UNIT -I

Biswajit Senapati,
Faculty,
Department of CSE,
NIT AP,
Tadepalligudem, AP.

Program paradigms

The programming paradigm is the way of writing computer programs.

The programming paradigm is the way of writing computer programs. There are four programming paradigms and they are as follows.

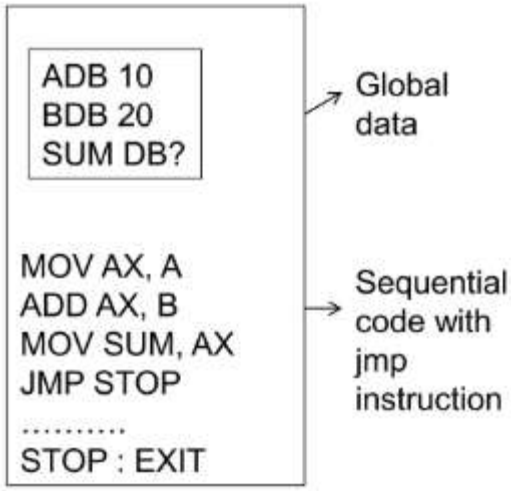
- **Monolithic programming paradigm**
- **Structured-oriented programming paradigm**
- **Procedural-oriented programming paradigm**
- **Object-oriented programming paradigm**

Monolithic Programming Paradigm

The Monolithic programming paradigm is the oldest. It has the following characteristics. It is also known as the imperative programming paradigm.

- In this programming paradigm, the whole program is written in a single block.
- We use the **goto** statement to jump from one statement to another statement.
- It uses all data as global data which leads to data insecurity.
- There are no flow control statements like if, switch, for, and while statements in this paradigm.
- There is no concept of data types.

An example of a Monolithic programming paradigm is **Assembly language**.

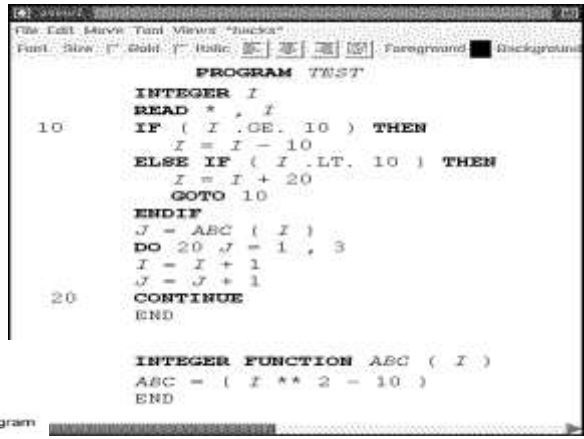


Structured-oriented Programming Paradigm

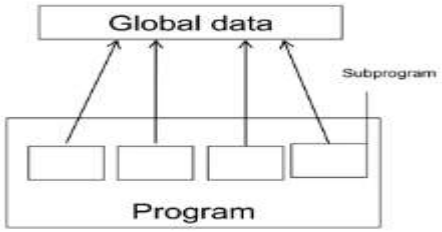
The Structure-oriented programming paradigm is the advanced paradigm of the monolithic paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- In this paradigm, all the data is used as global data which leads to data insecurity.

Examples of a structured-oriented programming paradigm is **ALGOL**, **Pascal**, **PL/I** and **Ada**.



```
      PROGRAM TEST
      INTEGER I
      READ *, I
10    IF ( I .GE. 10 ) THEN
         I = I - 10
      ELSE IF ( I .LT. 10 ) THEN
         I = I + 20
         GOTO 10
      ENDIF
      J = ABC ( I )
      DO 20 J = 1, 3
      I = I + 1
      J = J + 1
20    CONTINUE
      END

      INTEGER FUNCTION ABC ( I )
      ABC = ( I ** 2 - 10 )
      END
```

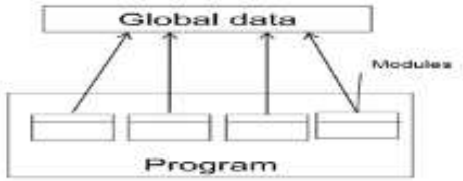


Procedural-oriented Programming Paradigm

The procedure-oriented programming paradigm is the advanced paradigm of a structure-oriented paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- It follows all the concepts of structure-oriented programming paradigm but the data is defined as global data, and also local data to the individual modules.
- In this paradigm, functions may transform data from one form to another.

Examples of procedure-oriented programming paradigm is **C**, **Visual Basic**, **FORTRAN**, etc.



Object-oriented Programming Paradigm

The object-oriented programming paradigm is the most popular. It has the following characteristics.

- In this paradigm, the whole program is created on the concept of objects.
- In this paradigm, objects may communicate with each other through function.
- This paradigm mainly focuses on data rather than functionality.
- In this paradigm, programs are divided into what are known as objects.
- It follows the bottom-up flow of execution.
- It introduces concepts like data abstraction, inheritance, and overloading of functions and operators overloading.
- In this paradigm, data is hidden and cannot be accessed by an external function.
- It has the concept of friend functions and virtual functions.
- In this paradigm, everything belongs to objects.

Examples of procedure-oriented programming paradigm is **C++**, **Java**, **C#**, **Python**, etc.

- The **POP** structure program follows the top-bottom flow of execution.
- The **OOP** structure program follows the bottom-top flow of execution.

General Structure of C++ Program (POP)

```
/*documentation*/
pre-processing statements
global declaration;
void main()
{
    Local declaration;
    Executable statements;
    .
    .
    .
}
User defined functions
{
    function body
    .
    .
}
```

C++ Program to perform addition of two numbers using POP structure

```
#include < iostream.h >
int a, b;
void main()
{
    int c;
    void get_data();
    int sum(int, int);

    get_data();
    c = sum(a, b);
    cout << "Sum = " << endl;
}
void get_data()
{
    cout << "Enter any two numbers: ";
    cin >> a >> b;
}

int sum(int a, int b)
{
    return a + b;
}
```

General Structure of C++ Program (OOP)

```
/*documentation*/
pre processing statements
class ClassName
{
    member variable declaration;
    .
    .
    .
    member functions(){
        function body
        .
        .
    }
};
void main ()
{
    ClassName object;
    object.member;
}
```

C++ Program to perform addition of two numbers using OOP structure

```
#include < iostream.h >

class Addition{
    int a, b;
    public:
        get_data(){
            cout << "Enter any two numbers: ";
            cin >> a >> b;
        }
        int sum(){
            get_data();
            return a + b;
        }
};

void main(){
    Addition obj;
    cout << "Sum = " << sum() << endl;
}
```

A WAY OF VIEWING THE WORLD

- A way of viewing the world is an idea to illustrate the object-oriented programming concept with an example of a real-world situation.
- Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?
- To solve the problem, let me call zomato (an **agent** in food delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my family members. Look at the following image.

Agents and Communities

- To solve my food delivery problem, I used a solution by finding an appropriate agent (Zomato) and pass a message containing my request. It is the responsibility of the agent (Zomato) to satisfy my request. Here, the agent uses some method to do this. I do not need to know the method that the agent has used to solve my request. This is usually hidden from me.

- So, in object-oriented programming, problem-solving is the solution to our problem which requires the help of many individuals in the community. We may describe agents and communities as follows.

An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community.

- In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

Messages and Methods

- To solve my problem, I started with a request to the agent zomato, which led to still more requestes among the members of the community until my request has done. Here, the members of a community interact with one another by making requests until the problem has satisfied.

In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action. The receiver is the object to whom the message was sent. In response to the message, the receiver performs some method to carry out the request. Every message may include any additional information as arguments.

- In our example, I send a request to zomato with a message that contains food items, the quantity of food, and the hotel details. The receiver uses a method to food get delivered to my home.

Responsibilities

- In object-oriented programming, behaviors of an object described in terms of responsibilities.
- In our example, my request for action indicates only the desired outcome (food delivered to my family). The agent (zomato) free to use any technique that solves my problem. By discussing a problem in terms of responsibilities increases the level of abstraction. This enables more independence between the objects in solving complex problems.

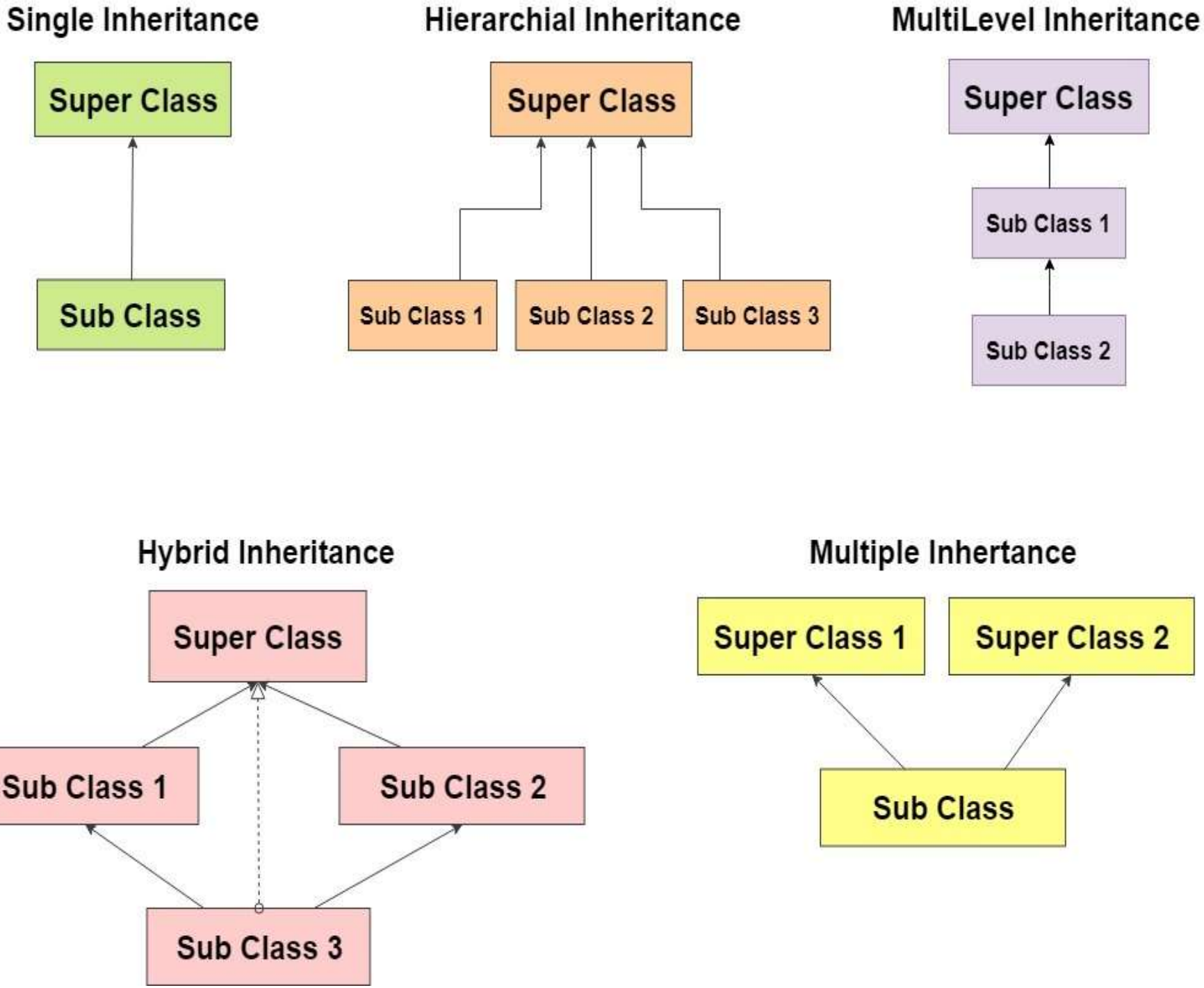
Classes and Instances

- In object-oriented programming, all objects are instances of a class. The method invoked by an object in response to a message is decided by the class. All the objects of a class use the same method in response to a similar message.
- In our example, the zomato a class and all the hotels are sub-classes of it. For every request (message), the class creates an instance of it and uses a suitable method to solve the problem.

Classes Hierarchies

- A graphical representation is often used to illustrate the relationships among the classes (objects) of a community. This graphical representation shows classes listed in a hierarchical tree-like structure. In this more abstract class listed near the top of the tree, and more specific classes in the middle of the tree, and the individuals listed near the bottom.

In object-oriented programming, classes can be organized into a hierarchical inheritance structure. A child class inherits properties from the parent class that higher in the tree.


Single Inheritance
Super Class
Sub Class
Hierarchial Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
MultiLevel Inheritance
Super Class
Sub Class 1
Sub Class 2
Hybrid Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
Multiple Inhertance
Super Class 1
Super Class 2
Sub Class

Method Binding, Overriding, and Exception

- In the class hierarchy, both parent and child classes may have the same method which implemented individually. Here, the implementation of the parent is overridden by the child. Or a class may provide multiple definitions to a single method to work with different arguments (overloading).
- The search for the method to invoke in response to a request (message) begins with the class of this receiver. If no suitable method is found, the search is performed in the parent class of it. The search continues up the parent class chain until either a suitable method is found or the parent class chain is exhausted. If a suitable method is found, the method is executed. Otherwise, an error message is issued.

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

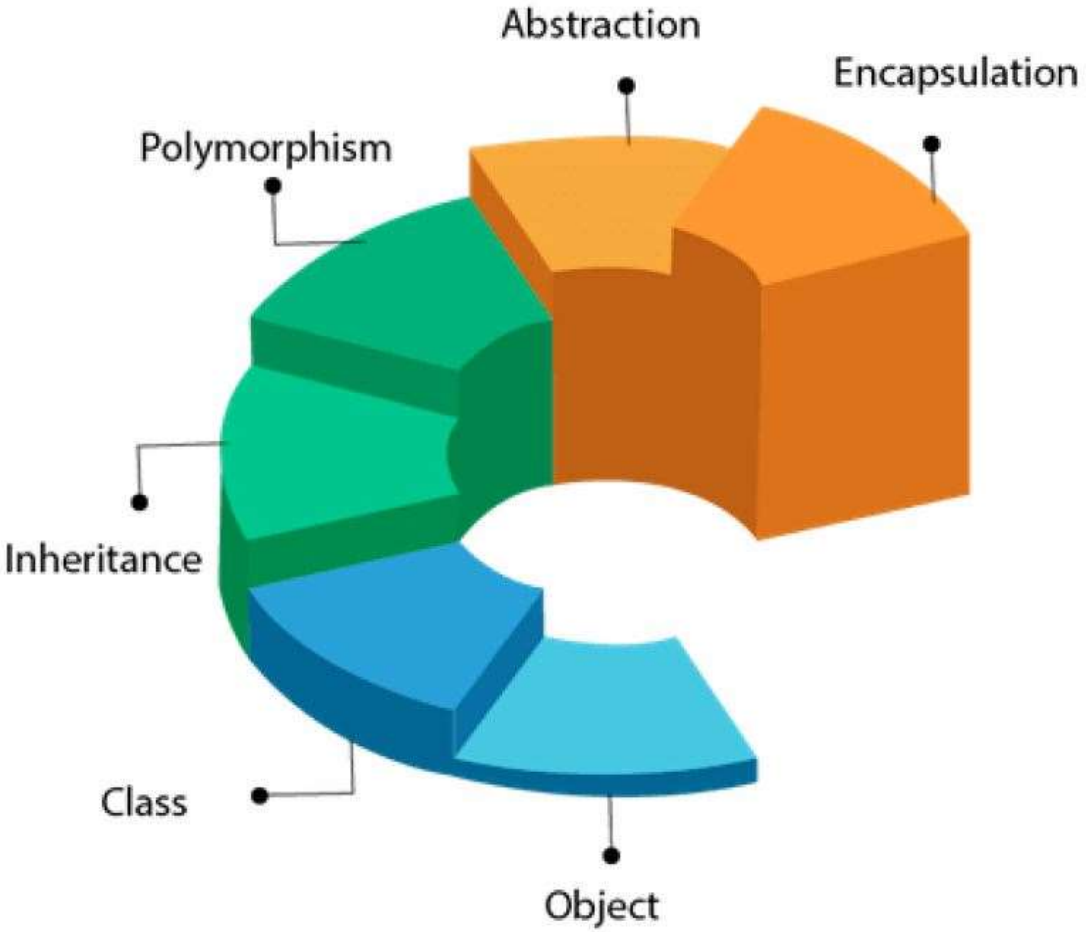
Same Method Name, Same parameter

Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name, Different Parameter

SUMMARY OF OBJECT-ORIENTED CONCEPTS

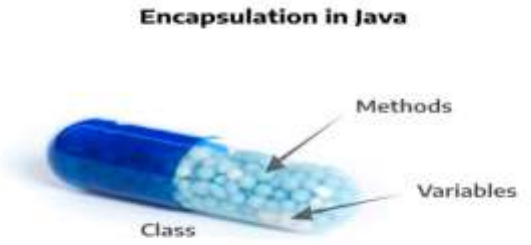


Encapsulation

Encapsulation is the process of wrapping up data (properties) and methods (behavior) of into a single units and unit here is class.

- Encapsulate in plain English means *to enclose or be enclosed in or as if in a capsule*. In Java, everything is enclosed within a class or interface, unlike languages such as C and C++ where we can have global variables outside classes.

- Encapsulation enables **data hiding,** hiding irrelevant information from the users of a class and exposing only the relevant details required by the user. We can expose our operations hiding the details of what is needed to perform that operation.



Abstraction

Abstraction is the act of representing the essential feature without knowing the background details.

- Abstraction captures only those details about an object that are relevant to the current perspective, so that the programmer can focus on a few concepts at a time.
- Java provides interfaces and abstract classes for describing abstract types.
  - An **interface** is a contract or specification without any implementation. An interface can't have behavior or state.
  - An **abstract class** is a class that cannot be instantiated, but has all the properties of a class including constructors. Abstract classes can have state and can be used to provide a skeletal implementation.

Polymorphism

Polymorphism is the ability of an object to take on many forms

Java supports different kinds of polymorphism like **overloading** **and** **overriding**.

- **Overriding** implements Runtime Polymorphism whereas **Overloading** implements Compile time polymorphism.
- The method **Overriding** occurs between super class and subclass. **Overloading** occurs between the methods in the same class.
- **Overriding** methods have the same signature i.e. same name and method arguments. **Overloaded** method names are the same but the parameters are different.
- With **Overloading**, the method to call is determined at the compile-time. With o**verriding**, the method call is determined at the runtime based on the object type.
- If **overriding** breaks, it can cause serious issues in our program because the effect will be visible at runtime. Whereas if **overloading** breaks, the compile-time error will come and it's easy to fix.

Inheritance

Inheritance describes the parent child relationship between two class

- A class can get some of its characteristics from a parent class and then add more unique features of its own. For example, consider a Vehicle parent class and a child class Car. Vehicle class will have properties and functionalities common for all vehicles. Car will inherit those common properties from the Vehicle class and then add properties which are specific to a car.

- In the above example, Vehicle parent class is known as base class or super class. Car is known as derived class, Child class or subclass.

- Java supports single-parent, multiple-children inheritance and multilevel inheritance (Grandparent-> Parent -Child) for classes and interfaces.

- Java supports multiple inheritance (multiple parents, single child) only through interfaces. This is done to avoid some confusions and errors such as diamond problem of inheritance.

Class

Collection of objects is called class

- It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class does not consume any space.

