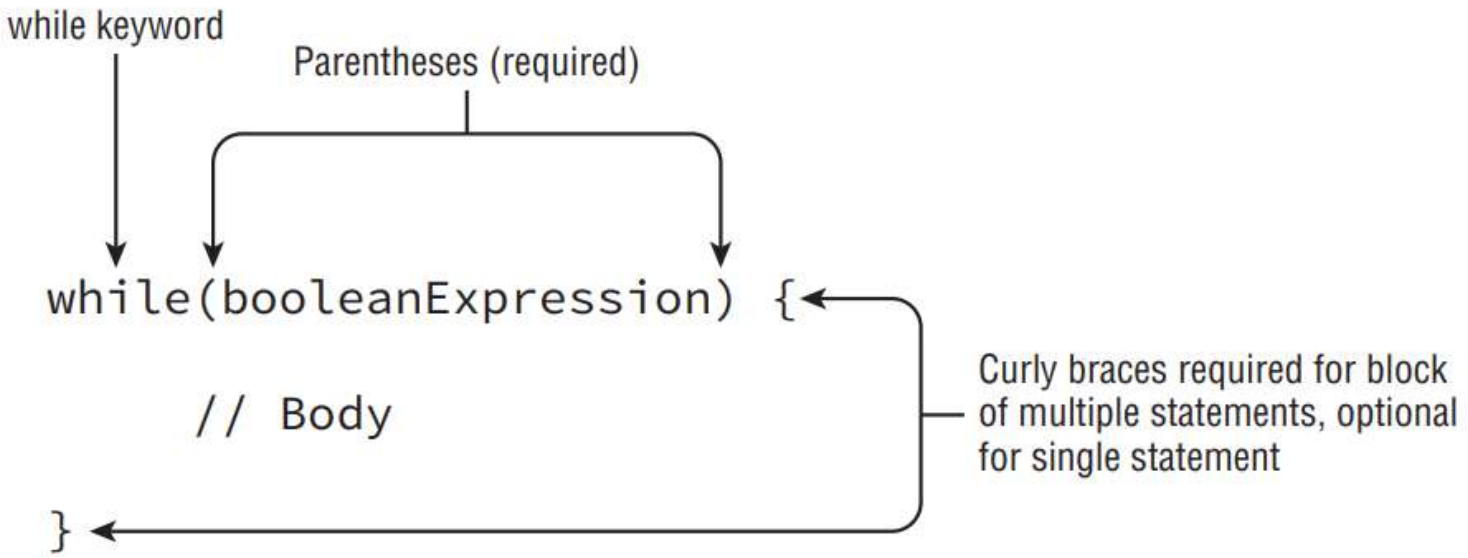
2nd Condition is true

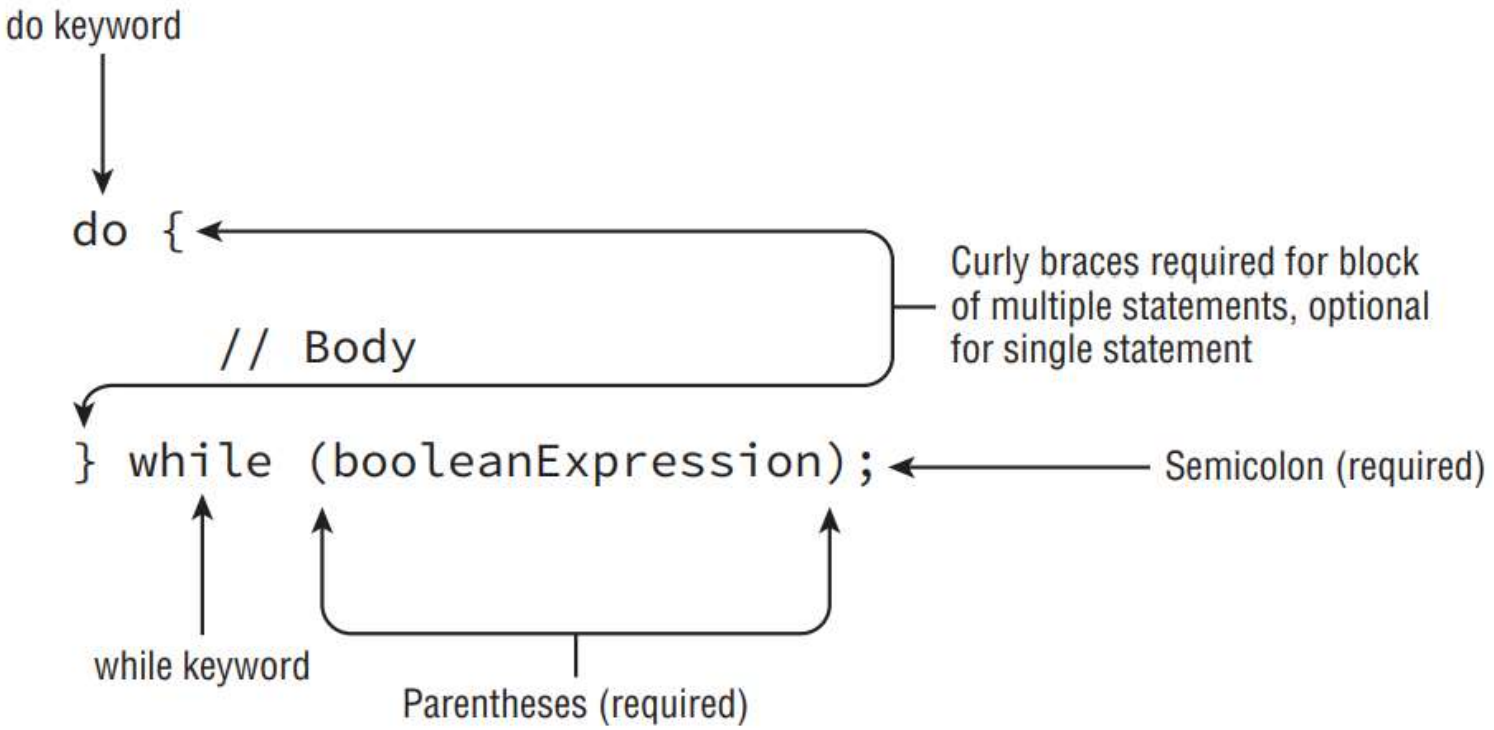
```
int number = 0;
if (number > 0) {
  // code
}
else if (number == 0){
  // code
}
else {
  //code
}

//code after if
```

All Conditions are false

```
int number = -2;
if (number > 0) {
  // code
}
else if (number == 0){
  // code
}
else {
  //code
}

//code after if
```


switch keyword
Parentheses (required)
switch(variableToTest) {
Beginning curly brace (required)
switch statement may contain 0 or more case branches
case constantExpression1:
// Branch for case1;
break;
case constantExpression2:
// Branch for case2;
break;
Optional break
...
Optional default that may appear anywhere within switch statement
default:
// Branch for default
}
Ending curly brace (required)


while keyword
Parentheses (required)
while(booleanExpression) {
// Body
}
Curly braces required for block of multiple statements, optional for single statement


do keyword
do {
// Body
Curly braces required for block of multiple statements, optional for single statement
} while (booleanExpression);
Semicolon (required)
while keyword
Parentheses (required)

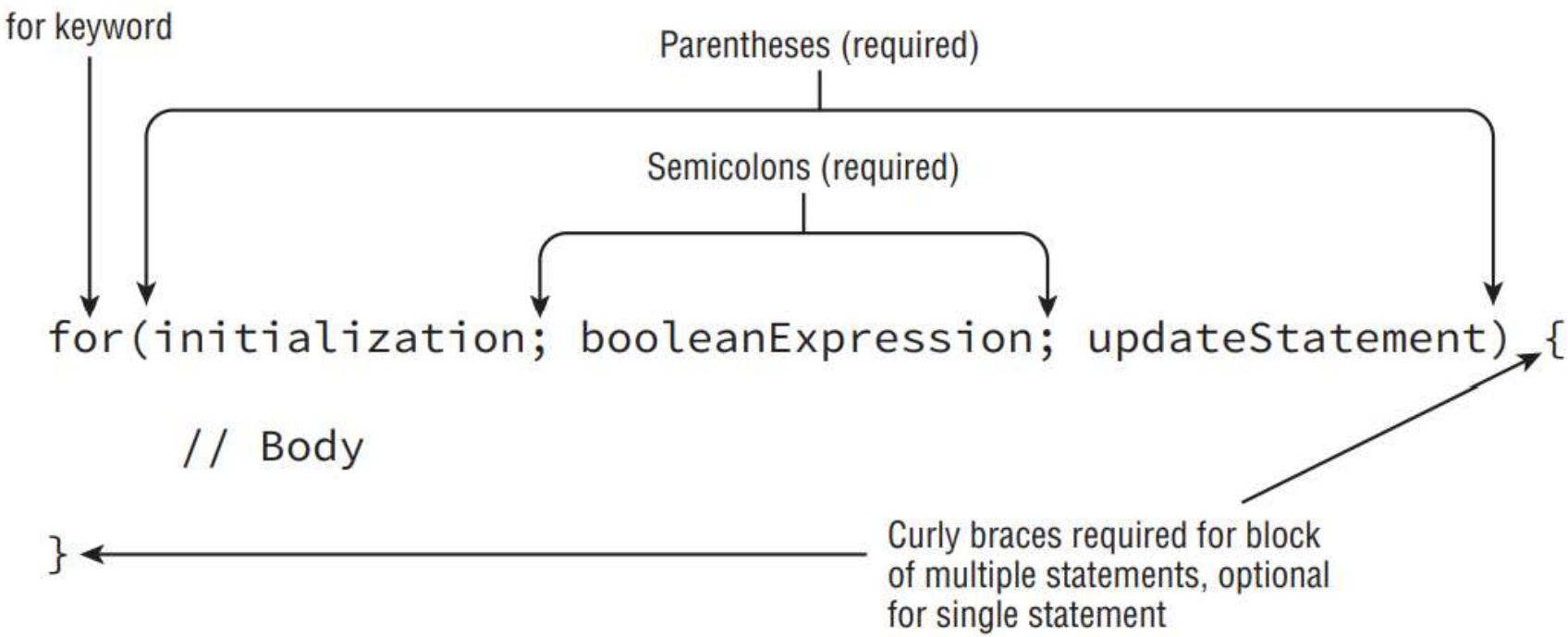
Differentiate between "while" and "do while" statements

while statement

1. It is an entry controlled loop.
2. Initially test condition is evaluated. If it is true body of the loop is executed.
3. If test condition is initially false the body of loop is not executed at all.

do while statement

1. It is an exit controlled loop.
2. Initially body of loop is executed and then test condition is evaluated.
3. The body of the loop will executed at least once even if the test condition is false.