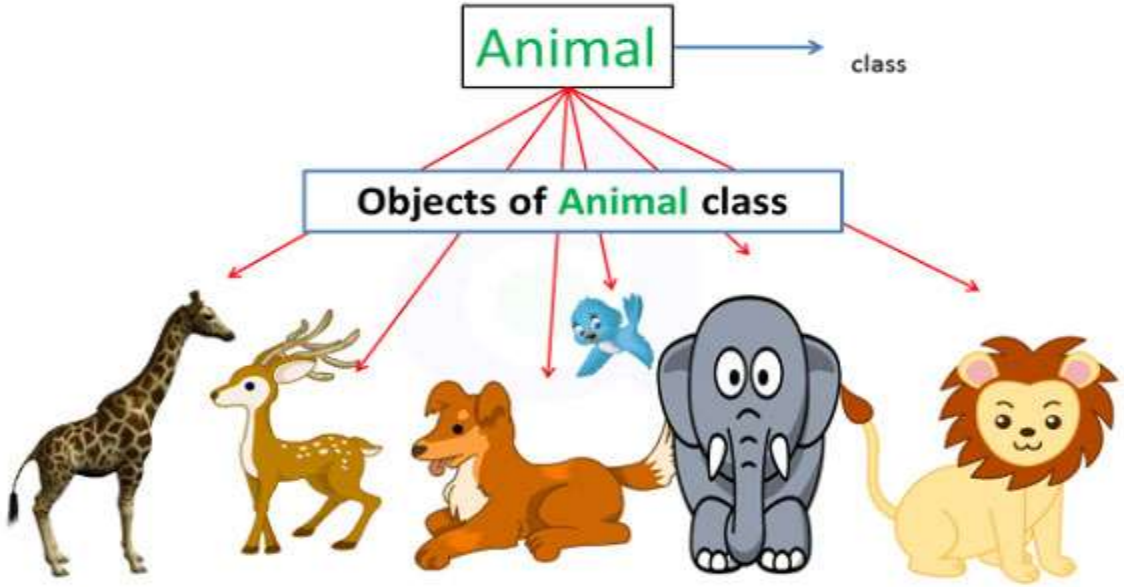
Example : Animal is a class

Object

An entity that have data (state) and method (behavior) is known as known as an object.

- Object can be defined as instance of a class
- Object can be physical or logical

Example : lion, deer, fox, bird, elephant and giraffe.



JAVA

.**Overview of Java:**

- **James Gosling, Mike Sheridan, and Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.
- Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .**gt.**
- After that, it was called **Oak** and was developed as a part of the **Green project.**
- **Why Oak?**

Oak is a symbol of **strength and chosen as a national tree** of many countries like the U.S.A., France, Germany, Romania, etc.

Why Java is named "Java"?

- They wanted something that reflected the essence of the technology: **revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.**

- In 1995, Oak was renamed as **Java**

  Java is an island in **Indonesia** where the first coffee was produced (called **Java coffee**). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

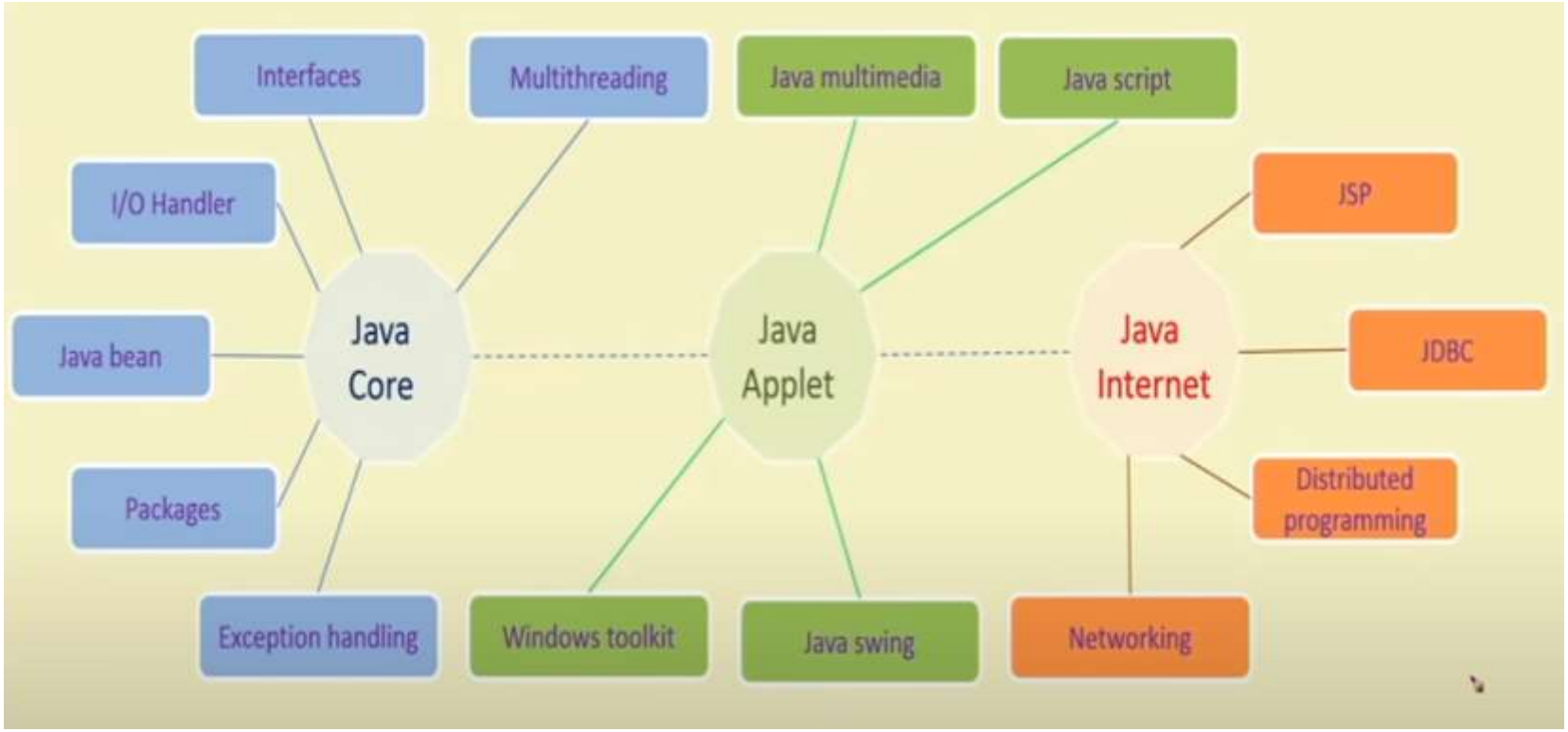
- Initially developed by James Gosling at **Sun Microsystems** (which is now a subsidiary of **Oracle Corporation**) and released in 1995.

- In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

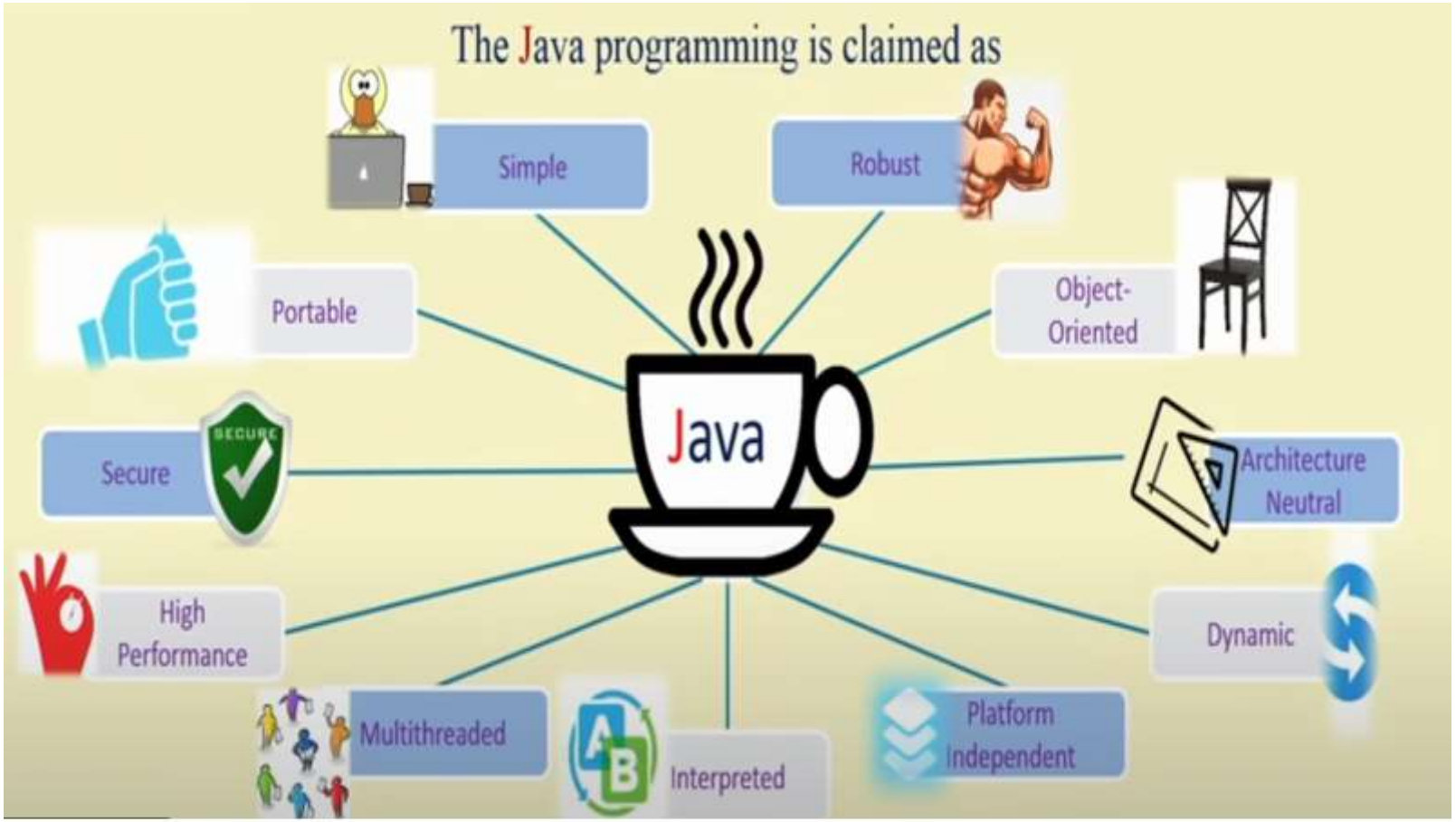
- **JDK (Java Development Kit)** 1.0 was released on January 23, 1996.

- Java divided into three categories, they are
  - J2SE (**Java 2 Standard Edition**)
  - J2EE (**Java 2 Enterprise Edition**)
  - J2ME (**Java 2 Micro or Mobile Edition**)

- **J2SE (**J2SE is used for developing **client side** applications.)
- **J2EE (**J2EE is used for developing **server side** applications.)
- **J2ME**J2ME is used for developing **mobile or wireless application** by making use of a predefined protocol called WAP (wireless Access / Application protocol.)



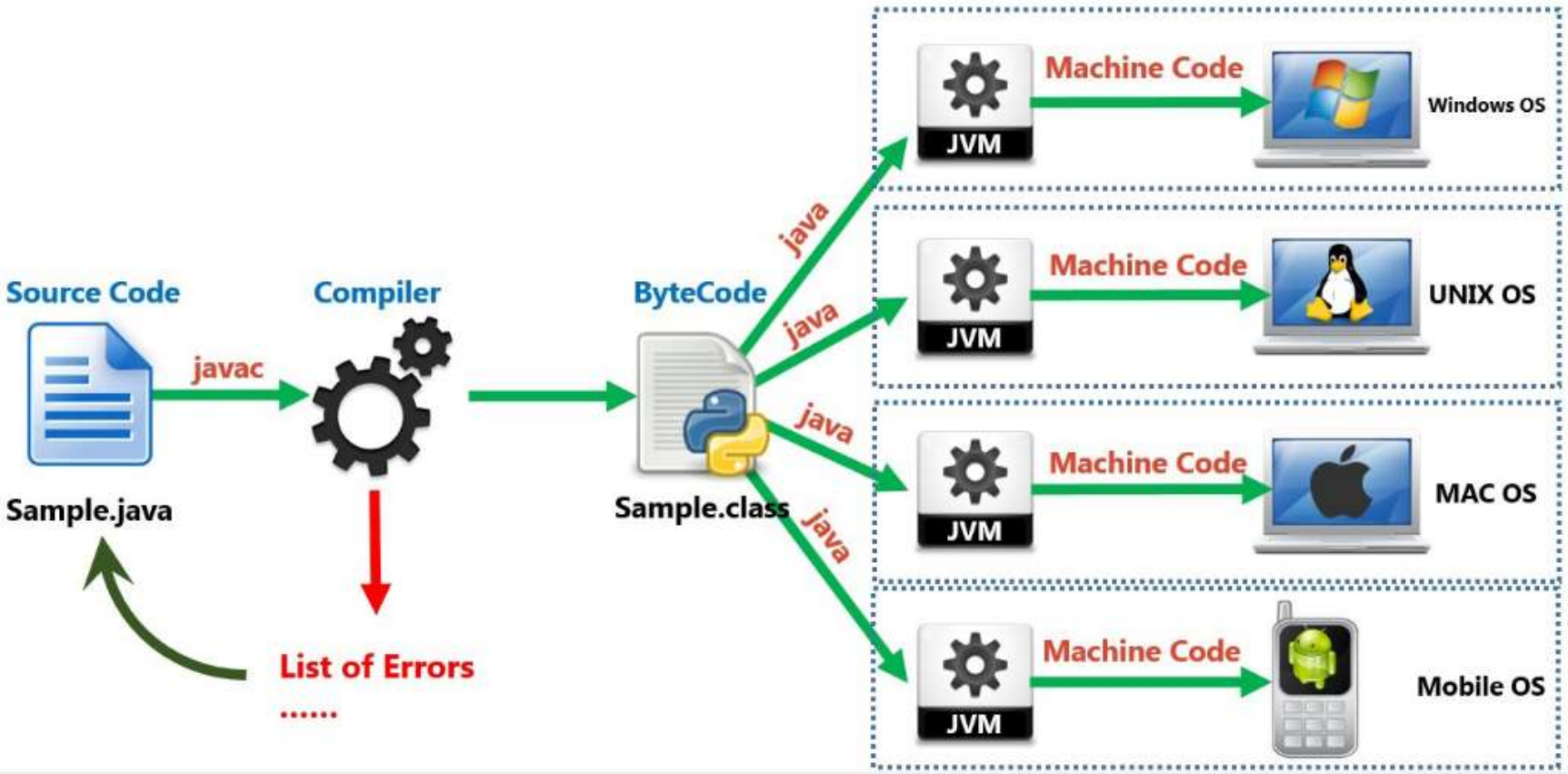
JAVA BUZZWORDS/ FEATURES



Execution Process of Java Program

The following three steps are used to create and execute a java program.

- Create a source code (.java file).
- Compile the source code using javac command.
- Run or execute .class file uisng java command.



MAIN METHOD

- A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task. The major advantage of methods is code re-usability (define the code once, and use it many times).
- In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.
- Every method in java must be declared inside a class.

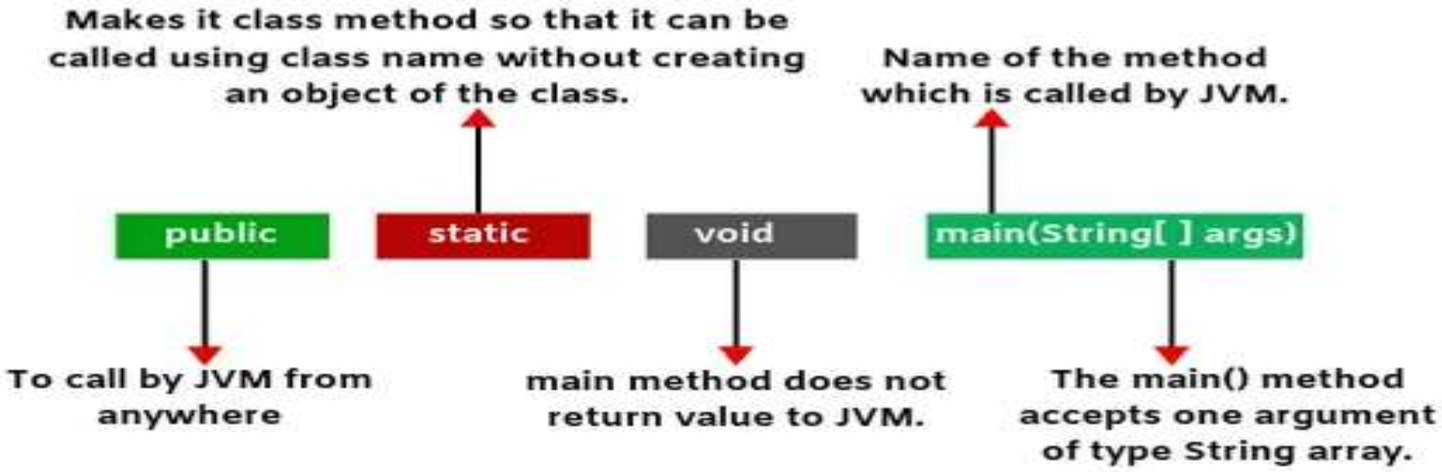
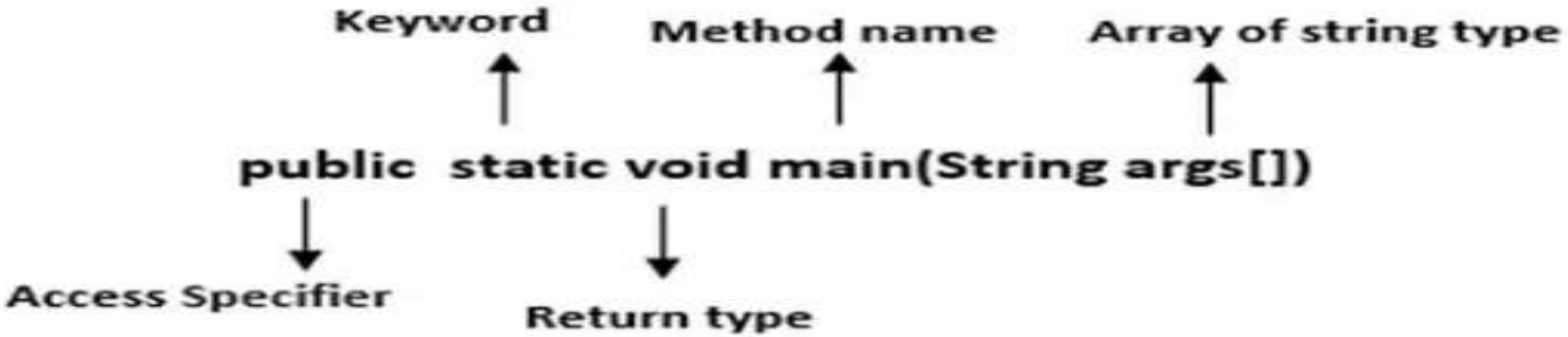
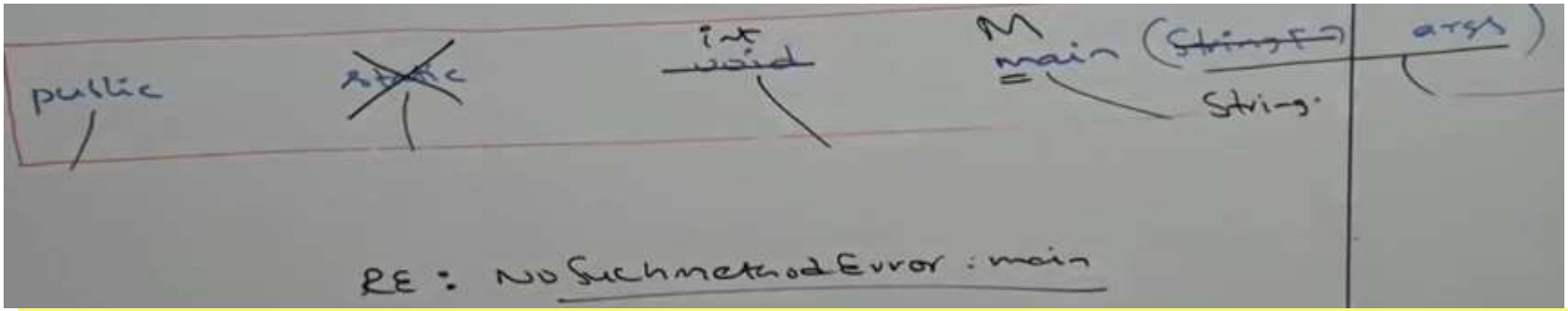


Fig: Java main method



- **Public :** It is an **access specifier** for the Method. Access specifiers are nothing but the keywords that set the Accessibility of the classes, methods, and other members. So, here **we need to set the access specifier as public for the main method to enable access to the Run time environment so that JVM can make a call to the main method directly.**
- **Static**: **Static keyword is basically used when there is no object instance for the class,** when the main method is called in run time it should be declared as static because we don't have any object instance for the main method, if we didn't declare as static for the main method, JVM can't load the main method into the memory, and it throws an error.
- **Void**: Void keyword is used to identify that the method is not returning any data type. **When the main program execution completes then Java Program also terminates, so if we are trying to return anything after the java program terminates it will throw a compilation error.**
- **Main:** the main method is the keyword used to identify the main method that JVM calls during the start of execution, **if we didn't provide a name as the main method, JVM can't identify the main method to start the execution.**
- **String [] args:** args is an array of type String. **Java's main method accepts the command line arguments of string data type.** We can pass command line arguments from Java Run configurations at the run time.