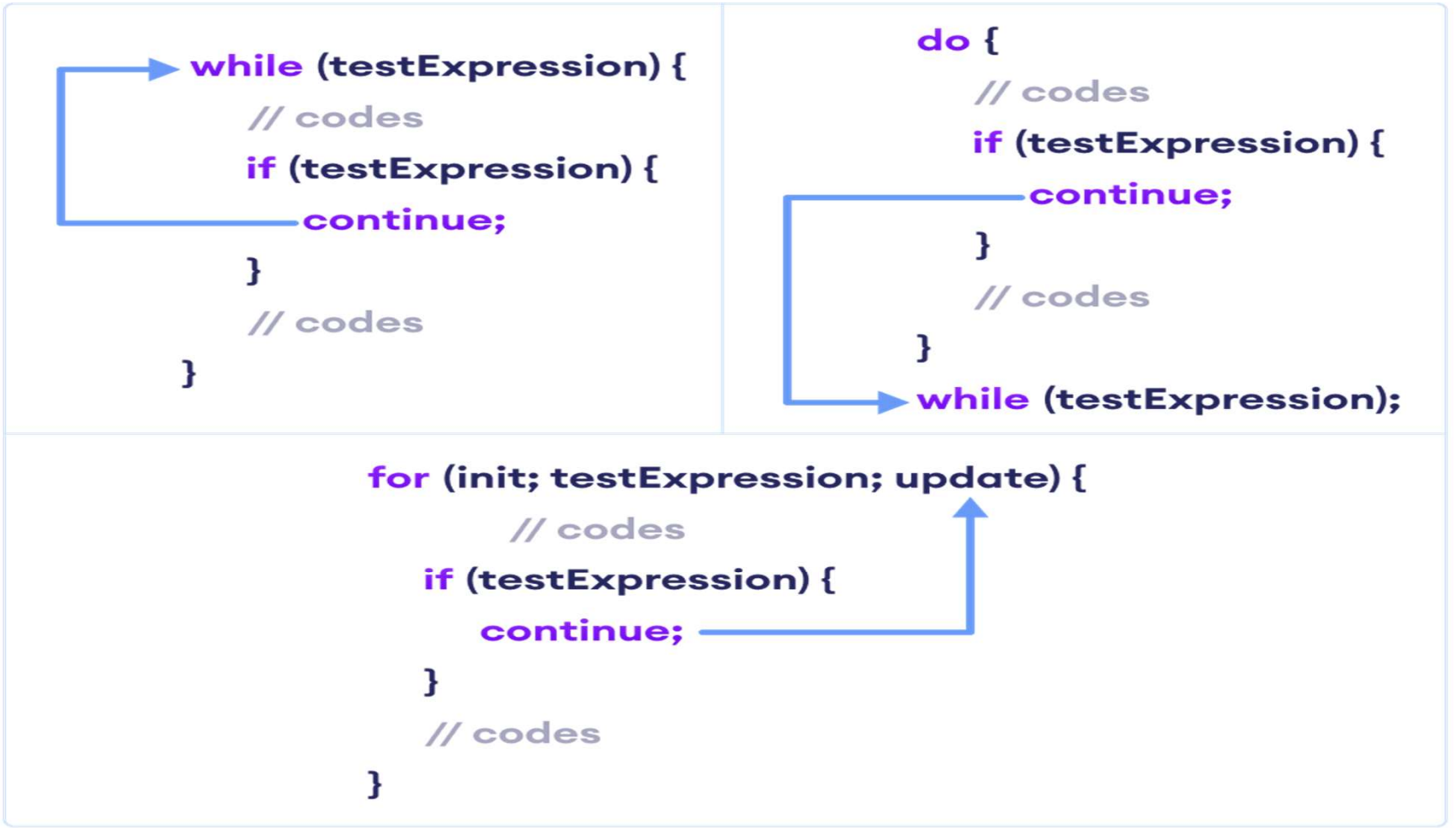


1. Initialization statement executes
2. If booleanExpression is true continue, else exit loop
3. Body executes
4. Execute updateStatements
5. Return to Step 2

```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```


while (testExpression) {
// codes
if (testExpression) {
continue;
}
// codes
}
do {
// codes
if (testExpression) {
continue;
}
// codes
}
while (testExpression);
for (init; testExpression; update) {
// codes
if (testExpression) {
continue;
}
// codes
}

BREAK	CONTINUE
A loop control structure that causes the loop to terminate and pass the program control to the next statement following the loop	A loop control structure that causes the loop to jump to the next iteration of the loop immediately
Helps to terminate the execution of the loop	Helps to skip statements inside the loop

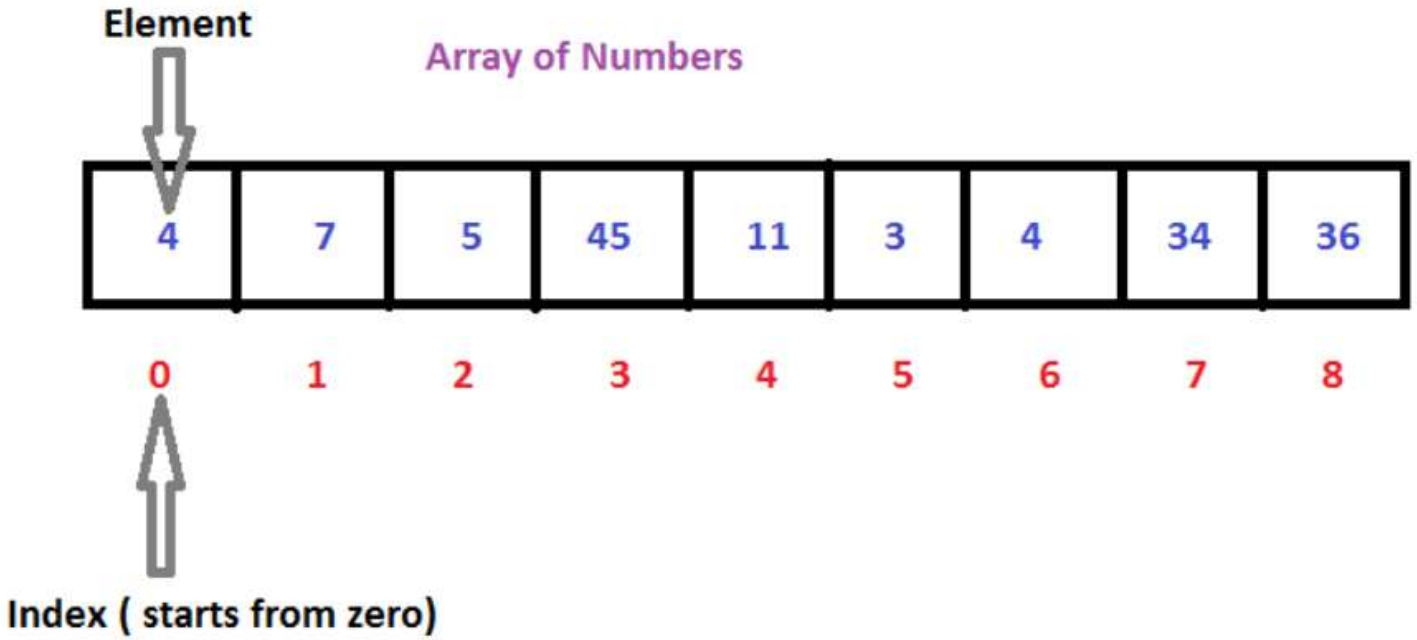
Arrays

- *Same kind of values/Homogenous elements*
- *Continuous stored in the memory*
- *Size of the array is fixed*
- *Index always starts from 0*
- *For example*

```
int array[]; //declaring array
array = new int[9]; // allocating memory to array
```

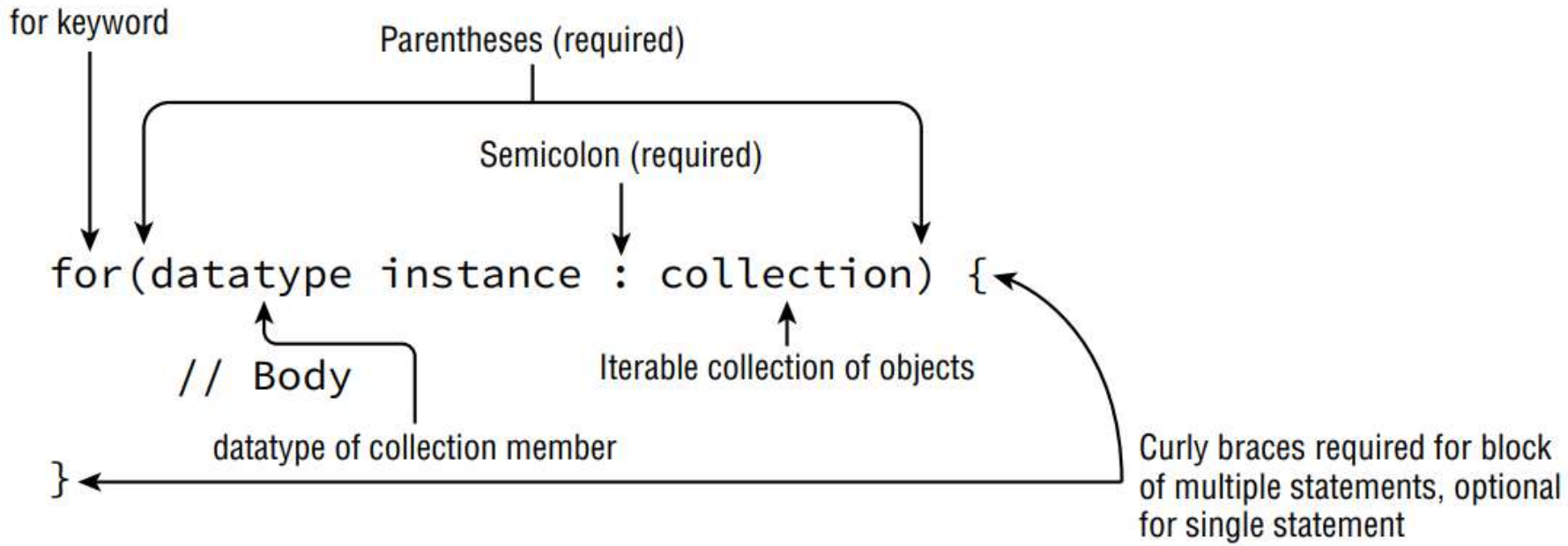
OR

```
int[] array = new int[9]; // combining both statements in one
```



```java
import java.util.*;

public class Array {
    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter n: ");
        int n = sc.nextInt();
        int[] a = new int[n];
        System.out.println("Enter values: ");
        for (int index = 0; index < n; index++) {
            a[index] = sc.nextInt();
        }
        System.out.println("Printing values values: ");
        for (int index = 0; index < n; index++) {
            System.out.println(a[index]);
        }
    }
}
```

for each


for keyword
Parentheses (required)
Semicolon (required)
for(datatype instance : collection) {
// Body
Iterable collection of objects
datatype of collection member
}
Curly braces required for block of multiple statements, optional for single statement

```
int arrSq[]={1,4,9,16,25};
```

for loop

```
for(int i=0; i<=4 ; i++)
{
 System.out.println(arrSq[i]);
}
```

for-each loop

```
for( int i : arrSq )
{
 System.out.println(i);
}
```

Arrays Exercise

1. *Get the size of the array from the user*
2. *Get the array values from the user*
3. *Print all the elements in the array*
4. *Print the even numbers in the arrays*
5. *Print the odd numbers in the array*
6. *Print the numbers which is greater than 50*
7. *Print the numbers between 51 to 100*
8. *Find out the total sum of the array*
9. *Find out the avg of the array number*
10. *Print the numbers in the even index of the array*
11. *Print the numbers in the odd index of the array*
12. *Print the max and min number in the array*

2D Arrays



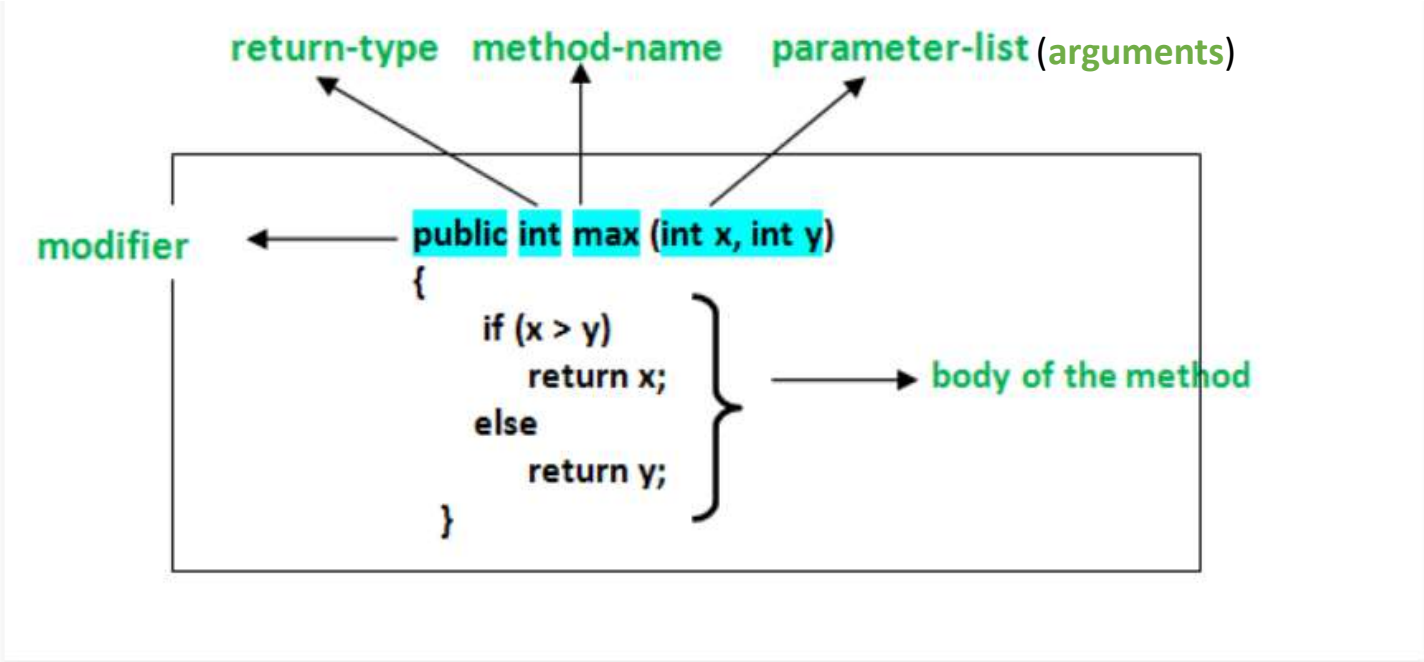
- Organized as matrices which can be represented as the collection of rows and columns.
- **Syntax:** <data type>[][] <arrayname> = new <datatype>[row_size][cols_size];
- **Example:** int [][] a = new int [3][4];

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```java
import java.util.*;
public class Array2d
{
    Run | Debug
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter rows: ");
        int rows = sc.nextInt();
        System.out.println("Enter cols: ");
        int cols = sc.nextInt();
        int[][] matrix = new int[rows][cols];
        System.out.println("Enter " + rows * cols + " values: ");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }
        System.out.println("Printing values");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Method

- *collection of statements that perform some specific task and with/without returning anything*



String Class and its methods

String s = “animals”;

a	n	i	m	a	l	s
0	1	2	3	4	5	6

String s = sc.next();

Capable of reading user inputs until receiving a space

String s = sc.nextLine();

Capable of reading user inputs until pressing the enter key or receiving a new line

STRING LITERAL	STRING OBJECT
Set of characters that is created by enclosing them inside a pair of double quotes	Set of characters that is created using the new() operator
If the String already exists, the new reference variable will be pointing to the already existing literal	Even the String already exists or not, a new String object will be created
String s = "Hello World";	String s = new String ("Hello World!");

Points to be Noted:

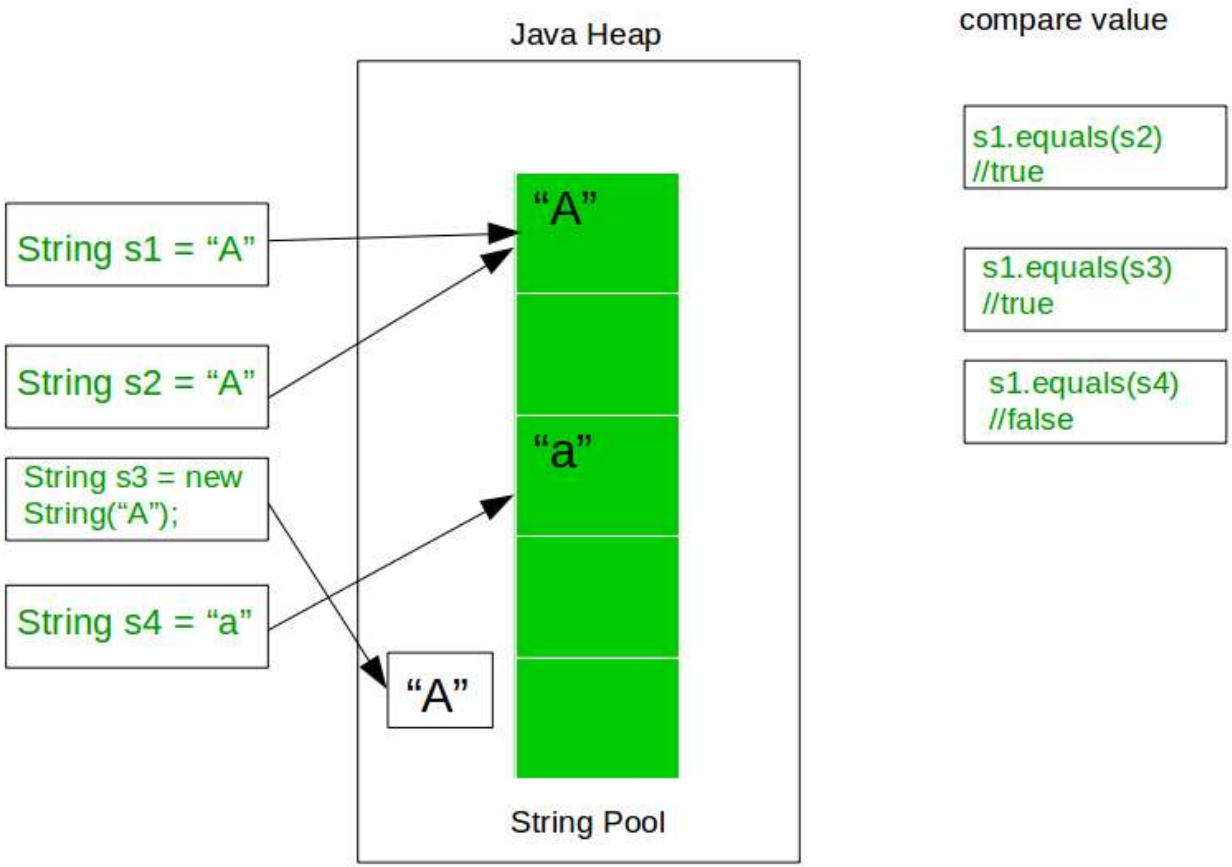
- *String Pool*
  - *It is a **Pool of Strings** stored in the **Heap Memory**.*
  - *All the strings created are stored in this pool if they do not already exist in the pool.*

- *String class is **immutable** class and hence all string objects created using **literals or new operators** cannot be changed/modified.*

- **String name1 = "Java";**
  - *The above statement creates a string object and places it in a string pool if it is not already present in the string pool and a reference is assigned to name1.*

- **String name2 = new String("newJava");**
  - *The above statement creates a string object in heap memory and checks whether it is present in the string pool or not.*
  - *If the **'newJava'** is not present in the string pool then it will place this string in the string pool else it will skip it.*
  - *In this case, two objects are created that is one in heap memory and the other in the string pool.*

.equals()



compare value

s1.equals(s2)
//true

s1.equals(s3)
//true

s1.equals(s4)
//false

1. If both operands are numeric, + means numeric addition.
2. If either operand is a `String`, + means concatenation.
3. The expression is evaluated left to right.

Now let's look at some examples:

```
System.out.println(1 + 2);              // 3
System.out.println("a" + "b");          // ab
System.out.println("a" + "b" + 3);      // ab3
System.out.println(1 + 2 + "c");        // 3c
```

Random Class in Java