* The above syntax is very strict if we perform any change we will get runtime exception saying NoSuchMethodError:main.

1. The order of modifiers is not important that is instead of "public static" we can take "static public" also.

2. We can declare "String[]" in any acceptable form.

```
main(String[] args)
main(String []args)
main(String args[])
```

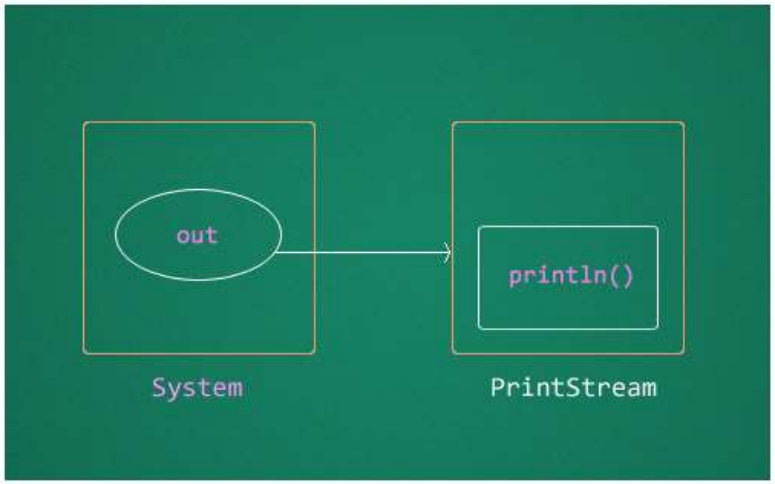
3. instead of 'args' we can take any valid java identifier.

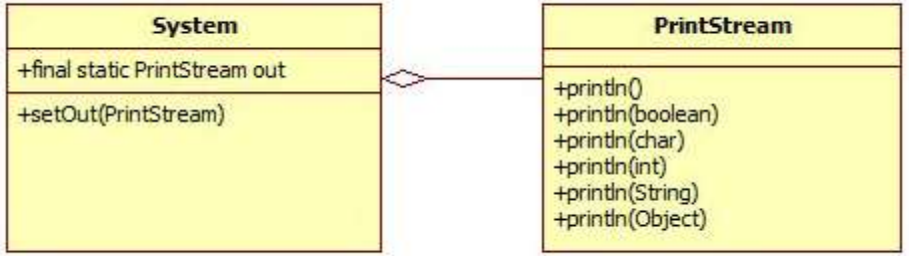
Identifier – Consist of letters (A-Z, a-z)

System.out.println()

System.out.println("Hello");

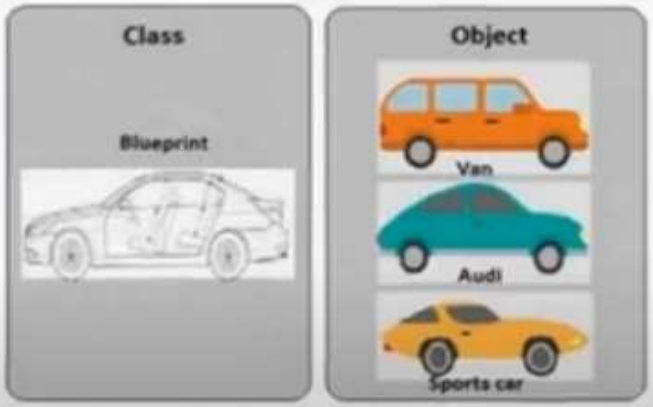
* System is class present in java.lang package
* 'out' is a static variable present in system class of type PrintStream
* Println() is a method present in PrintStream class.





Object Creation

"A class is the blueprint from which individual objects are created."



objects ie. instance of class(Blueprint/structure/design/template)

Blueprint or template

In real world car is object and it will have 2 characteristic

State

- size
- color
- make
- model

Variables/Fields

Behaviour

- move
- accelerate
- turn
- reverse
- shift

Methods/Functions

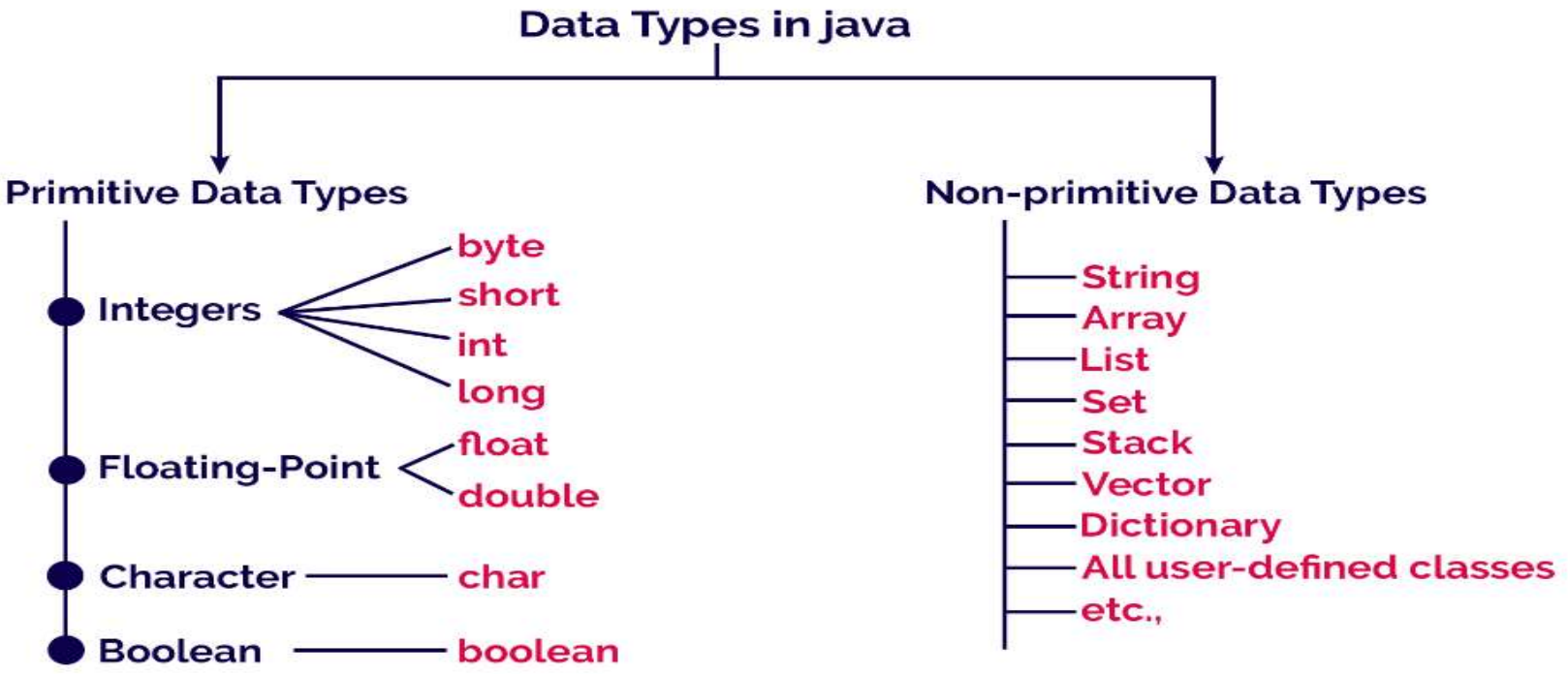
```
class Car {          (State or variables)
 int size;
 String color;

                              (behaviour or methods)
 public void setSize(int s){
    this.size = s;
 }

public void setColor(String c){
    this.color = c;
 }
//more code goes here
}
```

```
public static void main(String[] args) {

Car carObject1 = new Car();
Car carObject2 = new Car();
Car carObject3 = new Car();

carObject1.setSize(6);
carObject1.setColor("blue");

carObject1.setSize(4);
carObject1.setColor("red");

carObject1.setSize(8);
carObject1.setColor("grey");
}
```

DATA TYPES

Java programming language has a rich set of data types. The data type is a category of data stored in variables. In java, data types are classified into two types and they are as follows.

* Primitive Data Types
* Non-primitive Data Types



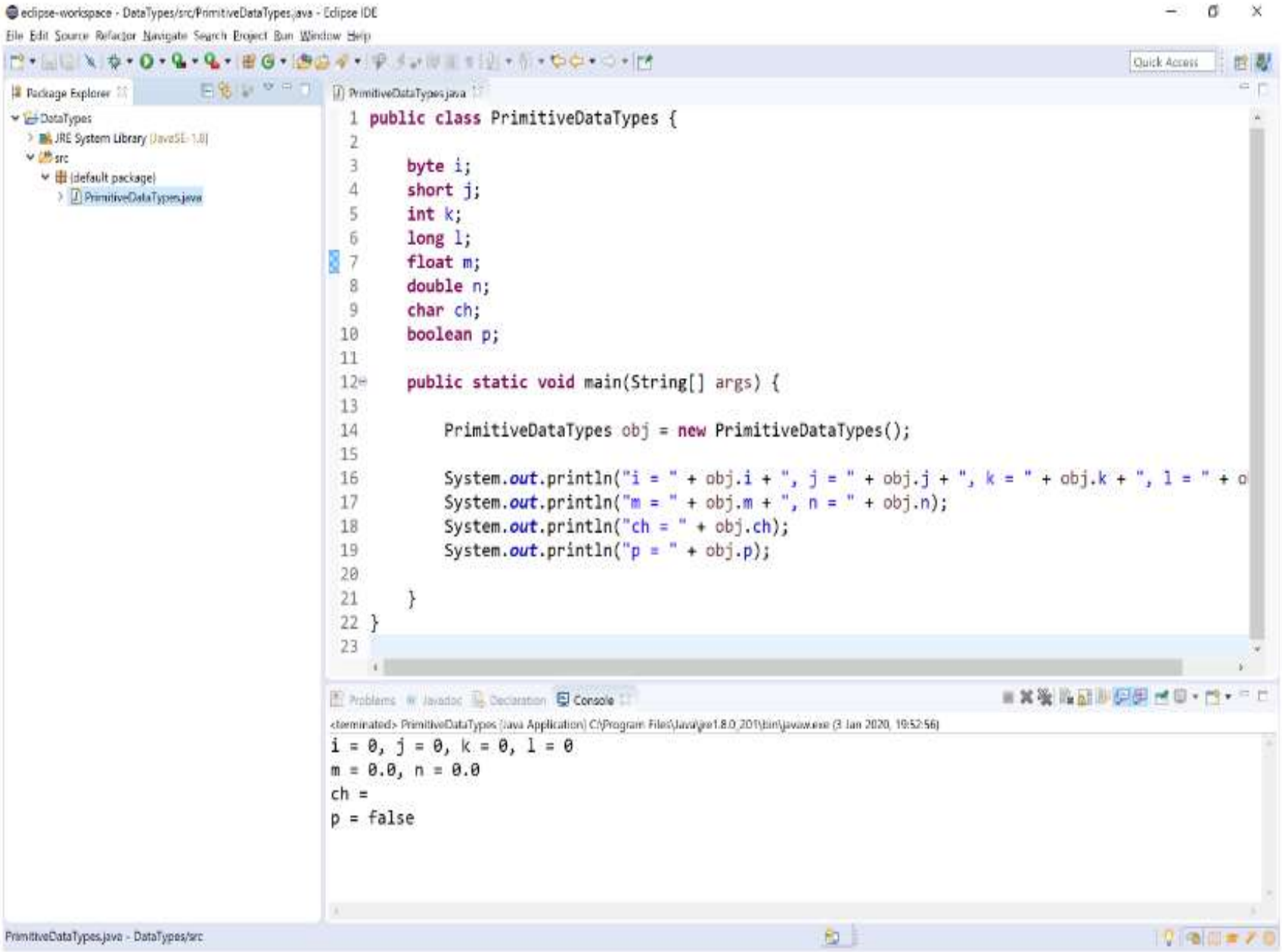
Primitive Data Types

The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size. The primitive data types do not have any additional methods.

In java, primitive data types includes `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

The following table provides more description of each primitive data type.

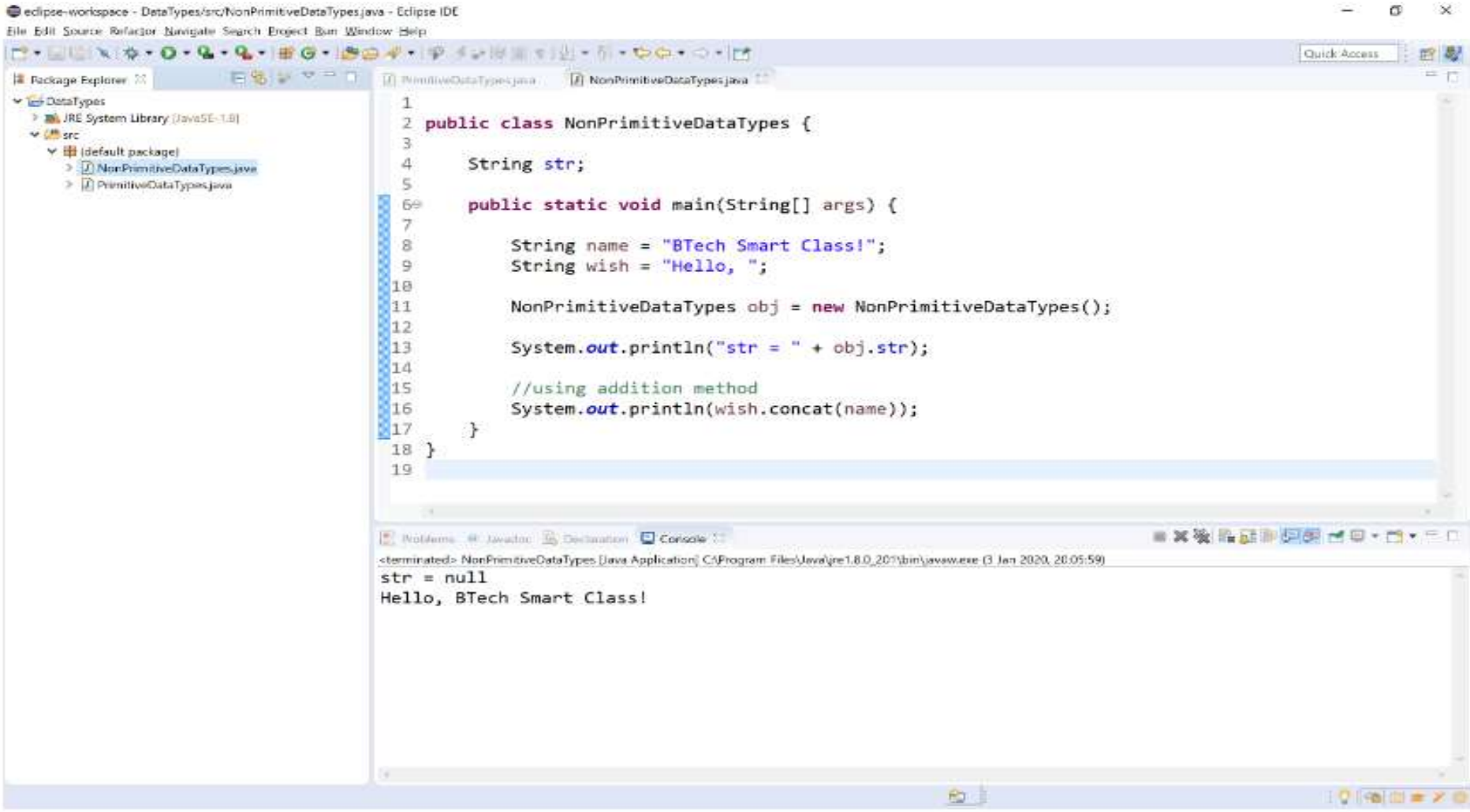
Data type	Meaning	Memory size	Range	Default Value
byte	Whole numbers	1 byte	-128 to +127	0
short	Whole numbers	2 bytes	-32768 to +32767	0
int	Whole numbers	4 bytes	-2.147.483.648 to +2.147.483.647	0
long	Whole numbers	8 bytes	-9.223.372.036.854.775.808 to +9.223.372.036.854.775.807	0L
float	Fractional numbers	4 bytes	-	0.0f
double	Fractional numbers	8 bytes	-	0.0d
char	Single character	2 bytes	0 to 65535	\u0000
boolean	unsigned char	1 bit	0 or 1	0 (false)


eclipse-workspace - DataTypes/src/PrimitiveDataTypes.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
DataTypes
JRE System Library [JavaSE-1.8]
src
(default package)
PrimitiveDataTypes.java
PrimitiveDataTypes.java
1 public class PrimitiveDataTypes {
2
3 byte i;
4 short j;
5 int k;
6 long l;
7 float m;
8 double n;
9 char ch;
10 boolean p;
11
12 public static void main(String[] args) {
13
14 PrimitiveDataTypes obj = new PrimitiveDataTypes();
15
16 System.out.println("i = " + obj.i + ", j = " + obj.j + ", k = " + obj.k + ", l = " + o
17 System.out.println("m = " + obj.m + ", n = " + obj.n);
18 System.out.println("ch = " + obj.ch);
19 System.out.println("p = " + obj.p);
20
21 }
22 }
23
Problems Javadoc Declaration Console
<terminated> PrimitiveDataTypes [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (3 Jan 2020, 10:52:56)
i = 0, j = 0, k = 0, l = 0
m = 0.0, n = 0.0
ch =
p = false
PrimitiveDataTypes.java - DataTypes/src

Non-primitive Data Types

In java, non-primitive data types are the reference data types or user-created data types. All non-primitive data types are implemented using object concepts. Every variable of the non-primitive data type is an object. The non-primitive data types may use additional methods to perform certain operations. The default value of non-primitive data type variable is null.

In java, examples of non-primitive data types are String, Array, List, Queue, Stack, Class, Interface, etc.



Primitive Vs Non-primitive Data Types

Primitive Data Type	Non-primitive Data Type
These are built-in data types	These are created by the users
Does not support additional methods	Support additional methods
Always has a value	It can be null
Starts with lower-case letter	Starts with upper-case letter
Size depends on the data type	Same size for all

Methods

A method is a block of statements under a name that gets executes only when it is called. Every method is used to perform a specific task.

The major advantage of methods is code re-usability (define the code once, and use it many times).

In a java programming language, a method defined as a behavior of an object. That means, every method in java must belong to a class.

Every method in java must be declared inside a class.

Every method declaration has the following characteristics.

* `returnType` - Specifies the data type of a return value.
* `name` - Specifies a unique name to identify it.
* `parameters` - The data values it may accept or recieve.
* `{}` - Defienes the block belongs to the method.

Creating a method

A method is created inside the class and it may be created with any access specifier. However, specifying access specifier is optional.

Following is the syntax for creating methods in java.

Syntax

```
class <ClassName>{
    <accessSpecifier> <returnType> <methodName>( parameters ){
        ...
        block of statements;
        ...
    }
}
```

- The methodName must begin with an alphabet, and the Lower-case letter is preferred.
- The methodName must follow all naming rules.
- If you don't want to pass parameters, we ignore it.
- If a method defined with return type other than void, it must contain the return statement, otherwise, it may be ignored.

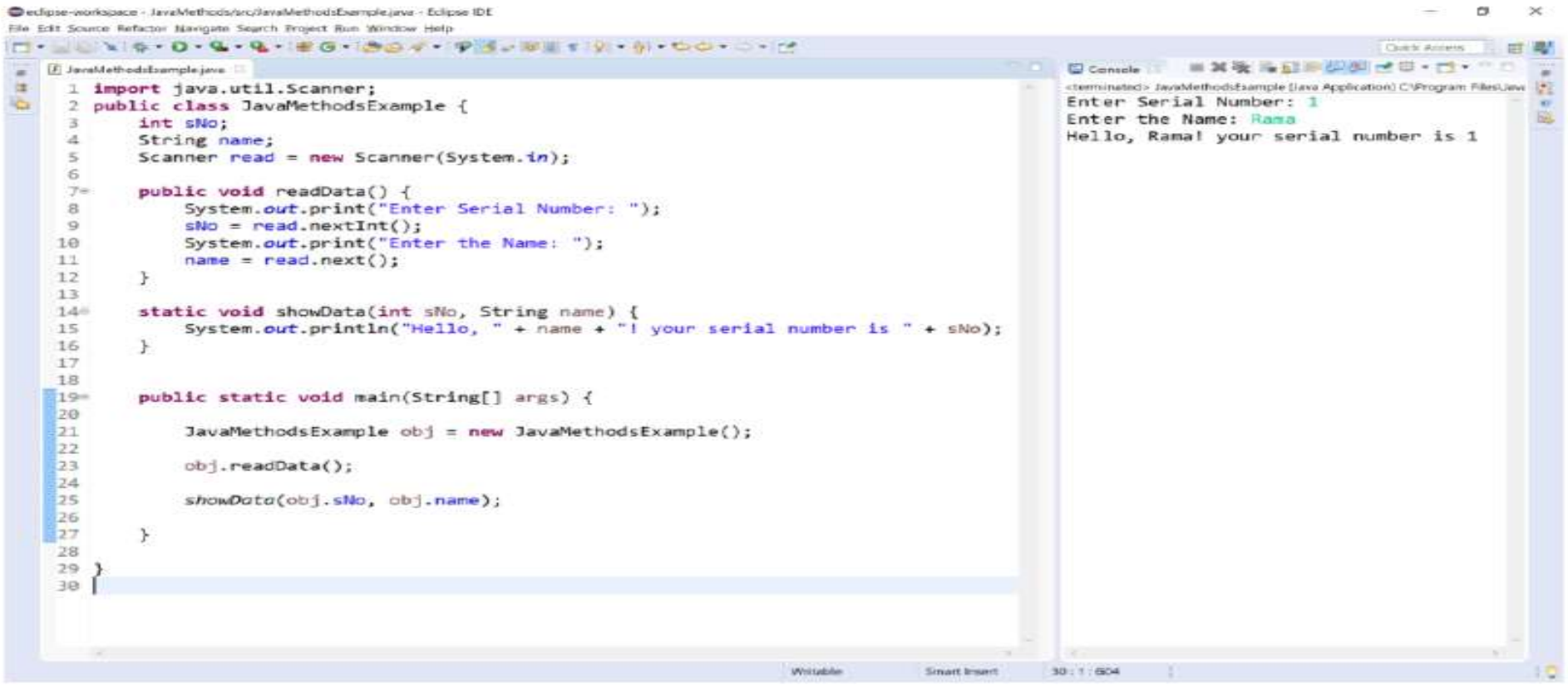
Calling a method

In java, a method call precedes with the object name of the class to which it belongs and a dot operator. It may call directly if the method defined with the static modifier. Every method call must be made, as to the method name with parentheses (), and it must terminate with a semicolon.

Syntax

```
<objectName>.<methodName>( actualArguments );
```

- The method call must pass the values to parameters if it has.
- If the method has a return type, we must provide the receiver.



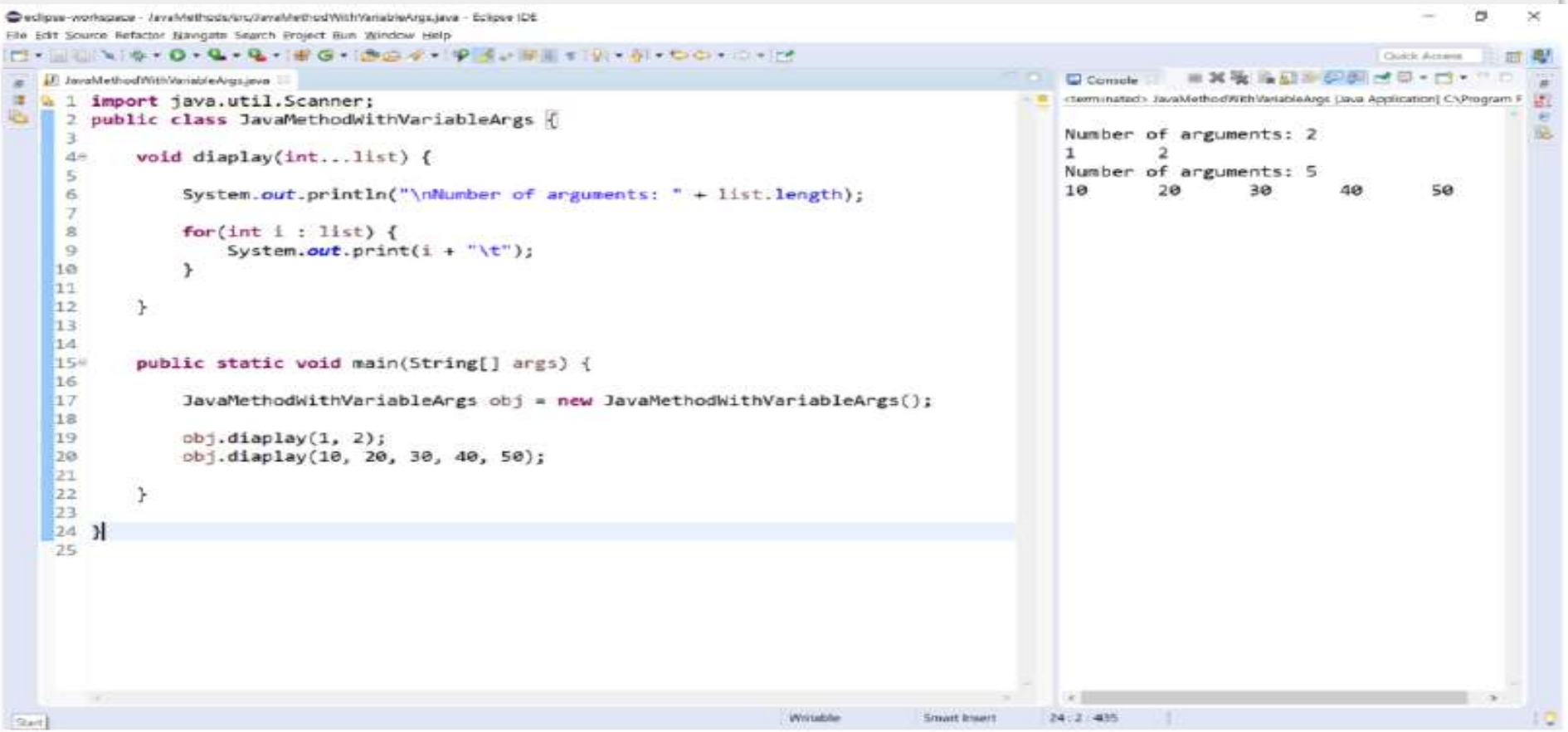
- The objectName must begin with an alphabet, and a Lower-case letter is preferred.
- The objectName must follow all naming rules.

Variable arguments of a method

In java, a method can be defined with a variable number of arguments. That means creating a method that receives any number of arguments of the same data type.

Syntax

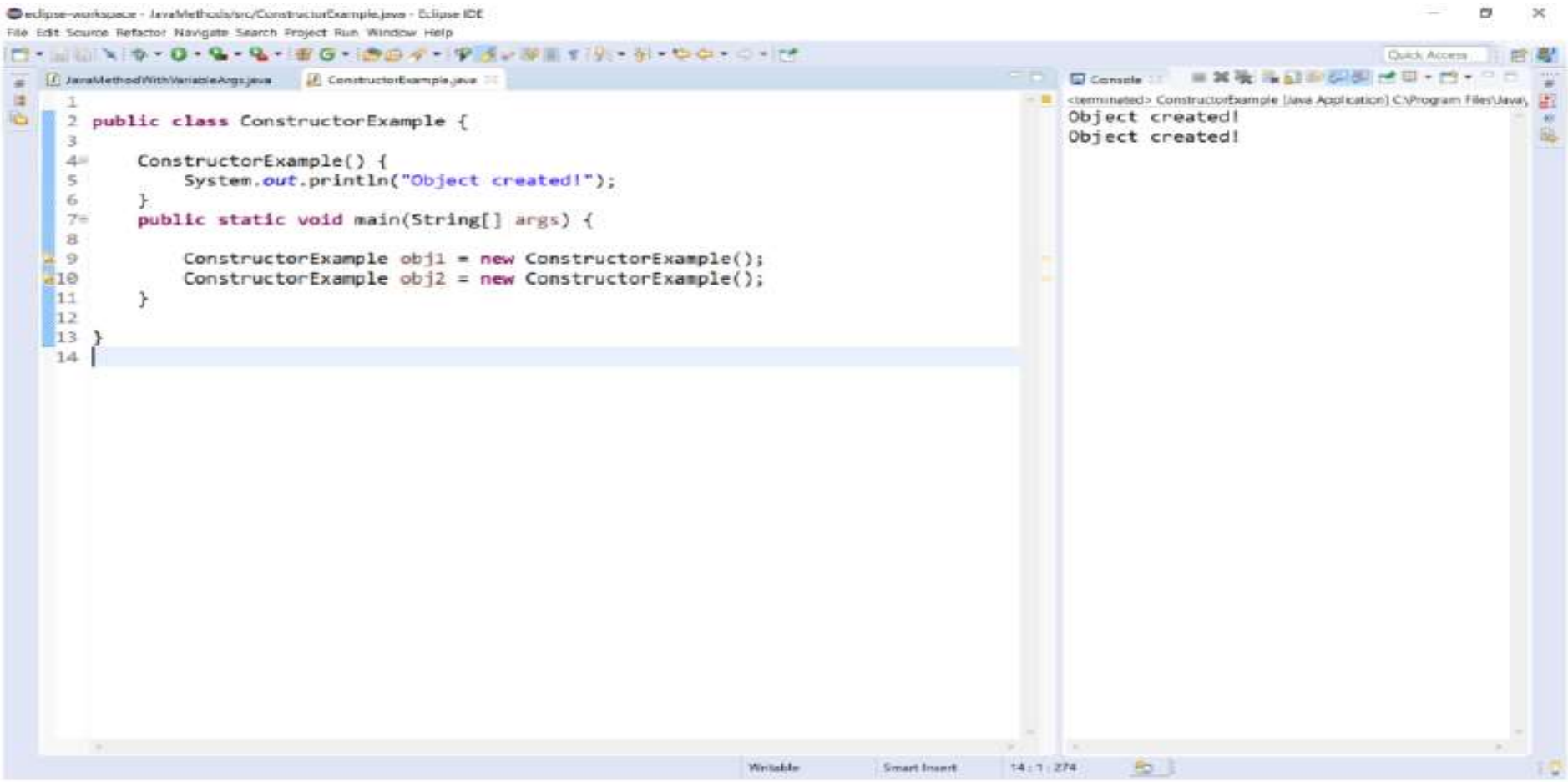
```
<returnType> <methodName>(dataType...parameterName);
```



When a method has both the normal parameter and variable-argument, then the variable argument must be specified at the end in the parameters list.

Constructor

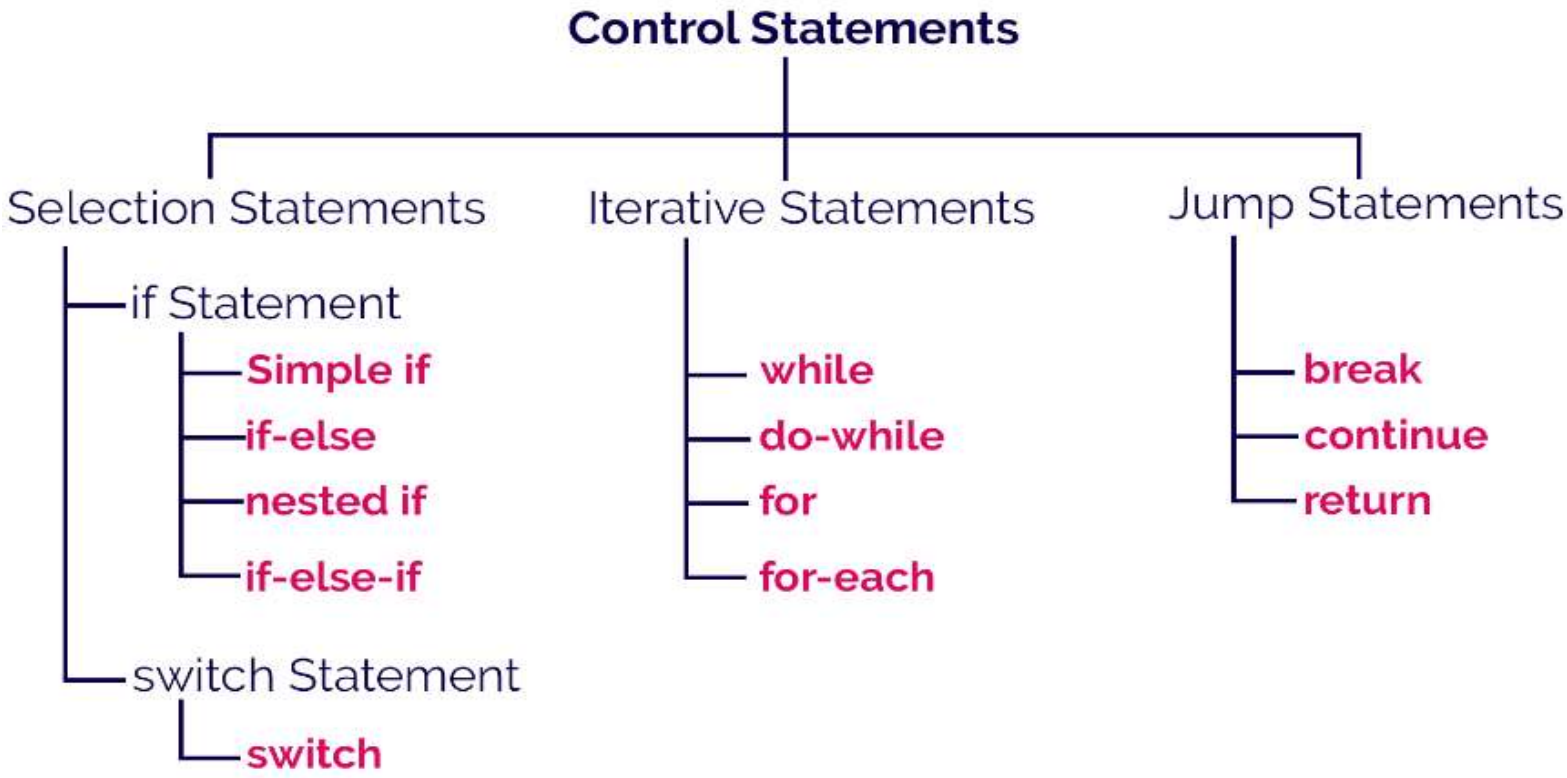
A constructor is a special method of a class that has the same name as the class name. The constructor gets executes automatically on object creation. It does not require the explicit method call. A constructor may have parameters and access specifiers too. In java, if you do not provide any constructor the compiler automatically creates a default constructor.



A constructor can not have return value.

Control Statements/Structures

In java, the default execution flow of a program is a sequential order. But the sequential order of execution flow may not be suitable for all situations. Sometimes, we may want to jump from line to another line, we may want to skip a part of the program, or sometimes we may want to execute a part of the program again and again. To solve this problem, java provides control statements.



In java, the control statements are the statements which will tell us that in which order the instructions are getting executed. The control statements are used to control the order of execution according to our requirements. Java provides several control statements, and they are classified as follows.

Types of Control Statements

In java, the control statements are classified as follows.

* Selection Control Statements (Decision Making Statements)
* Iterative Control Statements (Looping Statements)
* Jump Statements

Let's look at each type of control statements in java.

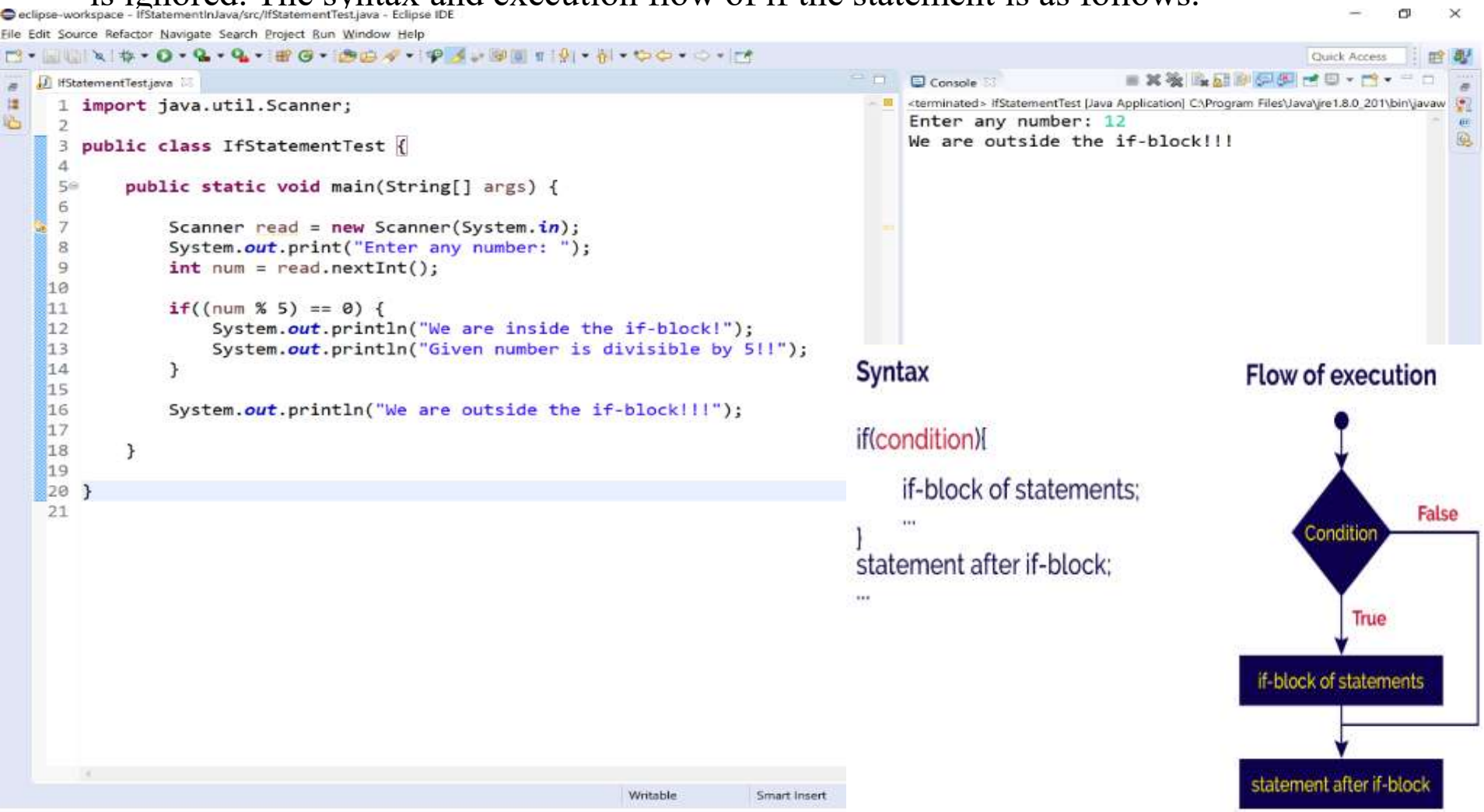
Selection Control Statements

In java, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be executed based on a condition. Java provides the following selection statements.