```
import java.util.Random;  // import tells us where to find Random
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10));  // print a number between 0 and 9
   }
}
```

Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

LUCKY DRAW

- Input – String
- Get a position - using Random Class, to pick a number within the input string Length
- Find out that character in that position (randomly picked)
- If that character is vowel, then tell the user “Congrats, you won Lucky draw”
- If that character is not a vowel, tell the user “Better Luck next time”

Aim

- *To enable your logical thinking ability*
- *To make yourself good in Java Syntax*
- *To impose the java programming ability*
- *To improve your problem-solving capability*

1. Write a program to swap two numbers and check which is greater

Enter a: **35**
Enter b: **25**
Before swapping the values of a and b are 35 and 25
After swapping the values of a and b are 25 and 35
b is greater

2. Write a program to reverse the given string.

Enter string: **Hello**
Reversed string: olleH

3. Write a program to reverse the right half of the given string.

Enter string: **Hello**
Reversed right-half string: Heoll

Enter string: **Good**
Reversed right-half: Godo

Enter string: **bad**
Reversed right-half: bda

4. Write a program to remove all the vowels in the given input string

Enter string: **Hello**
String without vowels: Hll

Enter string: **Goat**
String without vowels: **Gt**

5. Write a program to check the given string is palindrome or not

Enter string: **hello**
hello is not palindrome

Enter string: **madam**
madam is not palindrome

6. Write a program to check the given string to find whether a given string contains a number.

Enter string: **abc123**
Yes, abc123 contains number

Enter string: **abc**
No, abc doesn’t contains number

7. Write a program to check the middle character of two given strings with same length are same

Enter string 1 : **Goat**
Enter string 2 : **Roar**
Middle character are same. Mid character = a

Enter string 1: **Hello**
Enter string 2: **all23**
Middle character are same. Mid character = l

Enter string 1: Hello
Enter string 2: World
Middle character are different.

8. Write a program to check all the characters in the given string is in lowercase or not.

Enter string: **hello**
In hello, all characters are in lowercase

Enter string: **ManGO**
In ManGO, all characters are not in lowercase

Enter string: wel123
In wel123, all characters are not in lowercase

9. Write a program to count the digits in the given number

Enter number: **1232**
4 digits

Enter number: **10**
2 digits

Enter number: **345**
3 digits

10. Write a program to reverse the given number

Enter number: **1232**
Reversed number: 2321

Enter number: **10**
Reversed number: 01

Enter number: **345**
Reversed number: 543

Enter number: **121**
Reversed number: 121

11. Write a program to check the given number is prime or not

Enter number: **5**
5 is a prime number

Enter number: **10**
10 is not a prime number

Enter number: **17**
17 is a prime number

12. Write a program to check the given number is palindrome or not

Enter number: **1232**
1232 is not palindrome

Enter number: **11**
11 is palindrome

Enter number: **345**
345 is not a palindrome

Enter number: **121**
121 is a palindrome

13. Write a program to print factorial of a given number

Enter number: **5**
120

Enter number: **4**
24

Enter number: **7**
5040

14. Write a program to print Fibonacci series up to n

Enter n: **5**
1 1 2 3 5

Enter number: **6**
1 1 2 3 5 8

Enter number: **3**
1 1 2

13. Write a program to print factorial of a given number

Enter number: **5**
120

Enter number: **4**
24

Enter number: **7**
5040

14. Write a program to print Fibonacci series up to n

Enter n: **5**
1 1 2 3 5

Enter number: **6**
1 1 2 3 5 8

Enter number: **3**
1 1 2

Object-Oriented Programming

- *All about creating **objects***

- *Objects – group of variables and functions*
  - *Variables/Features/Properties/State/data members/attributes*
  - *Functions/Methods/Behavior/Action Performed/Member functions*

- *For Example*
  - *Car*
    - *its properties would be – its color, its model, its price, its brand, etc.*
    - *its behavior/function would be acceleration, slowing down, gear change*

  - *Dog*
    - *its properties would be- his color, his breed, his name, his weight, etc.*
    - *And his behavior/function would be walking, barking, playing, etc.*

4 Pillars – a pie

- ***A**bstraction*
  - *Hiding the implementation and showing only the necessary details to the user*

- ***P**olymorphism*
  - *An object exhibiting the different behaviors*

- ***I**nheritance*
  - *is a relationship*
  - *Inheriting the properties and behaviors form parent*

- ***E**ncapsulation*
  - *Encapsulating/binding/wrapping properties and behavior together into a single unit*

Class

- *In the real world, you often have many objects of the same kind.*
- *For example, your car is just one of many cars in the world.*
- *Using object-oriented terminology, we say that your car is an instance of the class Cars.*
- *A class is the blueprint/prototype from which individual objects are created.*

```
class Car{
    String brand;
    String color;
    long model;

    void accleration(){
      System.out.println("acclerating..");
    }

    void applyBrake(){
      System.out.println("applying brake..");
    }

    void gearChange(){
      System.out.println("changing gear");
    }
}
```

Object

- *Real world entities*
- *Instance of a class*
- *To create an object for any class, we specify the class name, followed by object name and we use new operator and constructor*

```
Car myCar = new Car();
```

Access Modifiers

- *public*
  - *Accessible Everywhere*
  - *can be accessed from within the class, outside the class, within the package and outside the package*

- *private*
  - *Accessible only inside the class*

- *protected*
  - *Accessible within that package and subclasses in the other packages*

- *default*
  - *Accessible only within that package*

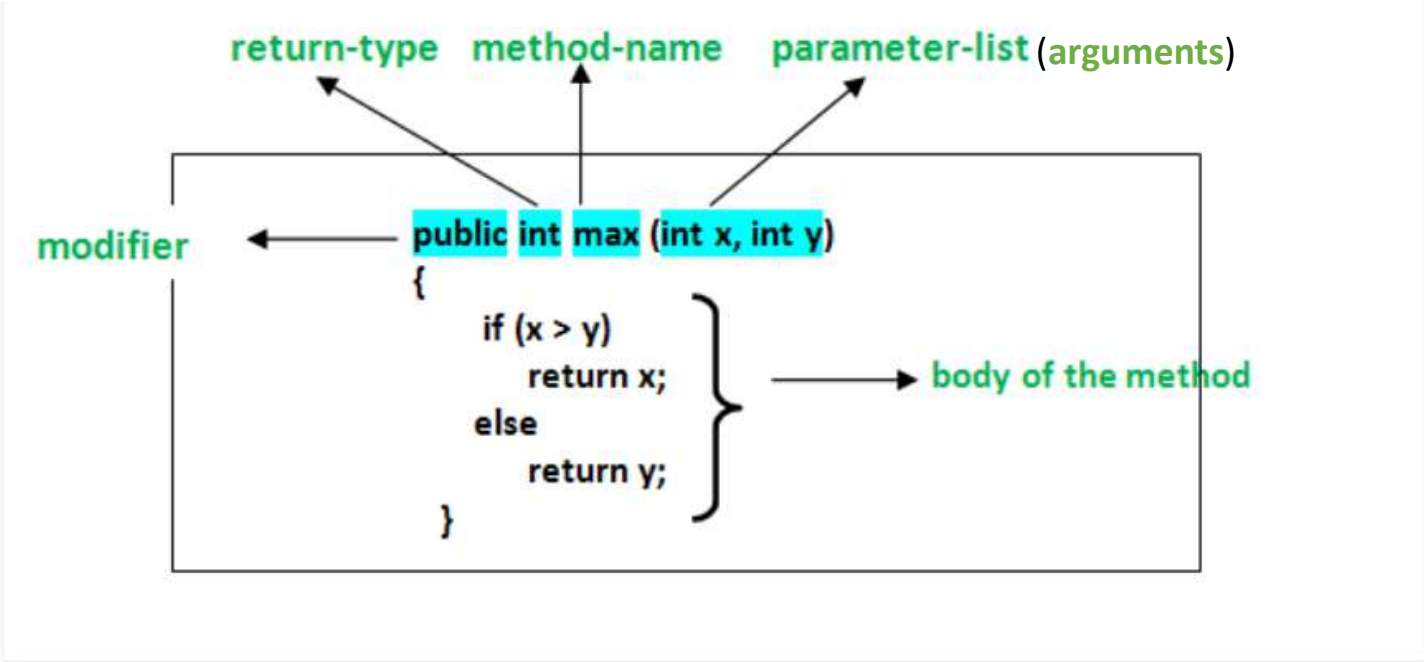
static

- *Non access modifier*
- *used to share the same variable or method of a given class*
- *When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object*

```
static int count;

public static void display() {
    System.out.println("Static method");

}
```

Recall - Method

- *collection of statements that perform some specific task and with/without returning anything*



Constructor

- *Similar to method, called when an instance of the class (or object) is created.*
- *Used to initialize the values*
- *When we define a class only the blueprint of the object is created. There is no memory allocated. Memory allocated for an object, at the time constructor is called*
- *Rules:*
  - *Same name as class name*
  - *Doesn't have return type*
  - *Can have no/one/more arguments/parameters*
  - *Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.*

Types of Constructor

1. ***No argument constructor or default constructor***

   - *If we don't define a constructor in a class, then the compiler creates a default constructor(with no arguments) for the class.*
   - *if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.*
   - *Provides the default values to the object like 0, null, etc. depending on the type.*

2. ***Parameterized Constructor***

   - *A constructor that has parameters is known as parameterized constructor.*
   - *If we want to initialize fields of the class with our own values, then use a parameterized constructor.*

```java
class Car{
    String brand;
    String color;
    long model;

    Car(){
        this.brand = "";
        this.color = "";
        this.model = 0;
    }

    Car(String brand){
        this.brand = brand;
    }

    Car(String brand, String color){
        this.brand = brand;
        this.color = color;
    }

    Car(String brand, String color, long model){
        this.brand = brand;
        this.color = color;
        this.model = model;
    }
```