* if statement
* if-else statement
* if-elif statement
* nested if statement
* switch statement

IF STATEMENT IN JAVA

In java, we use the if statement to test a condition and decide the execution of a block of statements based on that condition result. The if statement checks, the given condition then decides the execution of a block of statements. If the condition is True, then the block of statements is executed and if it is False, then the block of statements is ignored. The syntax and execution flow of if the statement is as follows.



IF-ELSE STATEMENT IN JAVA

In java, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result. The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result. If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed. The syntax and execution flow of if-else statement is as follows.

eclipse-workspace - IfStatementInJava/src/IfElseStatementTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

IfElseStatementTest.java

```
1  import java.util.Scanner;
2
3  public class IfElseStatementTest {
4
5      public static void main(String[] args) {
6
7          Scanner read = new Scanner(System.in);
8          System.out.print("Enter any number: ");
9          int num = read.nextInt();
10
11         if((num % 2) == 0) {
12             System.out.println("We are inside the true-block!");
13             System.out.println("Given number is EVEN number!!");
14         }
15         else {
16             System.out.println("We are inside the false-block!");
17             System.out.println("Given number is ODD number!!");
18         }
19
20         System.out.println("We are outside the if-block!!!");
21
22     }
23
24 }
25
```

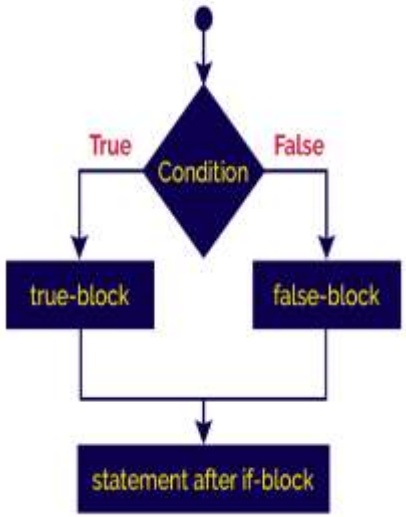
Console

```
<terminated> IfElseStatementTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\ja
Enter any number: 16
We are inside the true-block!
Given number is EVEN number!!
We are outside the if-block!!!
```

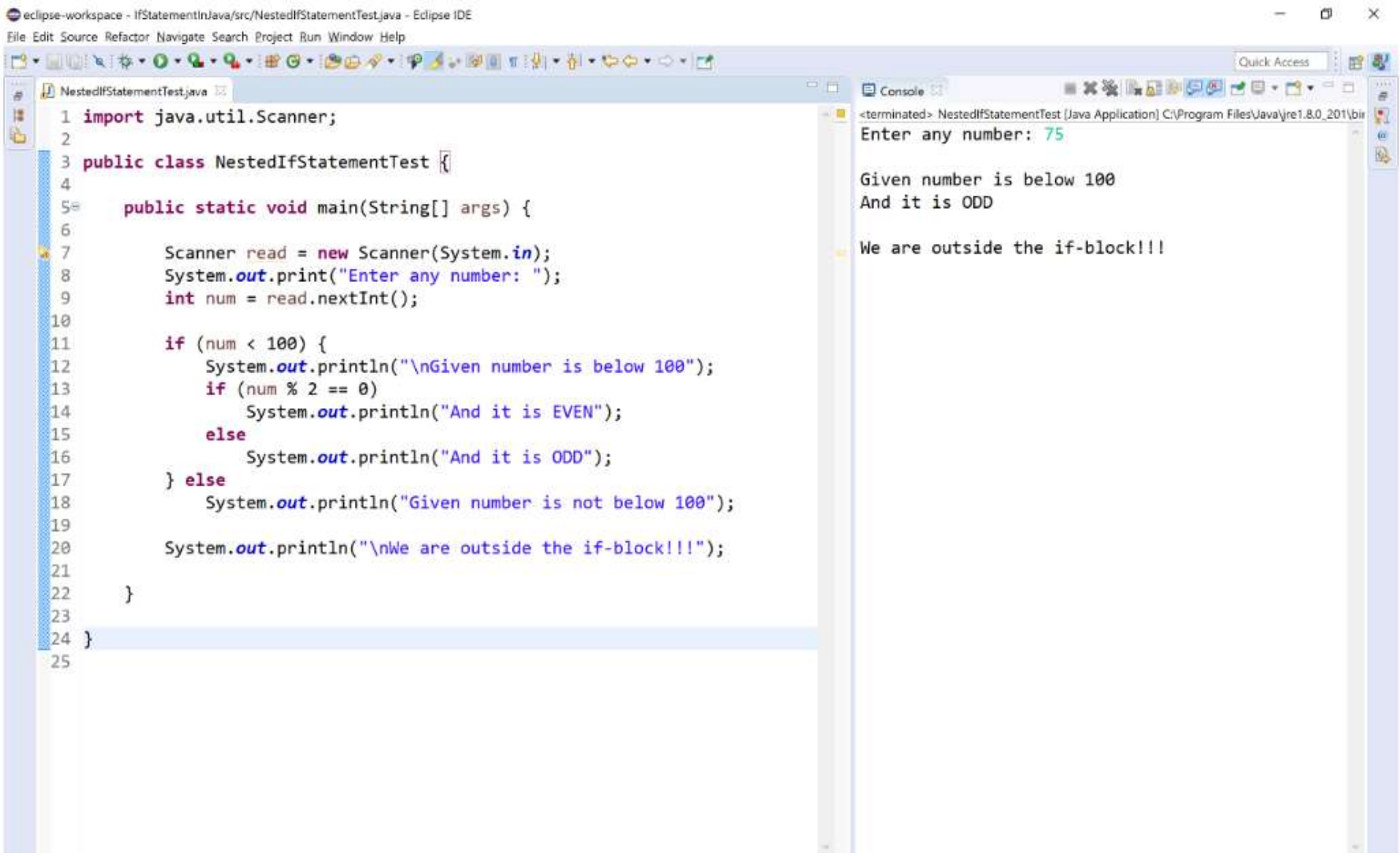
Syntax

```
if(condition){
    true-block of statements;
    ...
}
else{
    false-block of statements;
    ...
}
statement after if-block;
...
```

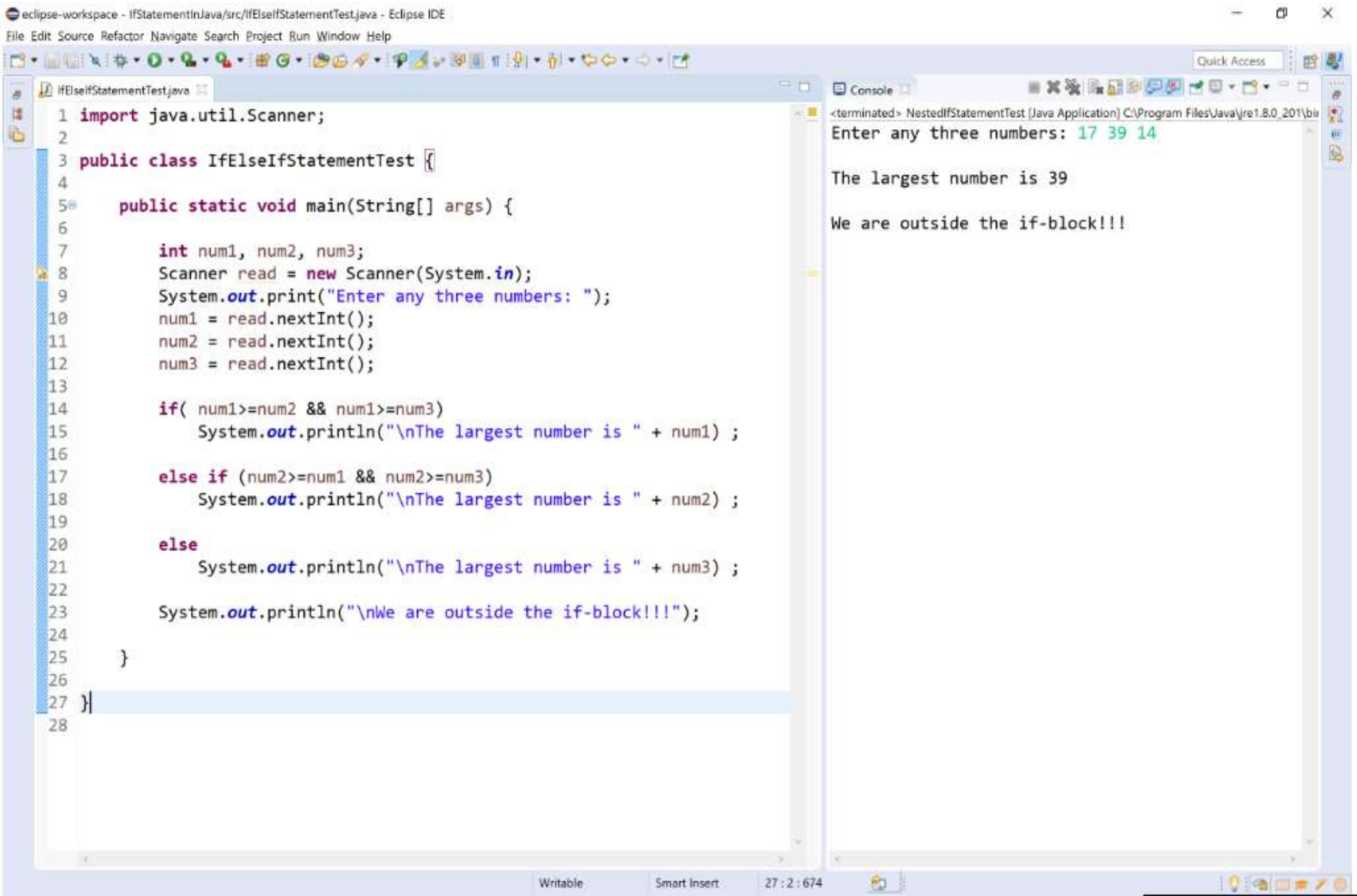
Flow of execution



NESTED IF STATEMENT IN JAVA



IF-ELSE IF STATEMENT IN JAVA



SWITCH STATEMENT IN JAVA

Using the switch statement, one can select only one option from more number of options very easily. In the switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches the value associated with an option, the execution starts from that option. In the switch statement, every option is defined as a **case**.

The switch statement has the following syntax and execution flow diagram.

eclipse-workspace - IfStatementInJava/src/SwitchStatementTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

SwitchStatementTest.java

```
1  import java.util.Scanner;
2
3  public class SwitchStatementTest {
4
5      public static void main(String[] args) {
6
7          Scanner read = new Scanner(System.in);
8          System.out.print("Press any digit: ");
9
10         int value = read.nextInt();
11
12         switch( value )
13            {
14             case 0: System.out.println("ZERO") ; break ;
15             case 1: System.out.println("ONE") ; break ;
16             case 2: System.out.println("TWO") ; break ;
17             case 3: System.out.println("THREE") ; break ;
18             case 4: System.out.println("FOUR") ; break ;
19             case 5: System.out.println("FIVE") ; break ;
20             case 6: System.out.println("SIX") ; break ;
21             case 7: System.out.println("SEVEN") ; break ;
22             case 8: System.out.println("EIGHT") ; break ;
23             case 9: System.out.println("NINE") ; break ;
24             default: System.out.println("Not a Digit") ;
25            }
26
27     }
28
29 }
30
```

Writable

Console

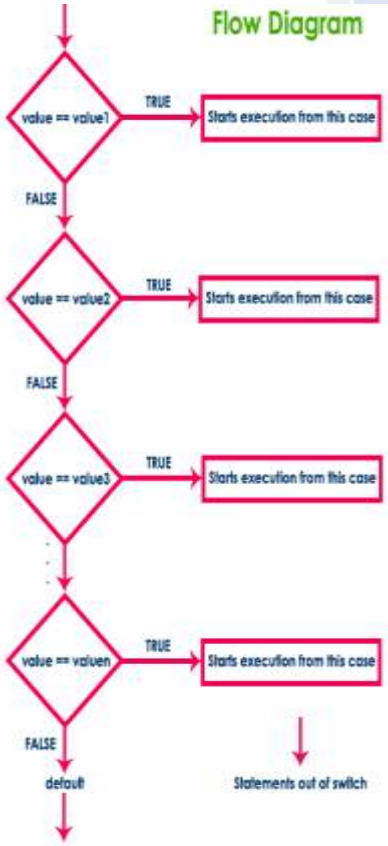
<terminated> SwitchStatementTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\

```
Press any digit: 8
EIGHT
```

Syntax

```
switch ( expression or value )
{
    case value1: set of statements;
                 ....
    case value2: set of statements;
                 ....
    case value3: set of statements;
                 ....
    case value4: set of statements;
                 ....
    case value5: set of statements;
                 ....
    .
    .
    default: set of statements;
}
```

Flow Diagram



Iterative Control Statements

In java, the iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as the given condition is True. Using iterative statements reduces the size of the code, reduces the code complexity, makes it more efficient, and increases the execution speed. Java provides the following iterative statements.

* while statement
* do-while statement
* for statement
* for-each statement

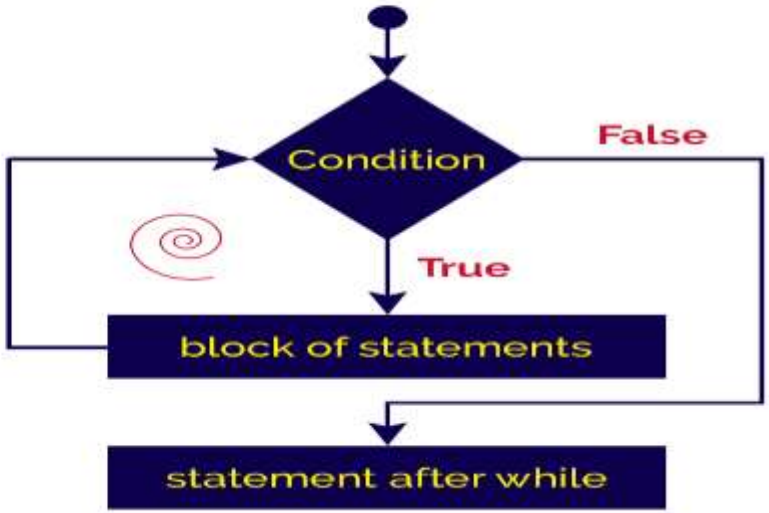
WHILE STATEMENT IN JAVA

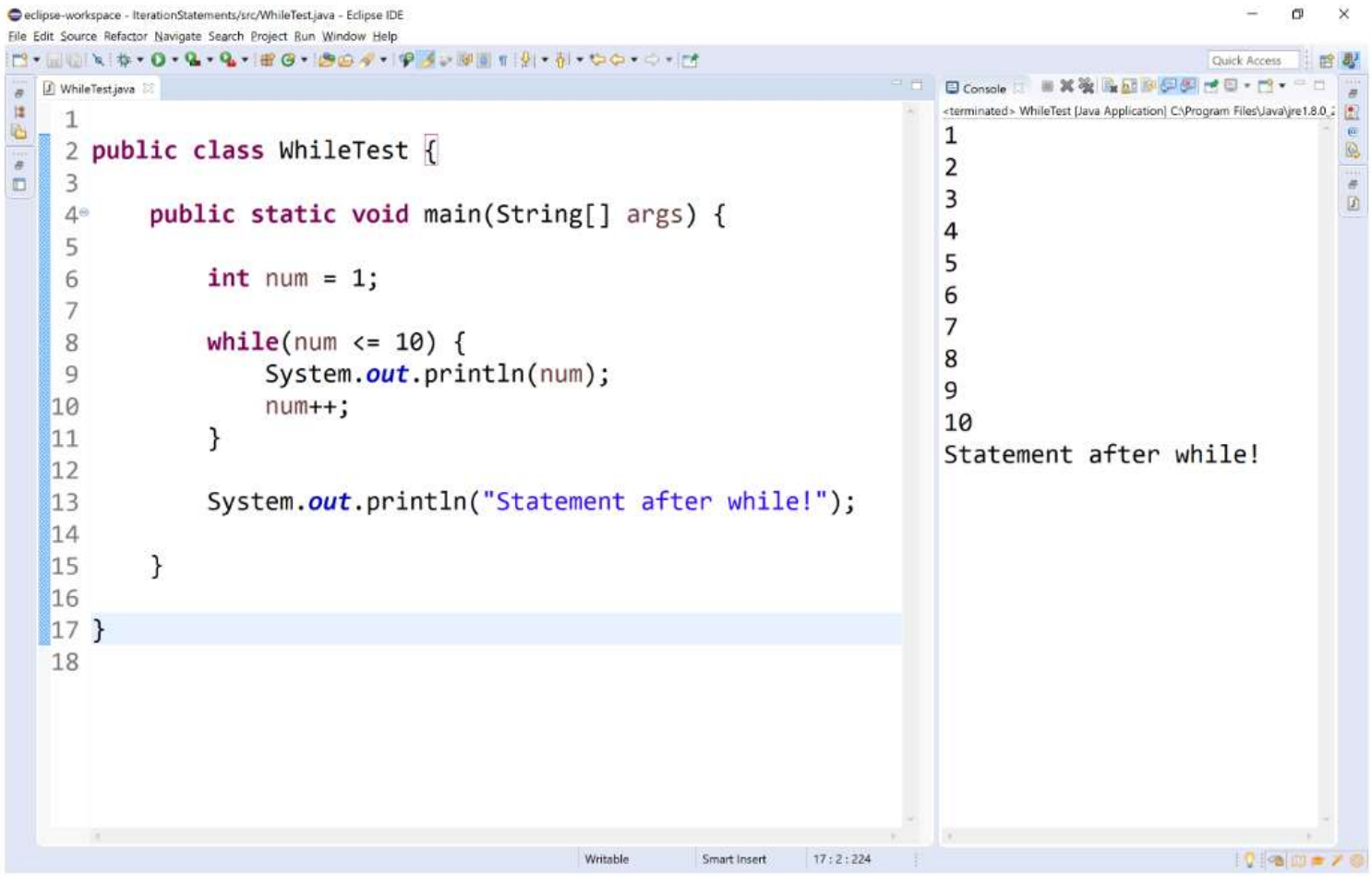
The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement. The syntax and execution flow of while statement is as follows.