```java
Car myCar1 = new Car();
Car myCar2 = new Car("Audi");
Car myCar3 = new Car("Toyota", "red");
Car myCar4 = new Car("Maruti Suzuki", "blue", 12345);
```

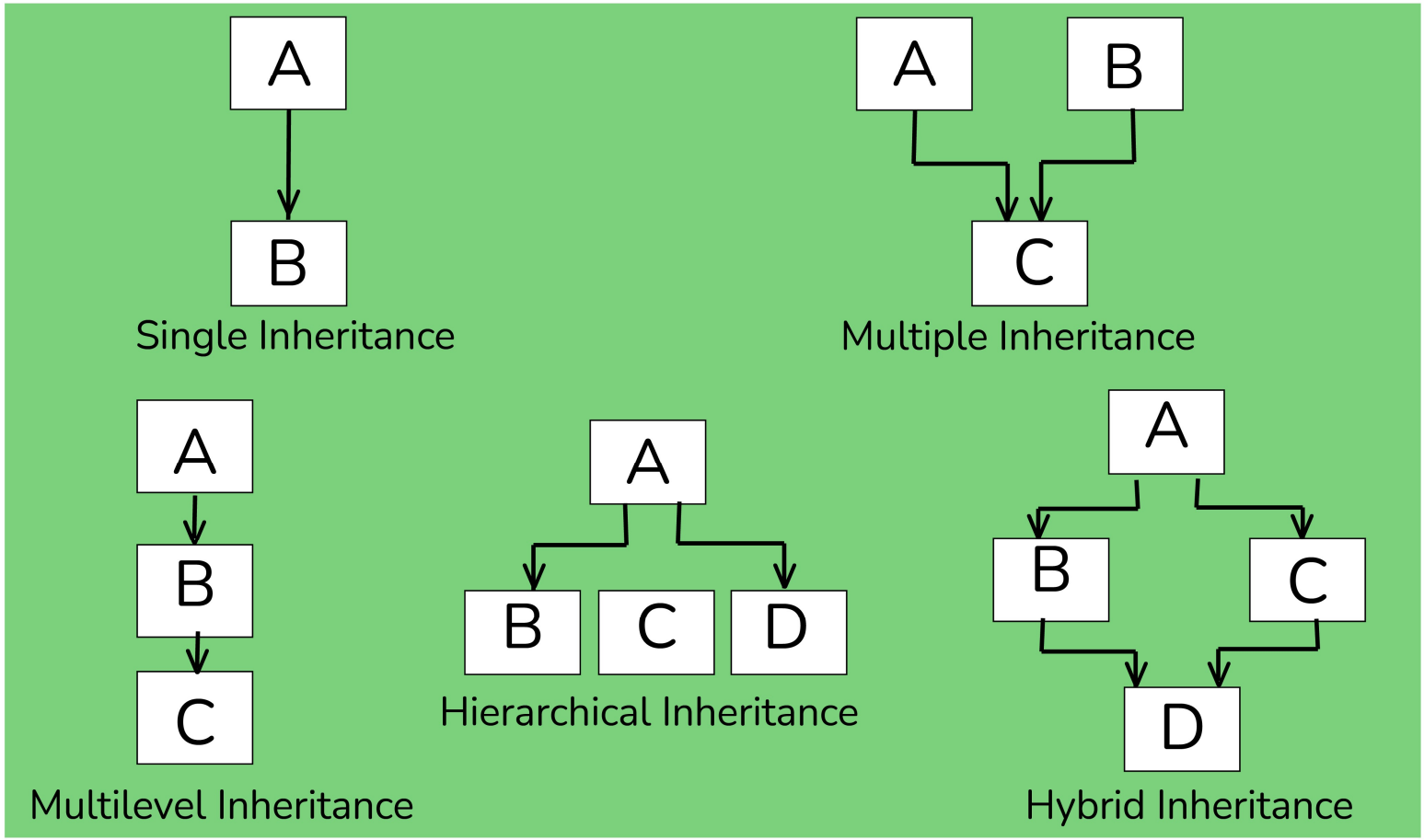
Exercise:

- *Create a class named Student with String variable name and integer variables roll_no*
- *Create a no argument constructor that initializes name and roll_no as null and 0 respectively*
- *Create a parameterized constructor, which updates the value upon object creation*
- *Create a static variable count to keep track of the object created for a class Student*
- *Create a display method which displays the name and roll_no for an object*
- *Create two objects using no argument constructor*
- *Create two objects using parameterized constructor*
- *Print the values for all the objects*

Inheritance

- *It is the procedure in which one class inherits the attributes and methods of another class.*
- *The class whose properties and methods are inherited is known as Parent class/Super class. And the class that inherits the properties from the parent class is the Child class/Sub class.*
- *Along with the inherited properties and methods, a child class can have its own properties and methods.*
- *extends keyword*
- *We can extend only class in java*
- *Types – Single, Multiple, Multilevel, Hybrid and Hierarchical*
- *Multiple Inheritance is not possible in java.. Because its results ambiguity.. But it can be achieved using interfaces*

Types Of Inheritance



instanceOf Keyword - used for checking if a reference variable is containing a given type of object reference or not.

super keyword - used to refer parent class methods, constructors and variables

```java
/* Using super with Methods*/
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}
/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }
    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}
```

```java
/**Using super with Constructors */
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}
/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();
        System.out.println("Student class Constructor");
    }
}
```

```java
/*Using super with variable*/
/* Base class vehicle */
class Vehicle
{
    int maxSpeed = 120;
}
/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;
    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}
```

Encapsulation

- *is a way to ensure security*

- *Basically, it hides the data from the access of outsiders. Such as if an organization wants to protect an object/information from unwanted access by clients or any unauthorized person then encapsulation is the way to ensure this.*

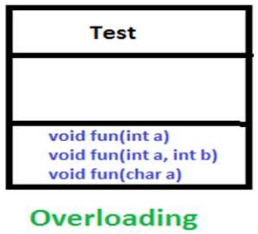
- *wrapping/binding/encapsulating the data (variables) and code acting on the data (methods) together as a single unit.*

- *In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.*

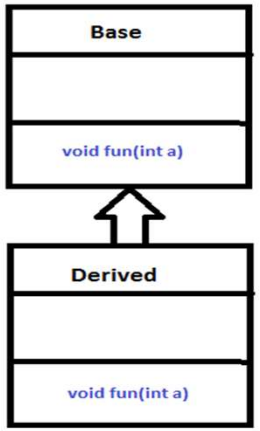
- *To achieve encapsulation in Java*
  - *Declare the variables of a class as **private**.*
  - *Provide **public setter and getter** methods to modify and view the variables values.*

Polymorphism

- *having many forms*
- *A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a son, a husband, an employee, a doctor. So, the same person posse's different behavior in different situations. This is called polymorphism.*
- *Types:*
  - *Compile time polymorphism/Method Overloading - When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.*
  - *Runtime polymorphism/Method Overriding - During inheritance in Java, if the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding.*



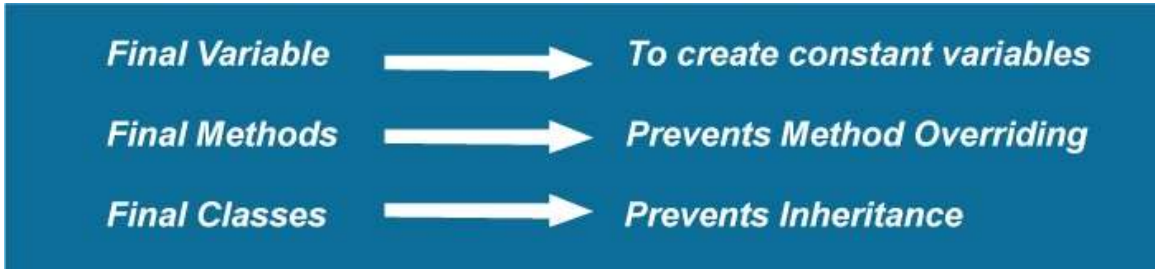
Overloading



Overriding

final

- *Non access modifiers*
- *Java final keyword is a non-access specifier that is used to restrict a class, variable, and method.*
- *If we initialize a variable with the final keyword, then we cannot modify its value.*
- *If we declare a method as final, then it cannot be overridden by any subclasses.*
- *And, if we declare a class as final, we restrict the other classes to inherit or extend it.*



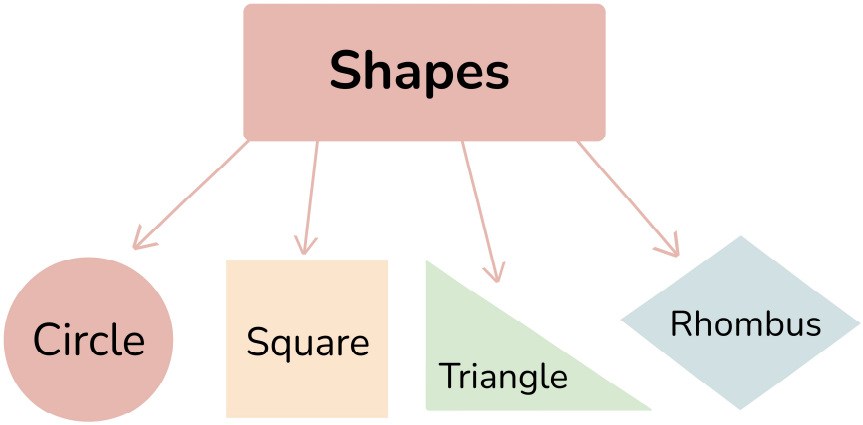
Abstraction

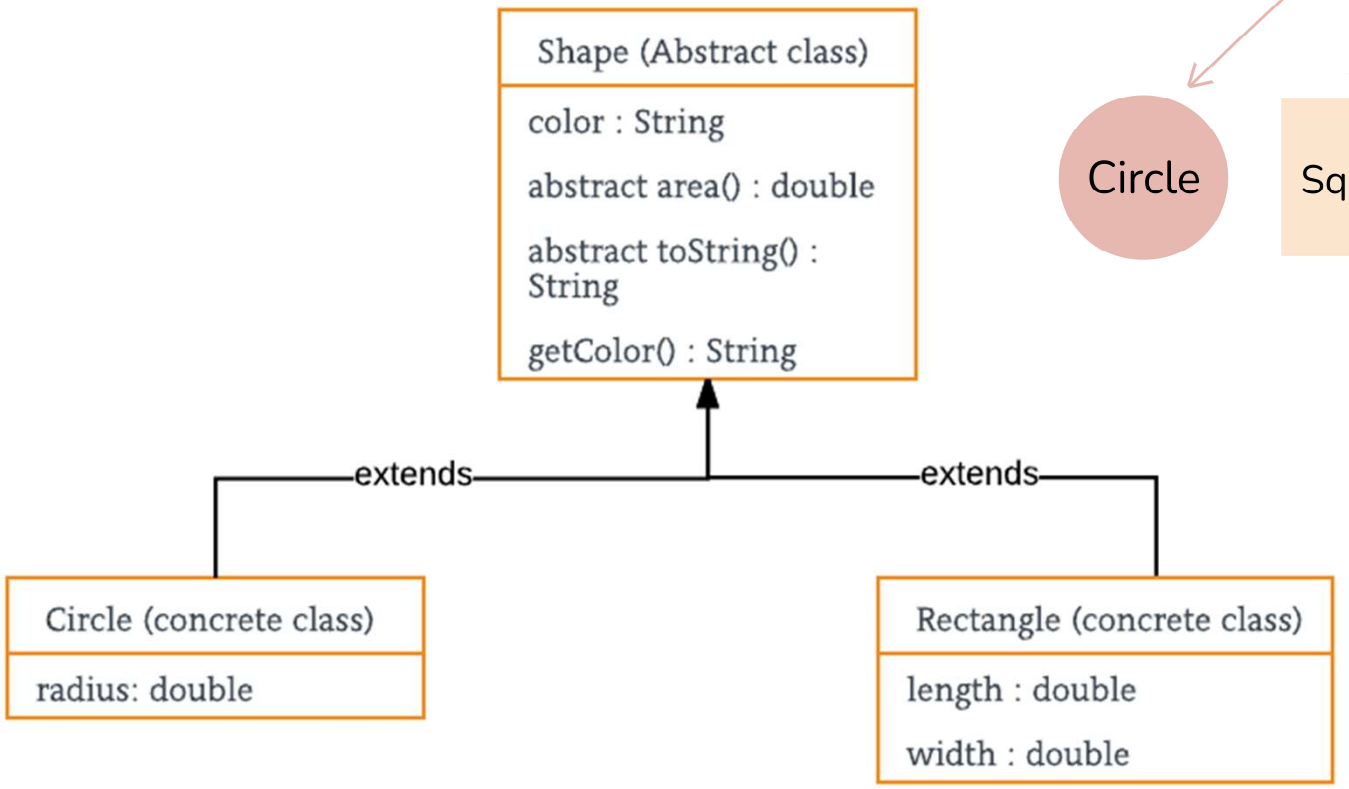
- *Hiding the Implementation and showing only the necessary details to the user*

- *This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch, but you don't know what is happening behind the socket.*

- *In Java, we can achieve abstraction using*
  - *Abstract classes – 0 to 100% abstraction*
  - *Interfaces – 100% abstraction*

- ***Abstract Classes***
  - *It is a class that is declared with an abstract keyword.*
  - *It has abstract methods and concrete methods (non abstract)*
  - *Abstract method is a method that is declared without implementation*
  - *A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR make the subclass itself abstract.*
  - *We can't create an object for abstract class, but we can use it as object reference*


Shapes
Circle
Square
Triangle
Rhombus


Shape (Abstract class)
color : String
abstract area() : double
abstract toString() : String
getColor() : String
extends
extends
Circle (concrete class)
radius: double
Rectangle (concrete class)
length : double
width : double

Interfaces

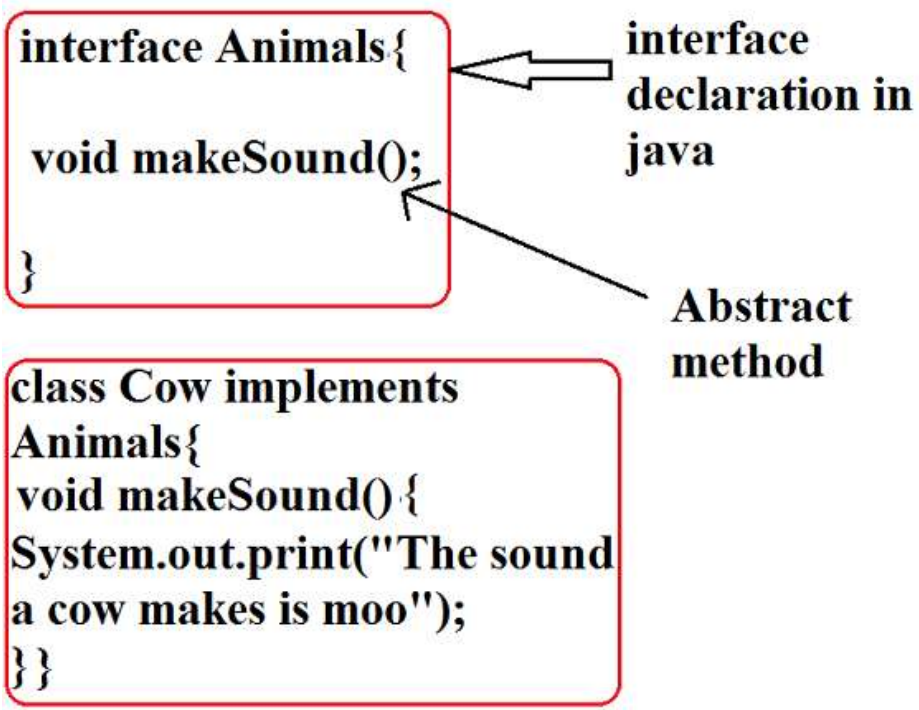
Why And When To Use Interfaces?

1. *To achieve security - hide certain details and only show the important details of an object (interface).*
2. *Java does not support **multiple inheritance** (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.*

Note:

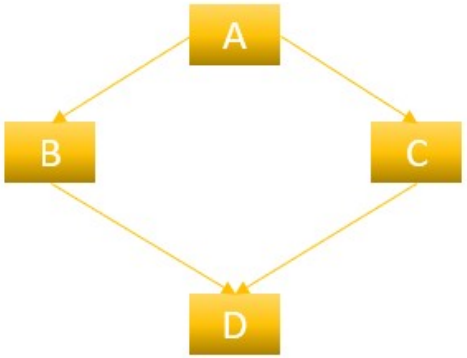
- *Like abstract classes, interfaces **cannot** be used to create objects*
- *Interface methods **do not have a body** - the body is provided by the "implement" class*
- *On implementation of an interface, you must **override** all of its methods*
- *Interface methods are by default **abstract** and **public***
- *Interface attributes are by **default public**, **static** and **final***
- *An interface **cannot** contain **a constructor** (as it cannot be used to create objects)*

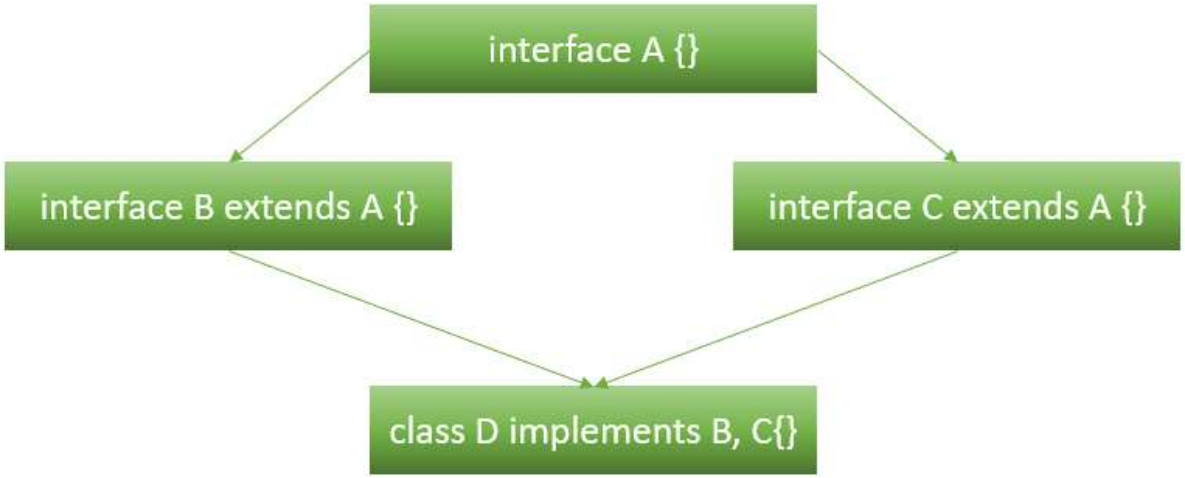
```
interface A {
        void method();
        default method(){
            //code
        }
}
```


interface Animals{
void makeSound();
}
interface declaration in java
Abstract method
class Cow implements Animals{
void makeSound(){
System.out.print("The sound a cow makes is moo");
}}
class Dog implements Animals{
void makeSound(){
System.out.println("The sound a dog makes is woof woof!! ");
}
}
class Cat implements Animals {
void makeSound(){
System.out.print("The sound a cat makes is meow");
}
}

Diamond Problem

- *Related to the multiple inheritance*
- *Solved using Interfaces*





Object Class

- *Present in java.lang package*
- *Every class in Java is directly or indirectly derived from the **Object** class.*
  - *If a class does not extend any other class, then it is a direct child class of **Object** and if extends another class then it is indirectly derived.*
- *Therefore, the Object class methods are available to all Java classes.*
- *Hence Object class acts as a root of inheritance hierarchy in any Java Program.*

Methods

- *getClass() - returns the class of "this" object*

- *equals() - It compares the given object to "this" object (the object on which the method is called).*
  - *It is recommended to override the **equals(Object obj)** method to get our own equality condition on Objects.*

- *clone() - returns a new object that is exactly the same as this object.*

- *finalize() - called by the garbage collector on an object **when garbage collection determines that there are no more references to the object**.*
  - *A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.*

- *hashCode() - For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects*
  - *A common **misconception** about this method is that the hashCode() method **returns the address of the object, which is not correct**.*
    - *It converts the internal address of the object to an integer by using an algorithm.*
    - *The hashCode() method is **native** because in Java it is impossible to find the address of an object, so it uses native languages like C/C++ to find the address of the object.*
  - ***Use of hashCode()***
    - *It returns a hash value that is used to search objects in a collection.*
    - *JVMuses the hashcode method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc.*
    - *The main advantage of **saving objects based on hash code is that searching becomes easy***

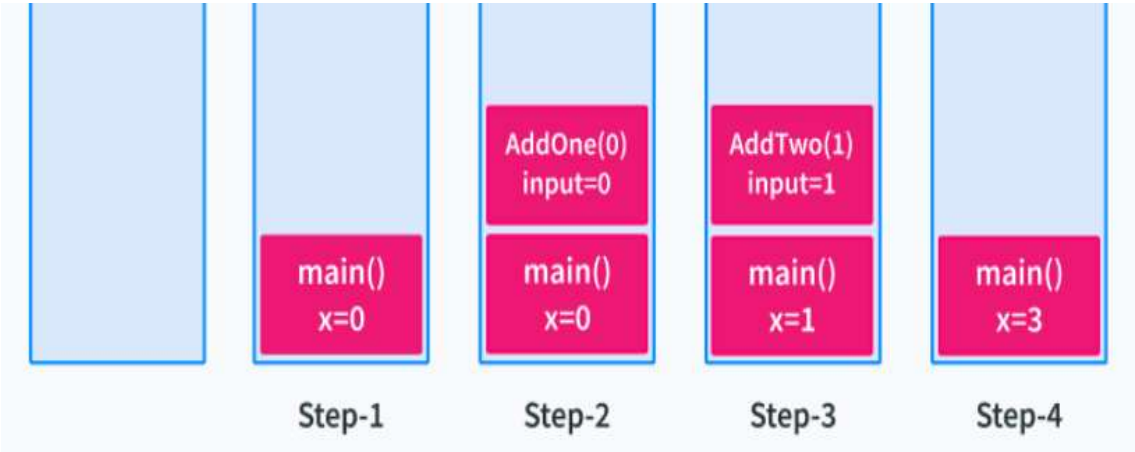
- *toString() - provides a String representation of an object and is used to convert an object to String*

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object

public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Stack Memory

- *Space allocate for all the function calls, primitive data types like int, double, etc., and local and reference variables of the functions are stored*
- *Always accessed in a Last-In-First-Out (LIFO) manner*
- *In the stack memory, a new memory block is created for every method that is executed.*
  - *All the primitive variables and references to objects inside the method are stored in this memory block.*
  - *When the method completes its execution, the memory block is cleared from the stack memory and the stack memory is available for use.*