Syntax

```
while(boolean-expression){
        block of statements;
        ...
}
statement after while;
...
```

Flow of execution




eclipse-workspace - IterationStatements/src/WhileTest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
WhileTest.java
1
2 public class WhileTest {
3
4 public static void main(String[] args) {
5
6 int num = 1;
7
8 while(num <= 10) {
9 System.out.println(num);
10 num++;
11 }
12
13 System.out.println("Statement after while!");
14
15 }
16
17 }
18
Console
<terminated> WhileTest [Java Application] C:\Program Files\Java\jre1.8.0_2
1
2
3
4
5
6
7
8
9
10
Statement after while!
Writable
Smart Insert
17 : 2 : 224

DO-WHILE STATEMENT IN JAVA

The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as the **Exit control looping statement**. The do-while statement has the following syntax.

eclipse-workspace - IterationStatements/src/DoWhileTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

DoWhileTest.java

```
1
2  public class DoWhileTest {
3
4      public static void main(String[] args) {
5
6          int num = 1;
7
8          do {
9              System.out.println(num);
10             num++;
11         }while(num <= 10);
12
13         System.out.println("Statement after do-while!");
14
15     }
16
17 }
18
```

Console

<terminated> DoWhileTest [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe

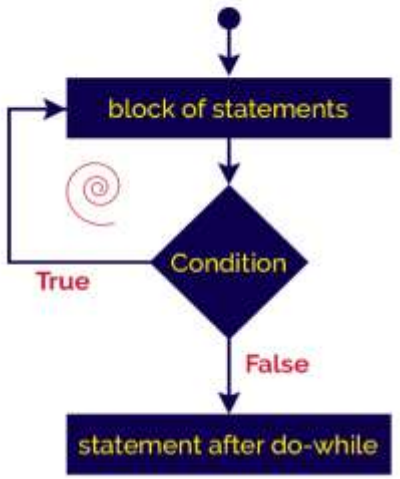
```
1
2
3
4
5
6
7
8
9
10
Statement after do-while!
```

Writable Smart Insert 17 : 2 : 232

Syntax

```
do{
      block of statements;
      ...
}while(boolean-expression);
statement after do-while;
...
```

Flow of execution



FOR STATEMENT IN JAVA

The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE. The for statement has the following syntax and execution flow diagram.

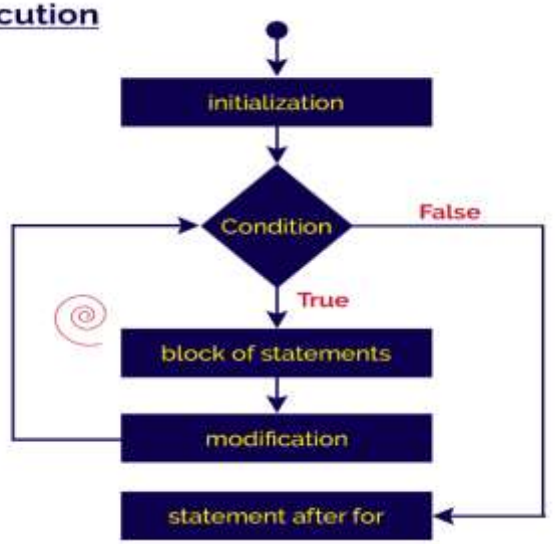
```
1
2  public class ForTest {
3
4      public static void main(String[] args) {
5
6          for(int i = 0; i < 10; i++) {
7              System.out.println("i = " + i);
8          }
9
10         System.out.println("Statement after for!");
11     }
12
13 }
14
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
Statement after for!
```

Syntax

```
for(initialization; boolean-expression; modofication){
    block of statements;
    ...
}
statement after for;
...
```

Flow of execution



FOR-EACH STATEMENT IN JAVA

The Java for-each statement was introduced since Java 5.0 version. It provides an approach to traverse through an array or collection in Java. The for-each statement also known as **enhanced for** statement. The for-each statement executes the block of statements for each element of the given array or collection.

eclipse-workspace - IterationStatements/src/ForEachTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

EnumExample.java | ForEachTest.java | WhileTest.java

```
1
2 public class ForEachTest {
3
4     public static void main(String[] args) {
5
6         int[] arrayList = {10, 20, 30, 40, 50};
7
8         for(int i : arrayList) {
9             System.out.println("i = " + i);
10        }
11
12        System.out.println("Statement after for-each!");
13    }
14
15 }
```

Console

<terminated> ForEachTest [Java Application] C:\Program Files\Java\jre1.8

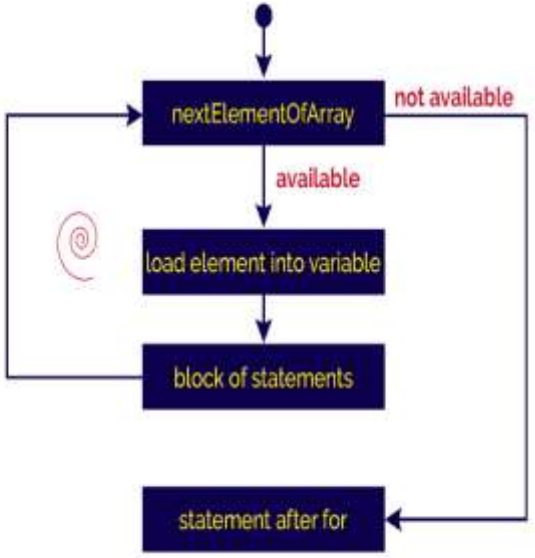
```
i = 10
i = 20
i = 30
i = 40
i = 50
Statement after for-each!
```

Syntax

```
for( dataType  variableName : Array ){
        block of statements;
        ...
}
statement after for;
...
```

Writable Smart Insert

Flow of execution



Java Jump Statements

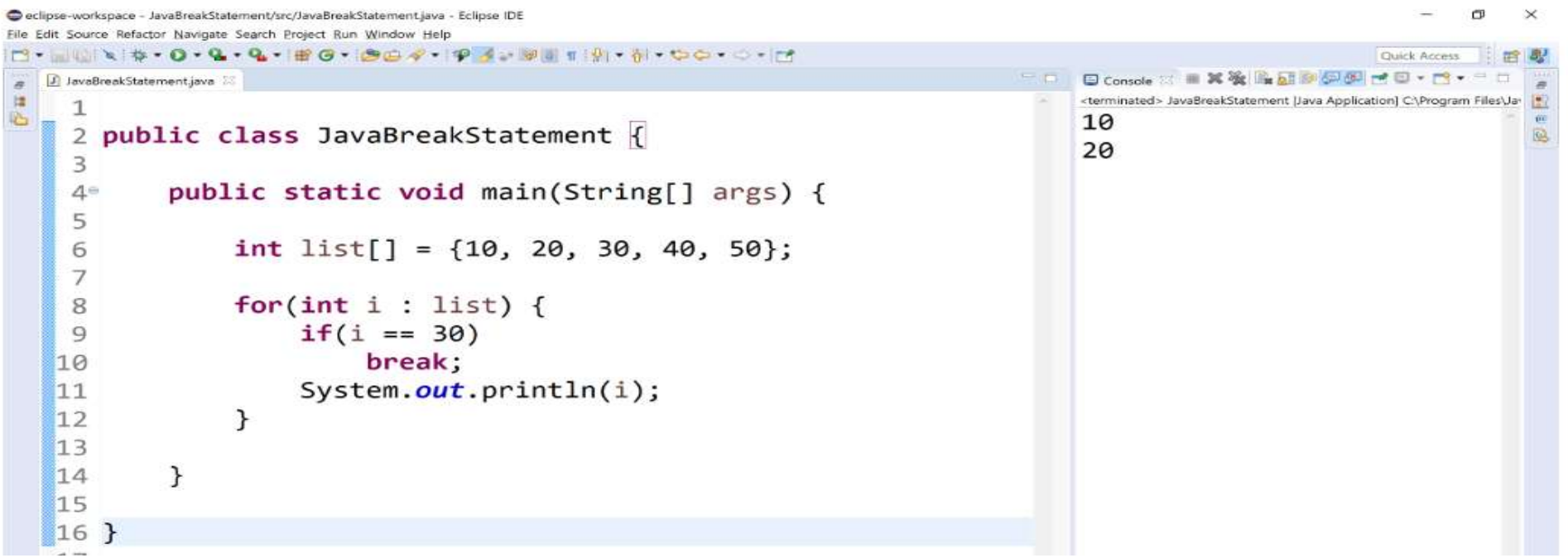
The java programming language supports jump statements that used to transfer execution control from one line to another line. The java programming language provides the following jump statements.

- break statement
- continue statement
- labelled break and continue statements
- return statement

break statement in java

The break statement in java is used to terminate a switch or looping statement. That means the break statement is used to come out of a switch statement and a looping statement like while, do-while, for, and for-each.

Using the break statement outside the switch or loop statement is not allowed.



The floowing picture depictes the execution flow of the break statement.

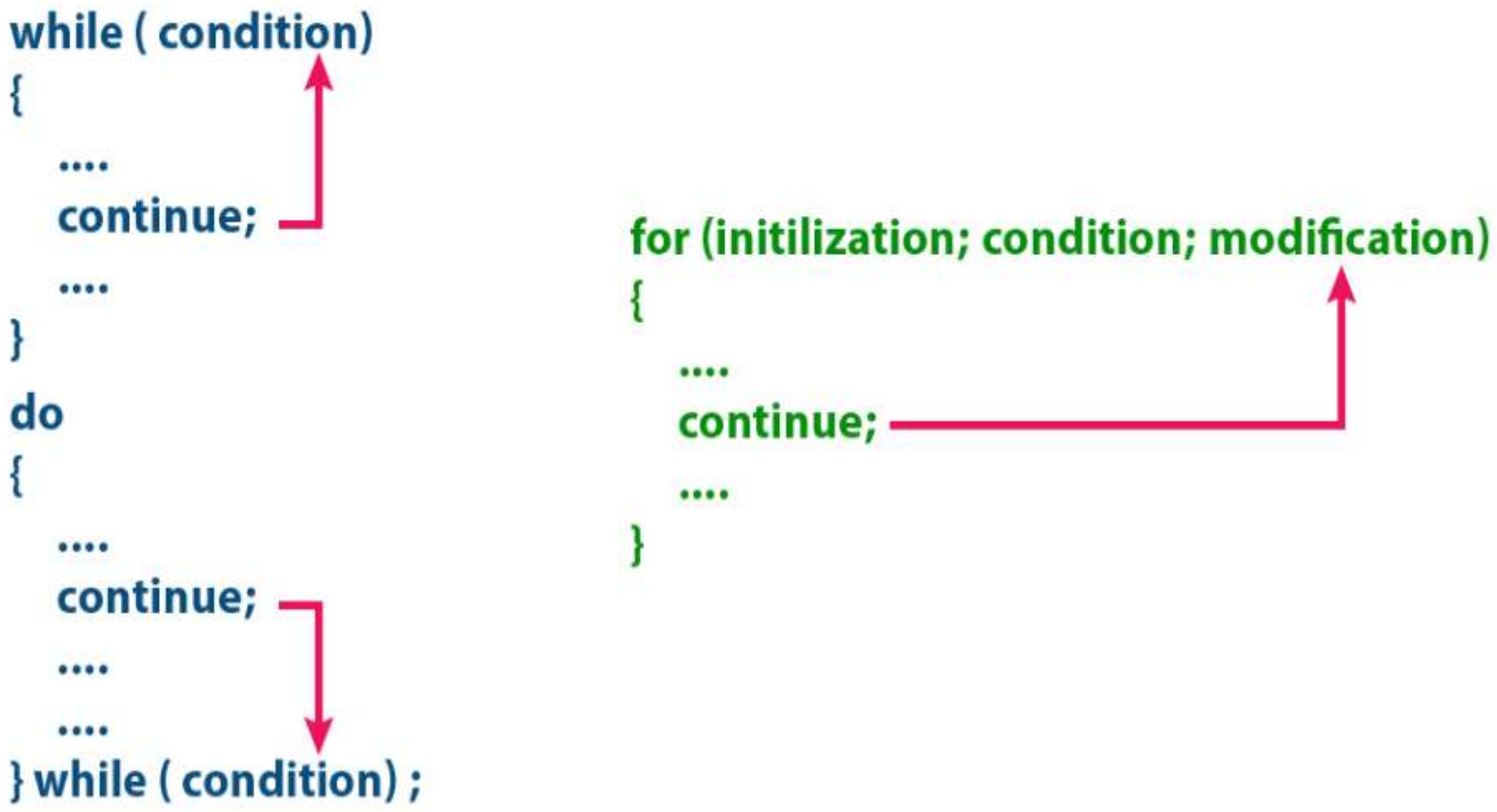
```
while ( condition)
{
   ....
   break ;
   ....
}

do
{
   ....
   break ;
   ....
} while ( condition) ;
```

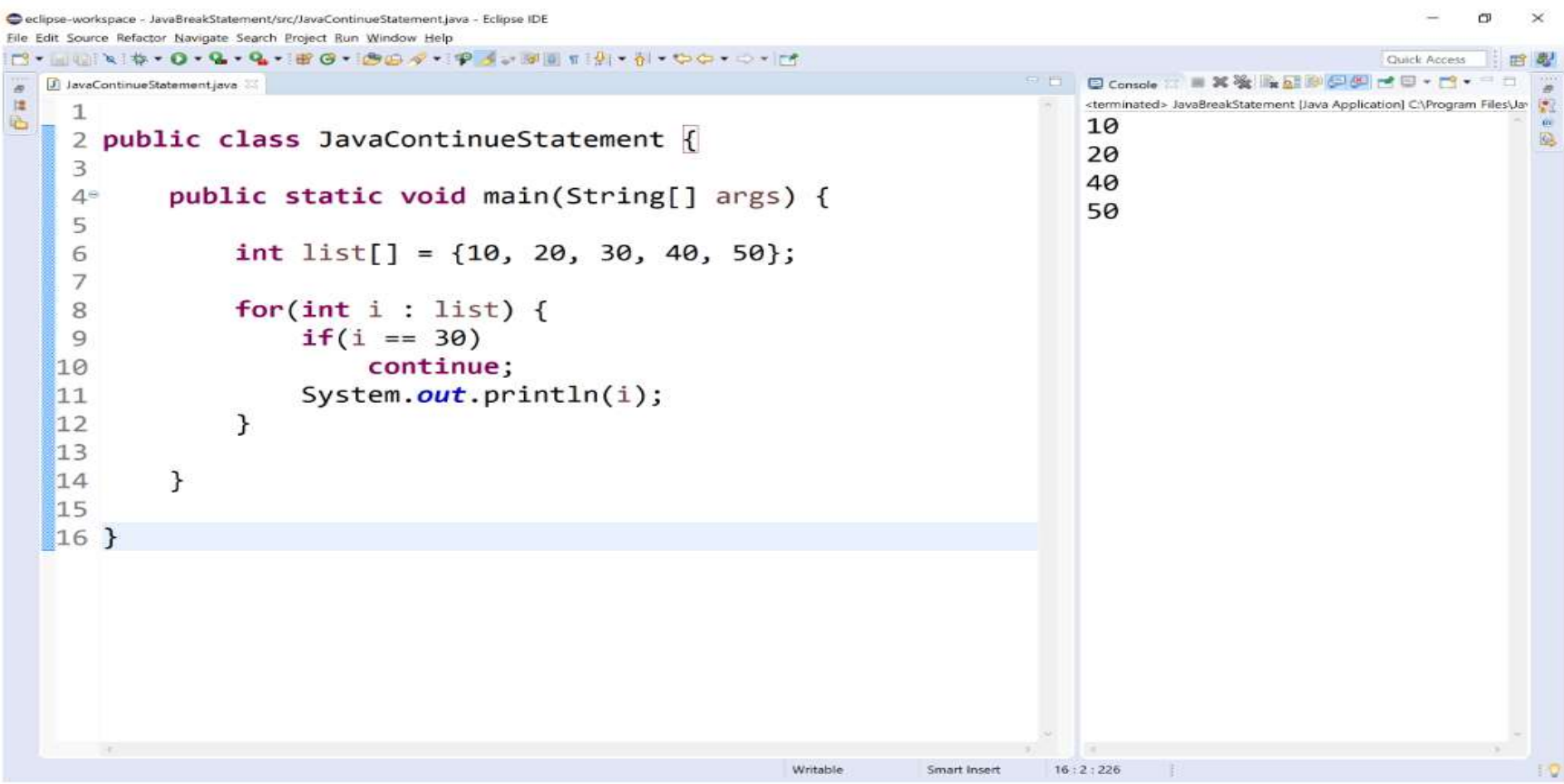
```
for (initilization; condition; modification)
{
   ....
   break ;
   ....
}
```

continue statement in java

The continue statement is used to move the execution control to the beginning of the looping statement. When the continue statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The continue statement can be used with looping statements like while, do-while, for, and for-each.

When we use continue statement with while and do-while statements, the execution control directly jumps to the condition. When we use continue statement with for statement the execution control directly jumps to the modification portion (increment/decrement/any modification) of the for loop. The continue statement flow of execution is as shown in the following figure.

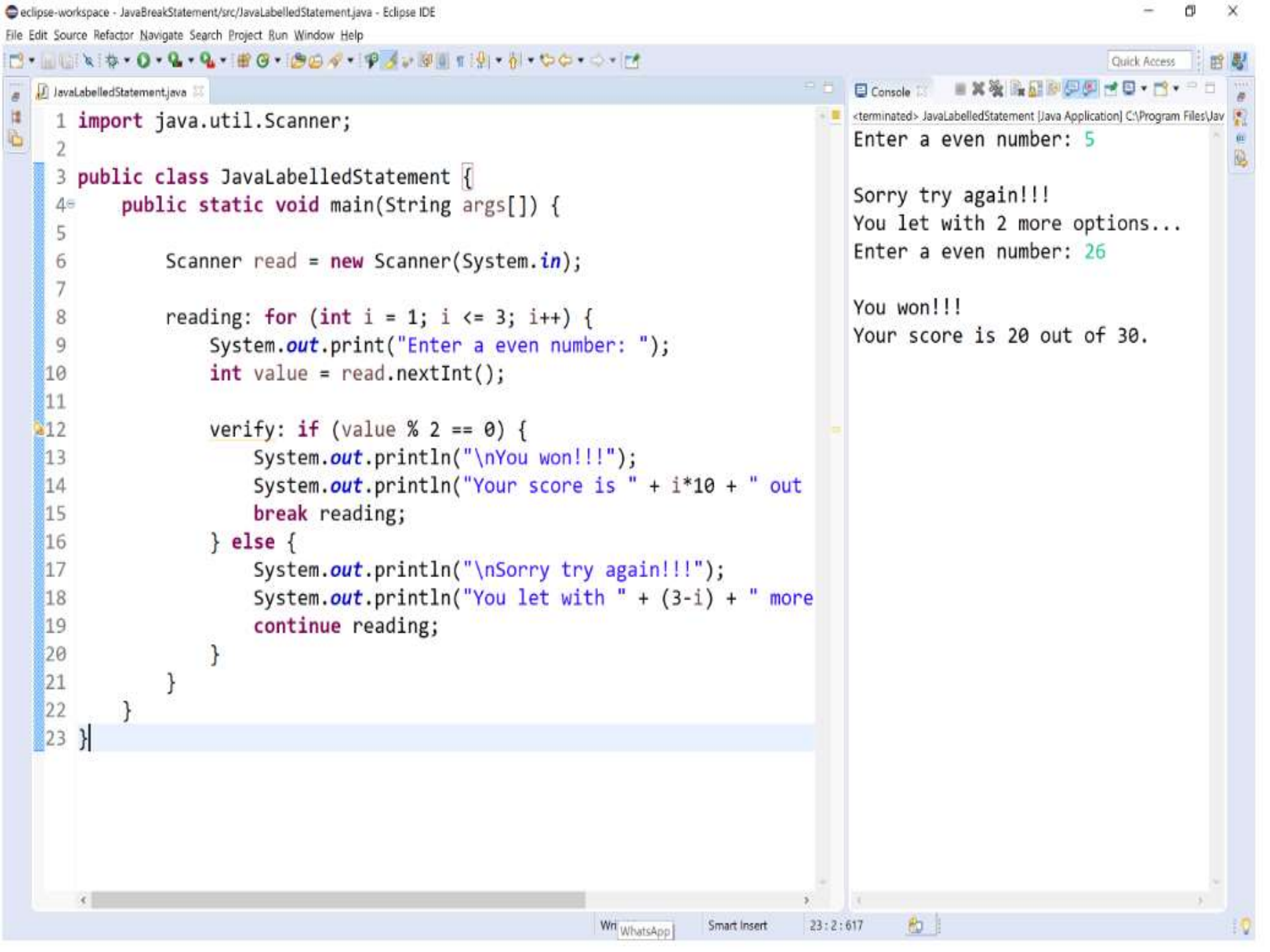




Labelled break and continue statement in java

The java programming langauge does not support `goto` statement, alternatively, the break and continue statements can be used with label.

The labelled break statement terminates the block with specified label. The labbeled contonue statement takes the execution control to the beginning of a loop with specified label.


eclipse-workspace - JavaBreakStatement/src/JavaLabelledStatement.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
JavaLabelledStatement.java
1 import java.util.Scanner;
2
3 public class JavaLabelledStatement {
4 public static void main(String args[]) {
5
6 Scanner read = new Scanner(System.in);
7
8 reading: for (int i = 1; i <= 3; i++) {
9 System.out.print("Enter a even number: ");
10 int value = read.nextInt();
11
12 verify: if (value % 2 == 0) {
13 System.out.println("\nYou won!!!");
14 System.out.println("Your score is " + i*10 + " out
15 break reading;
16 } else {
17 System.out.println("\nSorry try again!!!");
18 System.out.println("You let with " + (3-i) + " more
19 continue reading;
20 }
21 }
22 }
23 }
Console
<terminated> JavaLabelledStatement [Java Application] C:\Program Files\Jav
Enter a even number: 5
Sorry try again!!!
You let with 2 more options...
Enter a even number: 26
You won!!!
Your score is 20 out of 30.
Smart Insert
23 : 2 : 617

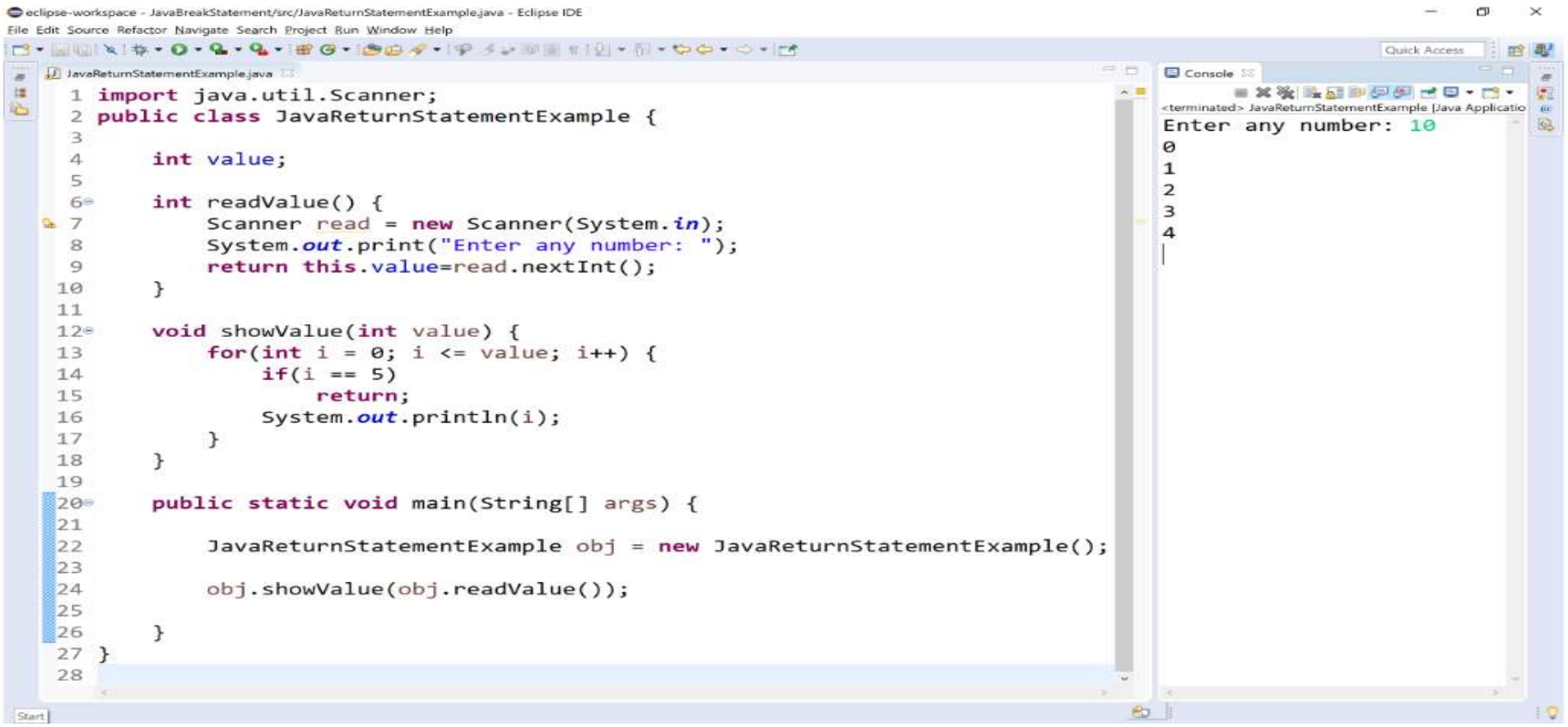
return statement in java

In java, the return statement used to terminate a method with or without a value. The return statement takes the execution control to the calling function. That means the return statement transfer the execution control from called function to the calling function by carrying a value.

> 🔔 Java allows the use of return-statement with both, with and without return type methods.

In java, the return statement used with both methods with and without return type. In the case of a method with the return type, the return statement is mandatory, and it is optional for a method without return type.

When a return statement used with a return type, it carries a value of return type. But, when it is used without a return type, it does not carry any value. Instead, simply transfers the execution control.



Polymorphism

The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.

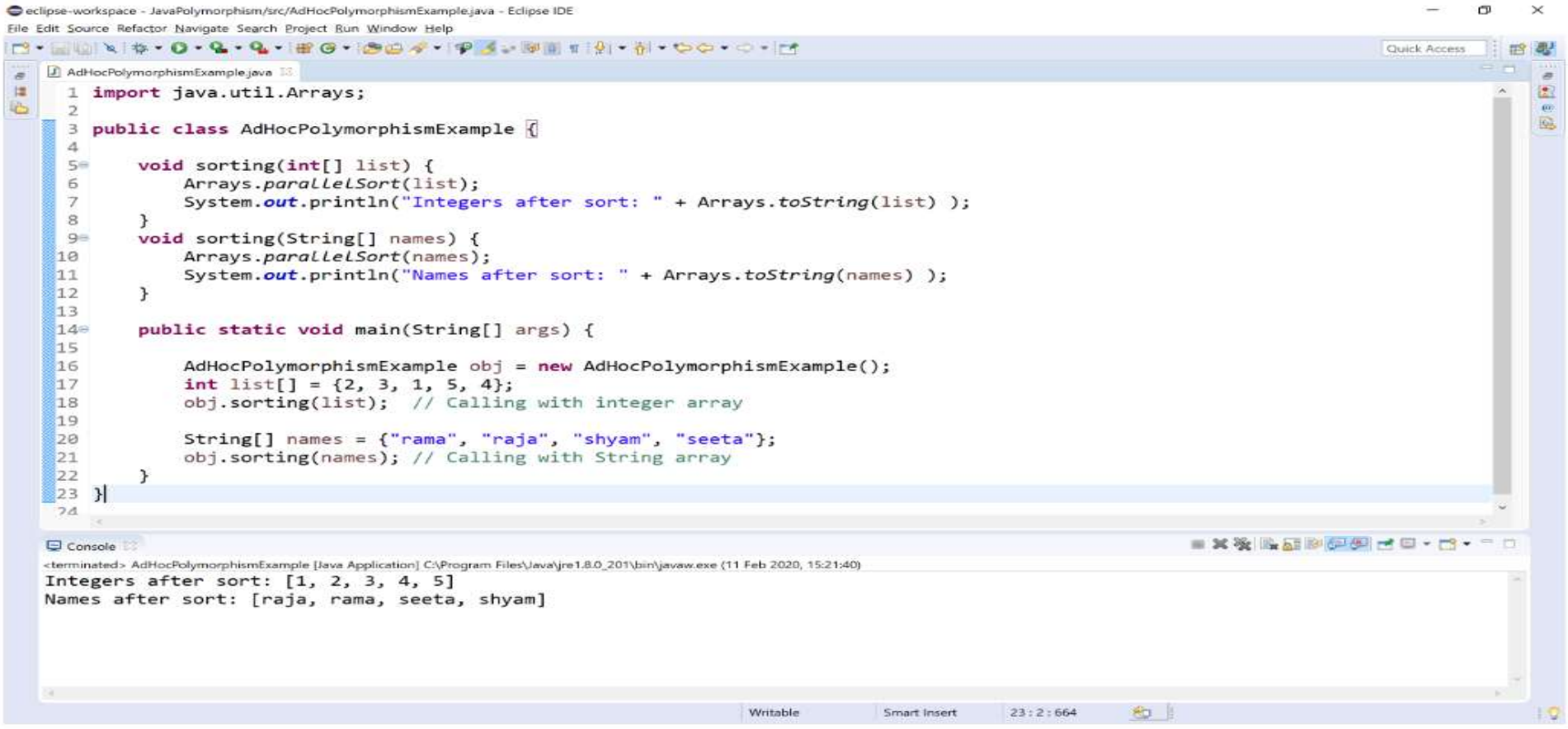
In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept.

In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments.

The ad hoc polymorphism implemented within the class only.



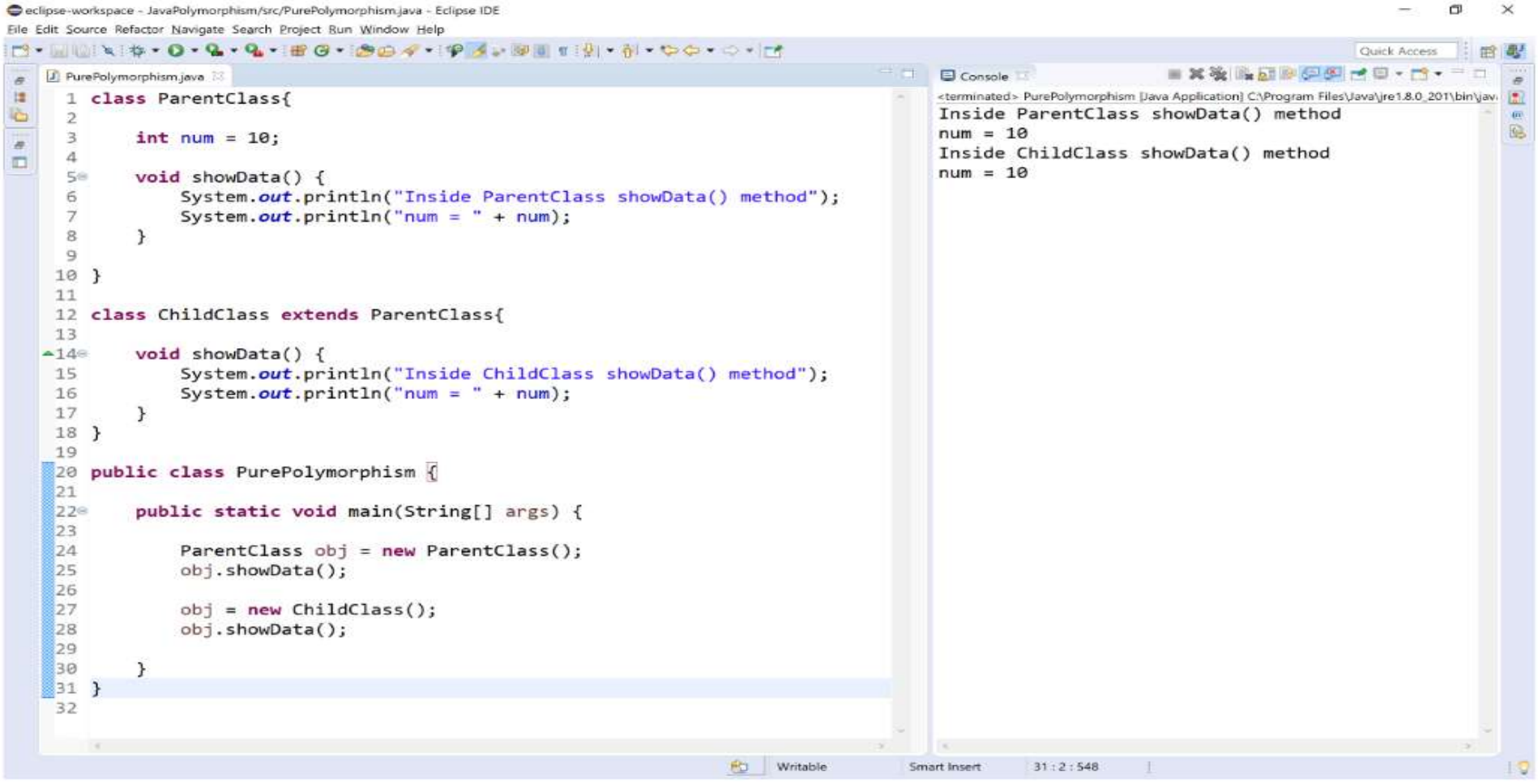
Pure polymorphism

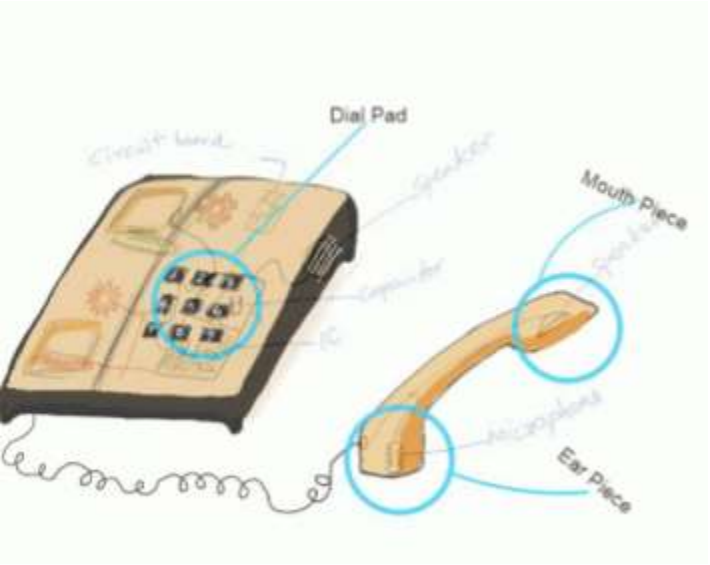
The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept.

In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference.

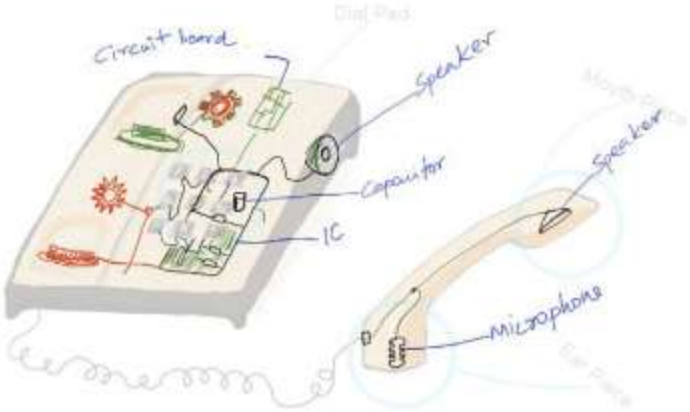
When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.

The pure polymorphism implemented in the inheritance concept only.





Abstraction



An abstract class is a class that created using abstract keyword. In other words, a class prefixed with abstract keyword is known as an abstract class.

In java, an abstract class may contain abstract methods (methods without implementation) and also non-abstract methods (methods with implementation).

We use the following syntax to create an abstract class.

Syntax

```
abstract class <ClassName>{
    ...
}
```

AbstractClassExample.java

```
1  import java.util.*;
2
3  abstract class Shape {
4      int length, breadth, radius;
5
6      Scanner input = new Scanner(System.in);
7
8      abstract void printArea();
9
10 }
11
12 class Rectangle extends Shape {
13     void printArea() {
14         System.out.println("*** Finding the Area of Rectangle ***");
15         System.out.print("Enter length and breadth: ");
16         length = input.nextInt();
17         breadth = input.nextInt();
18         System.out.println("The area of Rectangle is: " + length * breadth);
19     }
20 }
21
22 class Triangle extends Shape {
23     void printArea() {
24         System.out.println("\n*** Finding the Area of Triangle ***");
25         System.out.print("Enter Base And Height: ");
26         length = input.nextInt();
27         breadth = input.nextInt();
28         System.out.println("The area of Triangle is: " + (length * breadth) / 2);
29     }
30 }
31
32 class Cricle extends Shape {
33     void printArea() {
```

Console

```
<terminated> AbstractClassExample [Java Application] C:\Program Files\Jav
*** Finding the Area of Rectangle ***
Enter length and breadth: 2 4
The area of Rectangle is: 8

*** Finding the Area of Triangle ***
Enter Base And Height: 4 5
The area of Triangle is: 10

*** Finding the Area of Cricle ***
Enter Radius: 6
The area of Cricle is: 113.04
```

Writable | Smart Insert | 52 : 2 : 1298

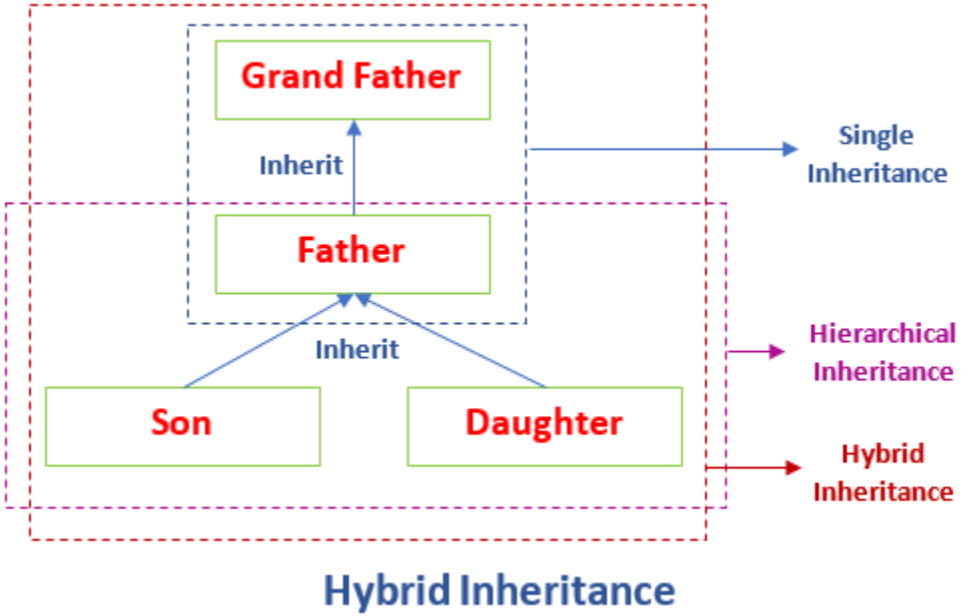
An abstract class can not be instantiated but can be referenced. That means we can not create an object of an abstract class, but base reference can be created.