```java
public class Main {
  public static int addOne(int input) {
    return input + 1;
  }

  public static int addTwo(int input) {
    return input + 2;
  }

  public static void main(String[] args) {
    int x = 0;

    x = addOne(x);
    x = addTwo(x);
  }
}
```

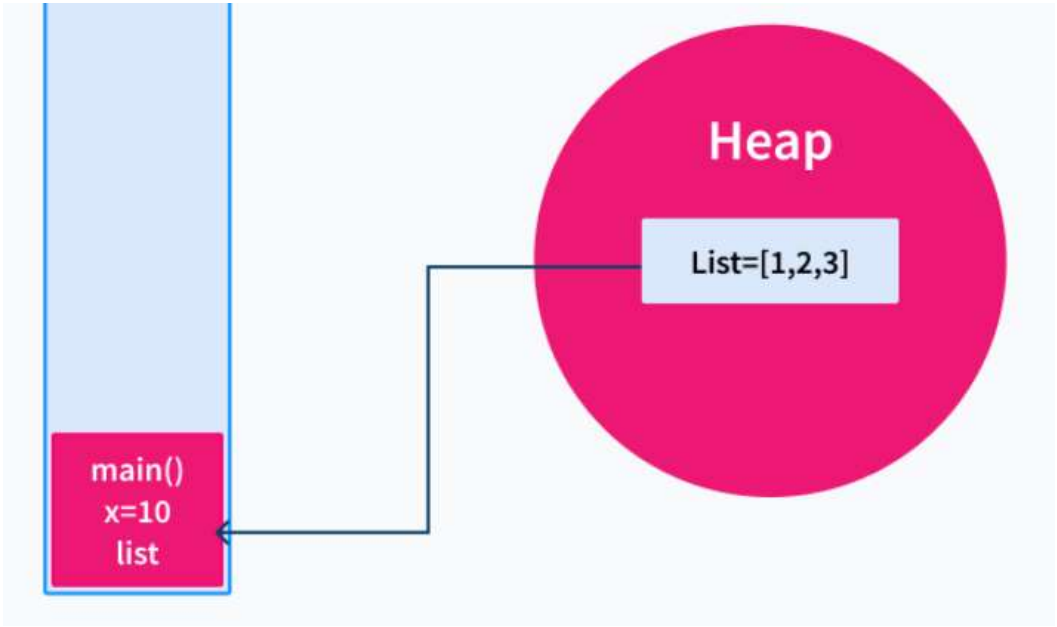


Note: *Stack memory is fixed and cannot be enlarged or shrunk once created. Therefore, if we use all the stack memory, there will be no space left for upcoming method calls, and we will get the* ***StackOverflowError***

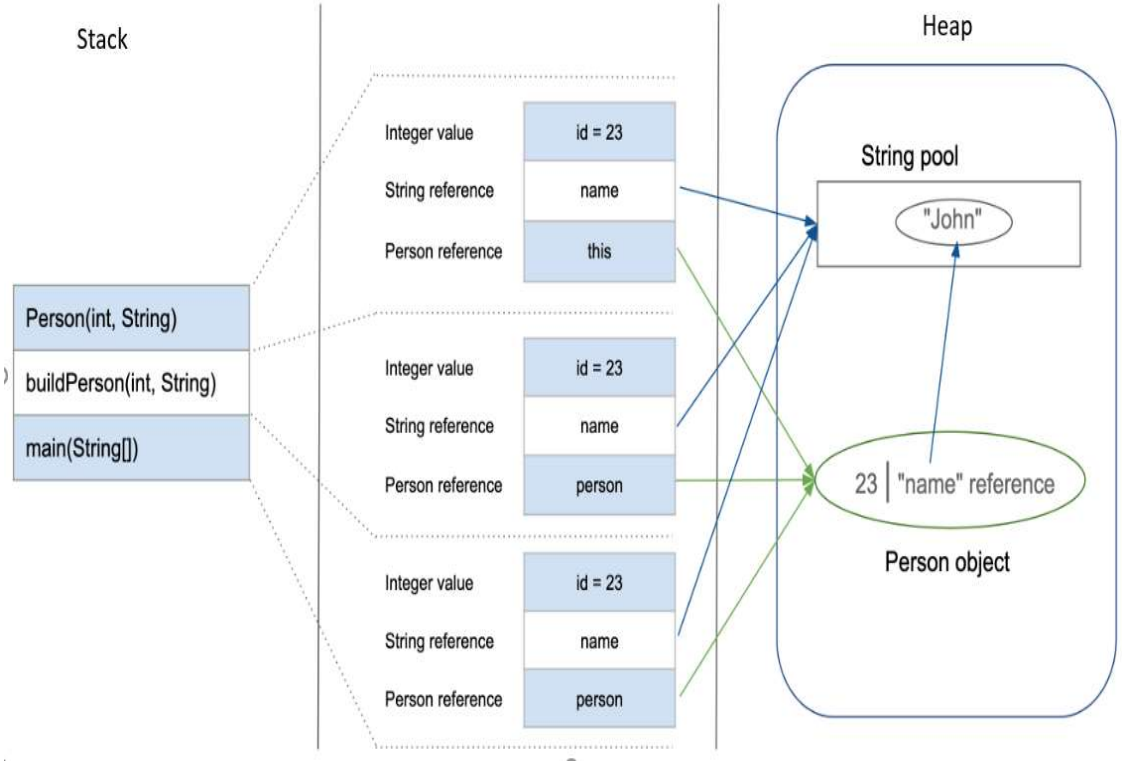
Heap Memory

- *Used to store the objects that are created during the execution of a Java program.*
- *The **reference to the objects** that are created is stored in **stack** memory.*
- *Heap follows **dynamic memory allocation** (memory is allocated during execution or runtime) and provides random access, unlike stack, which follows Last-In-First-Out (LIFO) order.*
- *The size of heap memory is **large** when compared to stack.*
- *The unused objects in the heap memory are cleared automatically by the **Garbage Collector**.*

```
import java.util.ArrayList;
import java.util.List;

public class HeapMemory {
  public static void main(String[] args) {
    int x = 10;

    List < Integer > list = new ArrayList < > ();
    list.add(1);
    list.add(2);
    list.add(3);
  }
}
```



```java
class Person {
    int id;
    String name;
    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
public class PersonBuilder {
    private static Person buildPerson(int id, String name) {
        return new Person(id, name);
    }
    Run | Debug
    public static void main(String[] args) {
        int id = 23;
        String name = "John";
        Person person = buildPerson(id, name);
    }
}
```



Garbage Collections

- *In languages like C and C++,*
  - *the programmer is responsible for both the creation and destruction of objects.*
  - *Sometimes, the programmer may forget to destroy useless objects, and the memory allocated to them is not released.*
  - *The used memory of the system keeps on growing and eventually there is no memory left in the system to allocate. Thus, results in **OutOfMemoryErrors**.*

- *Whereas in Java,*
  - *garbage collection happens **automatically** during the lifetime of a program.*
- *When Java programs run on the JVM, objects are created on the heap*
- *At some point, new objects are created and released. Eventually, some objects are no longer needed.*
- ***The garbage collector finds these unused objects and deletes them to free up memory**.*

```
/* By making a reference null */
Person p = new Person(12, "John");
p=null;
```

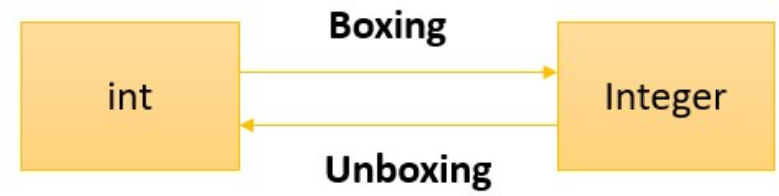
```
/* By assigning a reference to another */
Student studentOne = new Student();
Student studentTwo = new Student();studentOne=studentTwo;
// now the first object referred by studentOne is available for garbage collection
```

- *Since Strings are immutable in Java, whenever String manipulations are performed, automatically a new String would be generated by discarding the older one.*
- *Here, garbage collector finds the unused older string objects and deletes them.*
- *Think of a large application, garbage collecting is a huge task*
- *Well, to avoid garbage in heap, Java came up with **StringBuffer** and **StringBuilder***
- *The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable.*

StringBuffer	***StringBuilder***
StringBuffer operations are thread-safe and synchronized	*StringBuilder operations are not thread-safe are not-synchronized.*
StringBuffer is to used when multiple threads are working on the same String	*StringBuilder is used in a single-threaded environment.*
StringBuffer performance is slower when compared to StringBuilder	*StringBuilder performance is faster when compared to StringBuffer*
***Syntax:** StringBuffer var = new StringBuffer("HELLO");*	***Syntax:** StringBuilder var = new StringBuilder("HELLO");*

Wrapper Classes

- *provides the mechanism to convert primitive into object and object into primitive*
- *Widely used in collection framework*
- *Example*
  - *Primitive – int a = 5;*
  - *Wrapper class – Integer b = new Integer(5);*



Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

- *Boxing and Unboxing is done automatically in Java*
  - *Hence, it called as autoboxing and auto unboxing*

```
/*AutoBoxing*/
int aPrimitive = 30;
Integer aObject = aPrimitive; //Interger aObject2 = new Integer(aPrimitive);
/*AutoUnBoxing*/
Integer bObject = new Integer(33);
int  bPrimitive =  bObject; //int bPrimitive = bObject.intValue();
```

Collection Framework

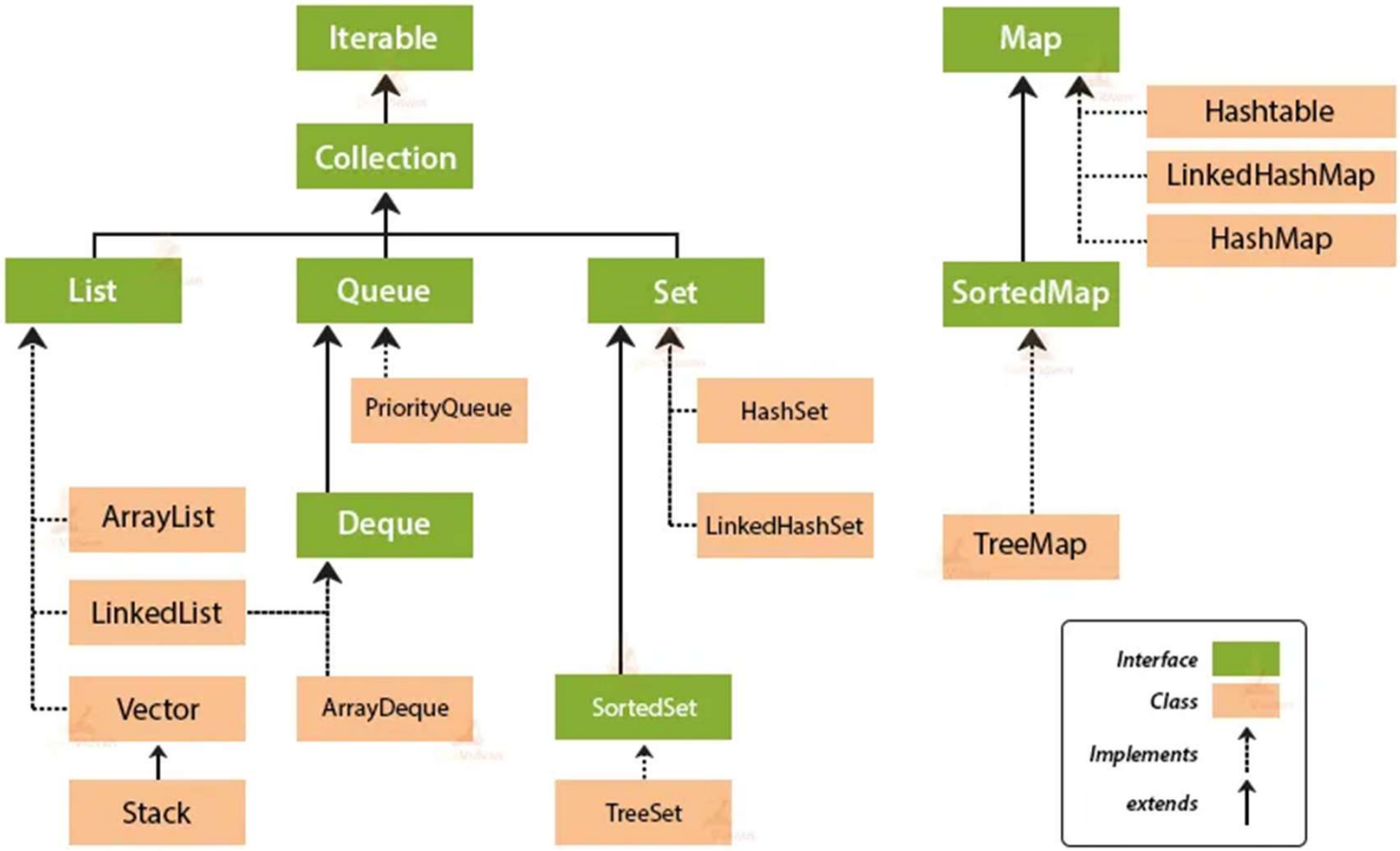
- *Framework is a Readymade architecture*
- *A collection is a single object that represents a group of objects.*
- *The Collections framework in Java is a set of classes and interfaces that implement commonly used **data structures and algorithms**.*
- *Collection Framework enables the user to perform various data manipulation operations like storing data, searching, sorting, insertion, deletion, and updating of data on the group of elements.*

- ***Advantages***
  - ***Reduces programming effort:** A programmer doesn't have to worry about how to implement these data structures and algorithms manually. But rather he can focus on its best use in his program. (Abstraction)*
  - ***Increases program speed and quality:** A programmer need not think of the best implementation of a specific data structure. As the collections framework is highly optimized, programmer can simply use it.*

- ***Important Interfaces** – List, Queue, Set and Map.*
  - *Defined in the **java.util** package*

- ***Note***
  - ***Collection Interface vs Collection Framework** - The Collection interface is the root interface of the collection's framework. The framework includes other interfaces as well: Map and Iterator. These interfaces may also have sub interfaces.*
  - ***Collections Class** - defines many static helper methods which operate on any given collection. Use this class for help with sorting, searching, reversing, or performing other operations on collections.*

Collection Framework Hierarchy in Java



Exceptions

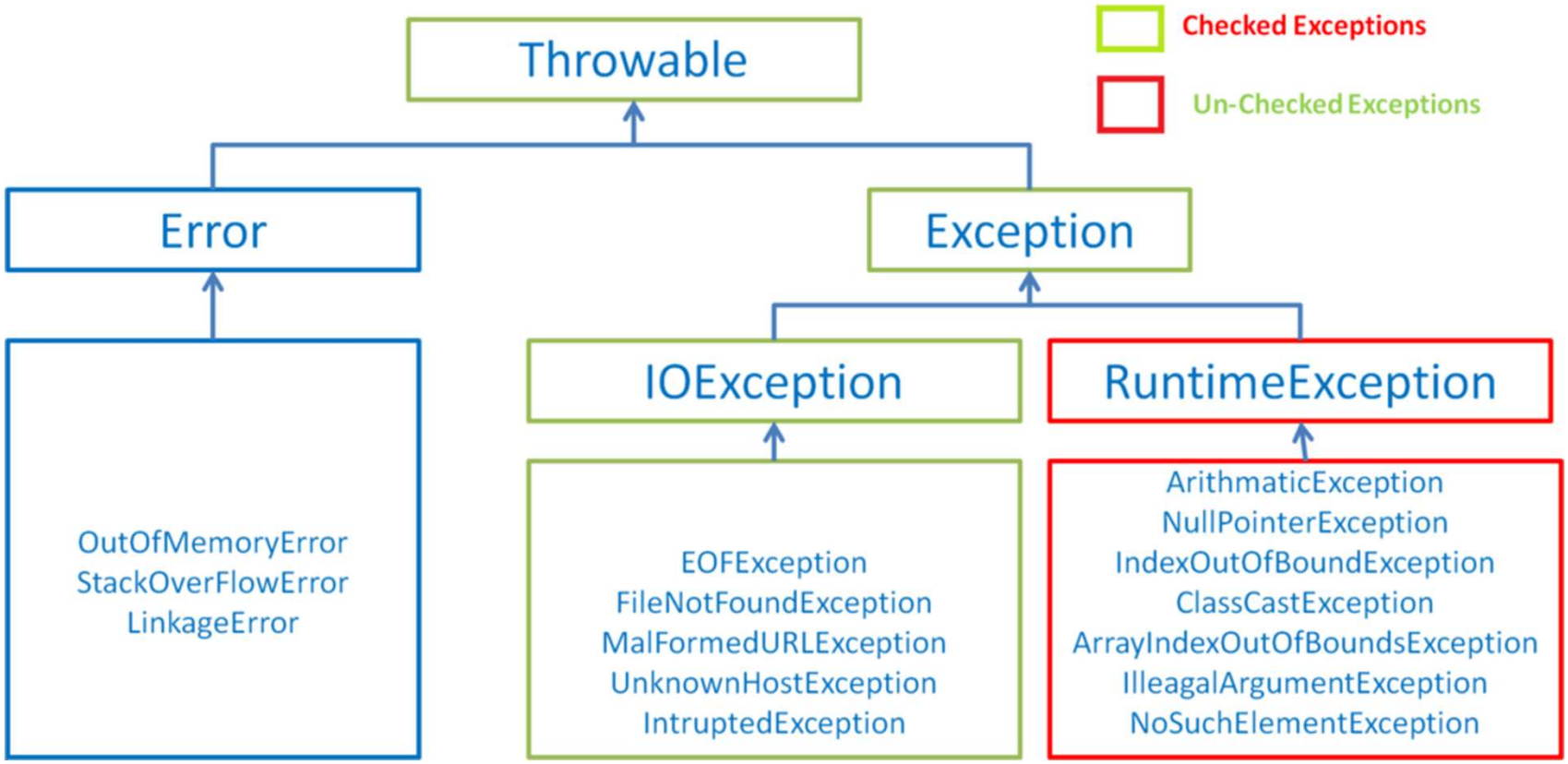
- *An exception is a problem/unexpected event that arises during the execution of a program*

- *When an exception occurs the normal flow of the program is disrupted, and the program/application terminates abnormally.*
    - *which is not recommended; therefore, these exceptions are to be handled.*

- *An exception can occur for many reasons. Some of them are:*
    - *Invalid user input*
    - *Device failure*
    - *Loss of network connection*
    - *Physical limitations (out of disk memory)*
    - *Code errors*
    - *Opening an unavailable*

- *Exception can be handled using*
    - *try..catch block*
    - *finally block*
    - *throw and throws keyword*

Exception Hierarchy in Java



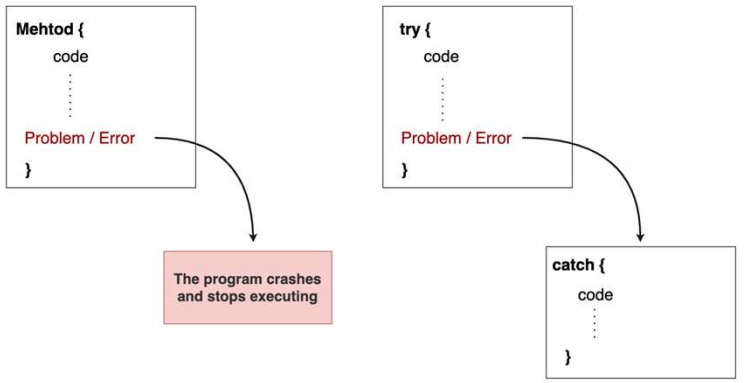
Error

- ***Errors*** *represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.*
- *Errors are usually beyond the control of the programmer, and we should not try to handle errors.*

Exception

- ***Un Checked Exception*** *- An unchecked exception is an exception that occurs at the time of execution. These are also called as* ***Runtime Exceptions****. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.*

- ***Checked Exception*** *- A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as* ***compile time exceptions****. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.*



Lambda Expression

- *Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.*
- ***Syntax:***
  - *parameters -> expression*
  - *(parameter 1, parameter 2) -> expression*
  - *parameters -> { code block}*
- ***Examples:***
  - *() -> System.out.println("Hello")*
  - *(int a, int b) -> System.out.println(a+b)*
  - *() -> {*

```
        System.out.println("Hello");
        System.out.println("Hello")
    }
```

Note: *The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.*

Functional Interface

- *interfaces that* ***have only one abstract method****.*
- *This method is what lambdas are implementing when they are declared - the parameter types and return types of the lambda must match the functional interface method declaration.*