8 Rules for method overriding

An abstract class must follow the below list of rules.

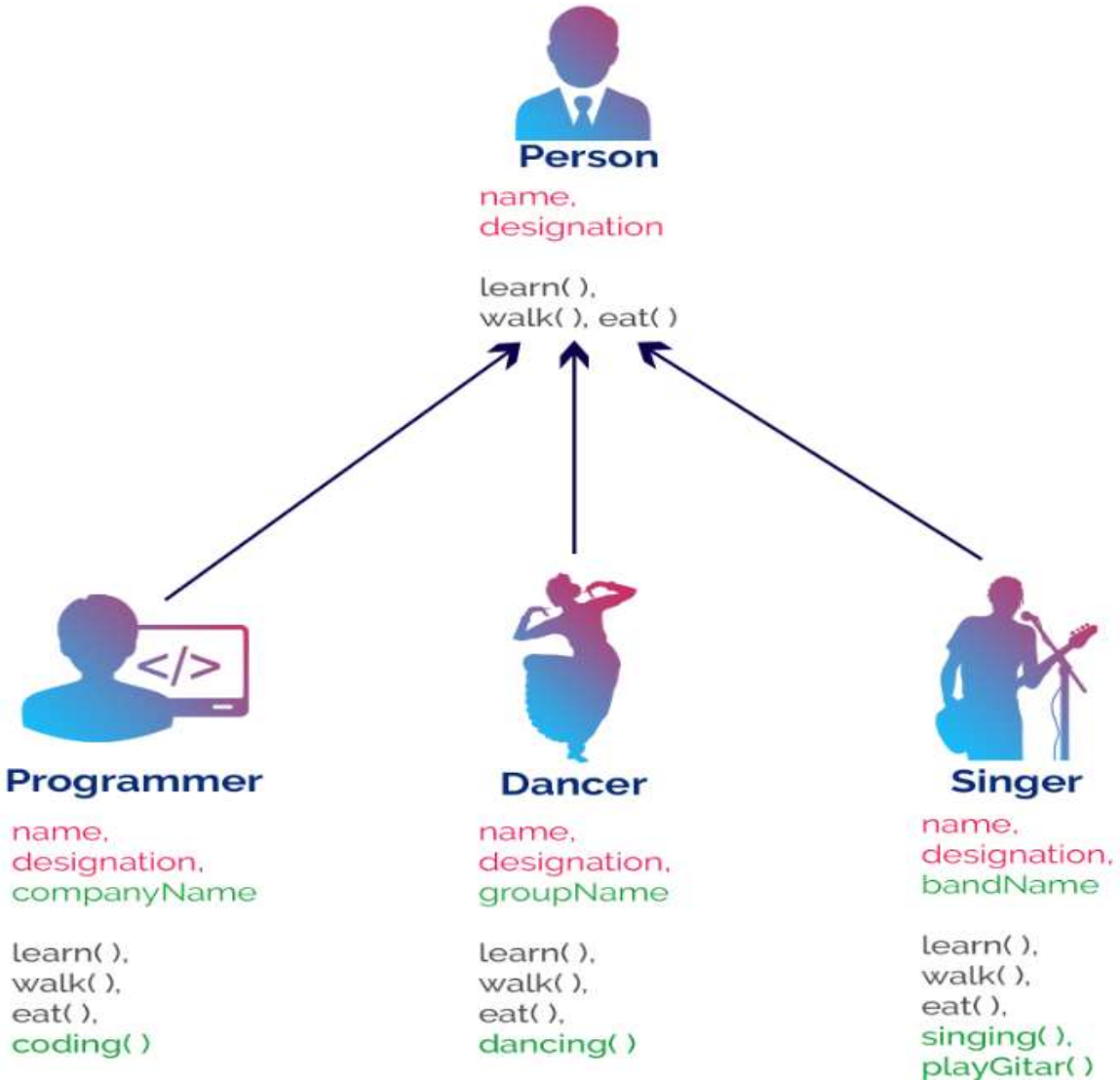
- An abstract class must be created with abstract keyword.
- An abstract class can be created without any abstract method.
- An abstract class may contain abstract methods and non-abstract methods.
- An abstract class may contain final methods that can not be overridden.
- An abstract class may contain static methods, but the abstract method can not be static.
- An abstract class may have a constructor that gets executed when the child class object created.
- An abstract method must be overridden by the child class, otherwise, it must be defined as an abstract class.
- An abstract class can not be instantiated but can be referenced.

Inheritance



Inheritance Concept

The inheritance is a very useful and powerful concept of object-oriented programming. In java, using the inheritance concept, we can use the existing features of one class in another class. The inheritance provides a greate advantage called code re-usability. With the help of code re-usability, the commonly used code in an application need not be written again and again.



The inheritance can be defined as follows.

> The inheritance is the process of acquiring the properties of one class to another class.

Inheritance Basics

In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.

The `Parent class` is the class which provides features to another class. The parent class is also known as `Base class` or `Superclass`.

The `Child class` is the class which receives features from another class. The child class is also known as the `Derived Class` or `Subclass`.

> In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

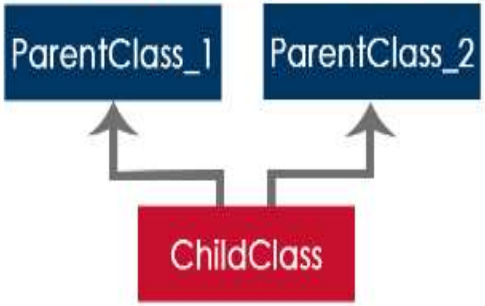
There are five types of inheritances, and they are as follows.

- Simple Inheritance (or) Single Inheritance
- Multiple Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

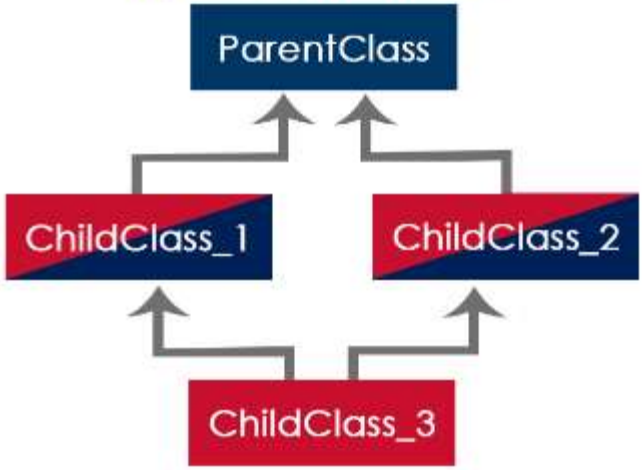
The following picture illustrates how various inheritances are implemented.

Simple Inheritance

Multiple Inheritance

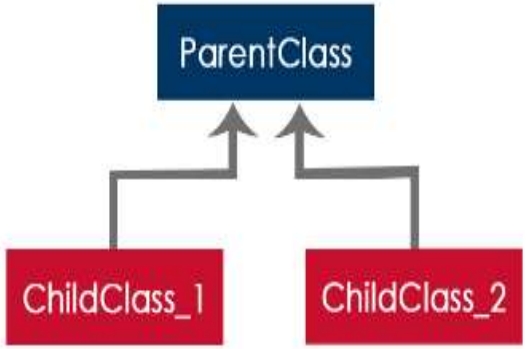


Hybrid Inheritance



Multi Level Inheritance

Hierarchical Inheritance



The java programming language does not support multiple inheritance type. However, it provides an alternate with the concept of interfaces.

Creating Child Class in java

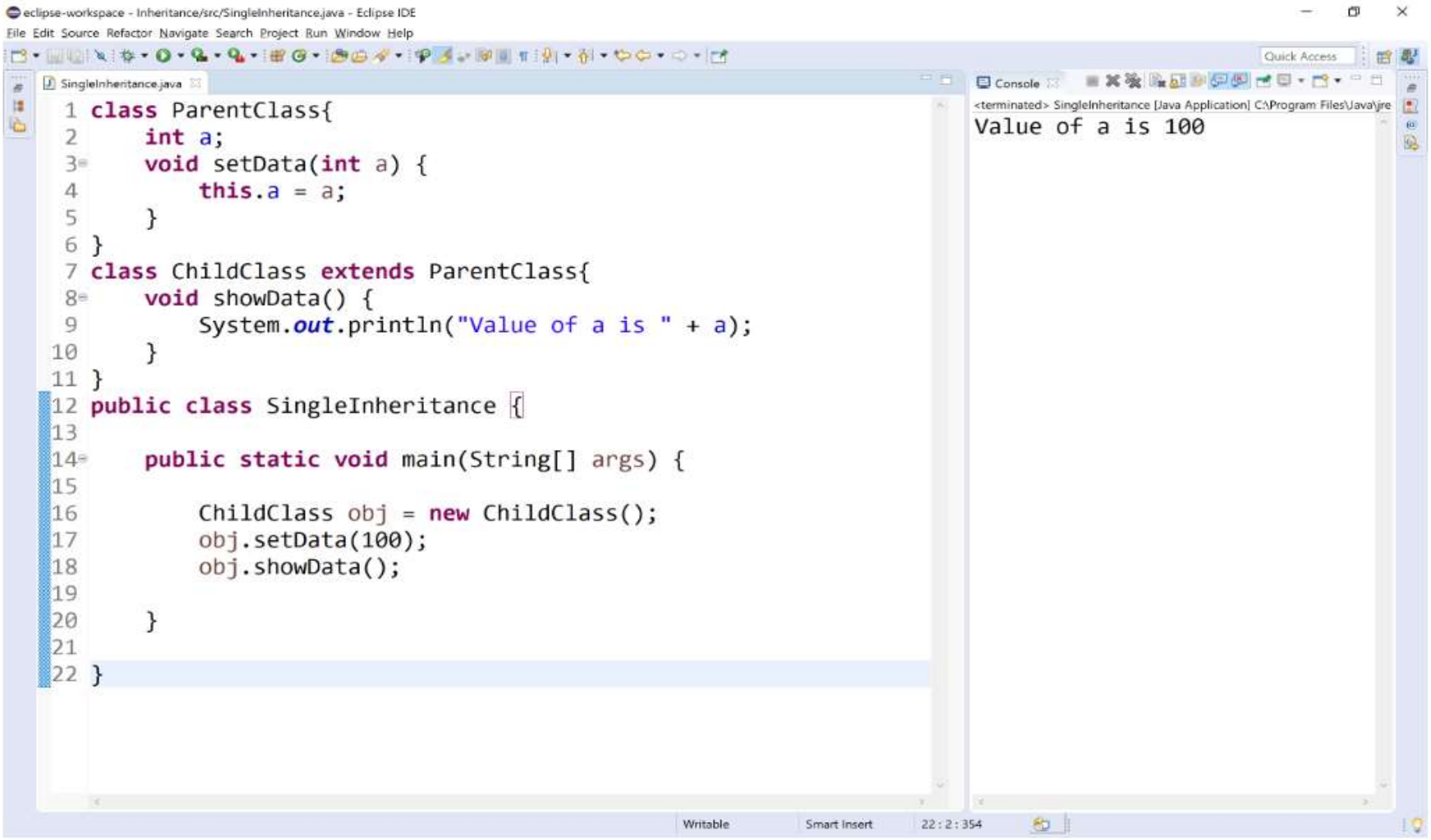
In java, we use the keyword `extends` to create a child class. The following syntax used to create a child class in java.

Syntax

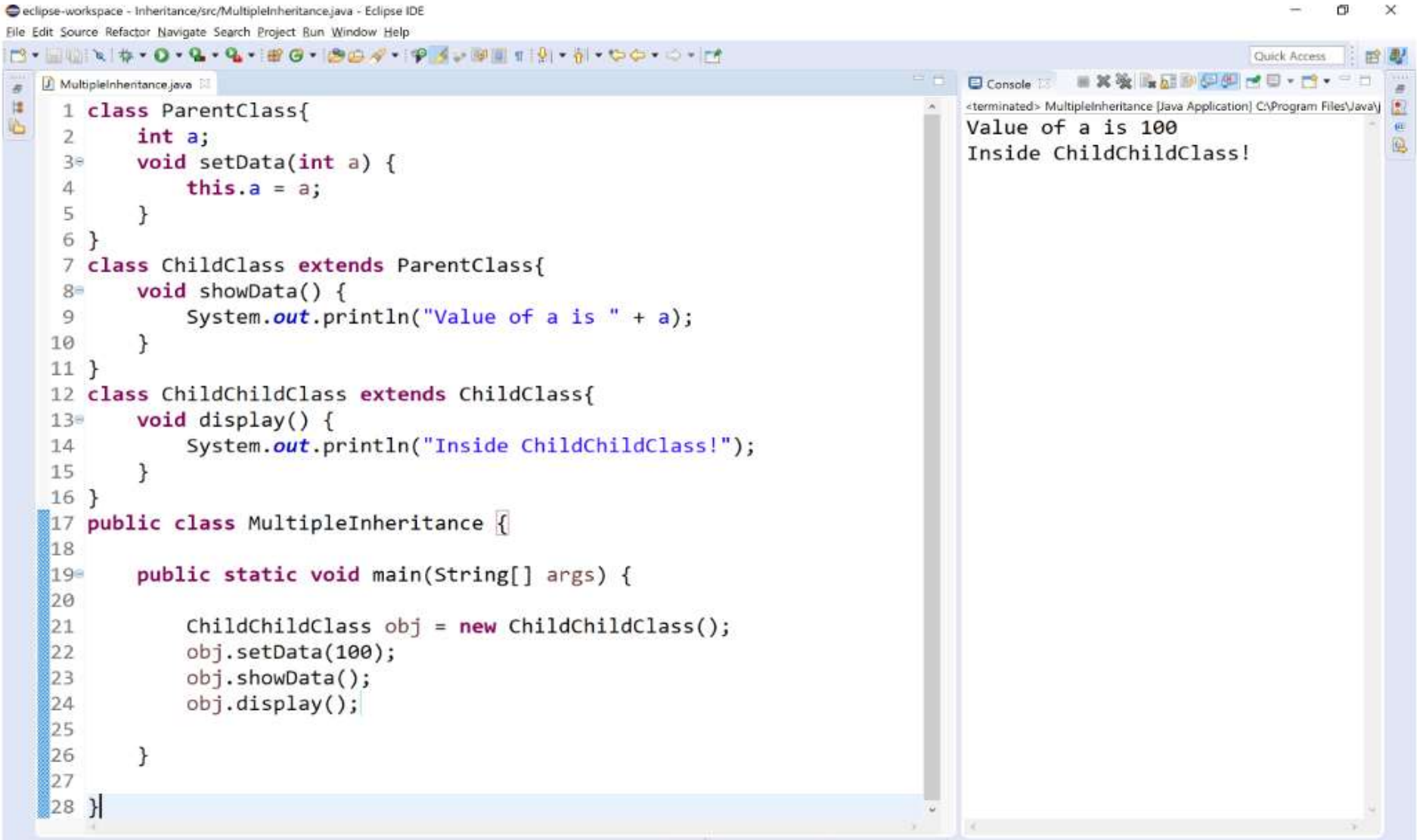
```
class <ChildClassName> extends <ParentClassName>{
    ...
    //Implementation of child class
    ...
}
```

In a java programming language, a class extends only one class. Extending multiple classes is not allowed in java.

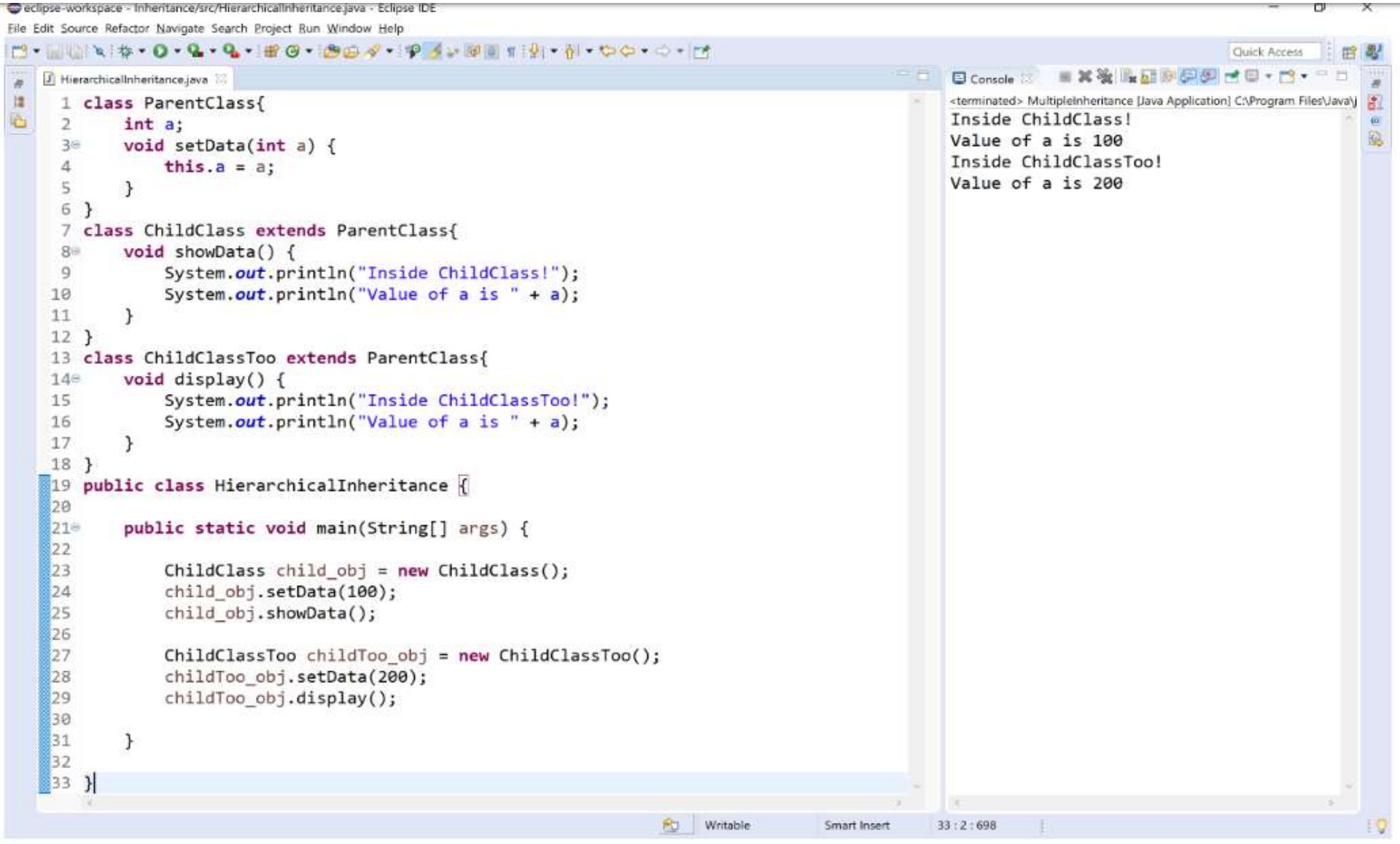
Single Inheritance



Multi-level Inheritance



Hierarchical Inheritance



Interface

Banking System

Every bank
should have
Credit money
Deposite money
And
ATM feature

Bank Interface

1. Deposite Money
2. Credit Money
3. ATM

SBI

HDFC BANK

```
interface Bank
{

  boolean credit (int amount);

  boolean deposite(int amount);

}
```

```
class HDFC implements Bank
{
    boolean create(int amount)
    {
      //Logic for credit money
    }
    boolean deposite(int amount)
    {
      //Logic for depositing money
    }
```

Interface

In java, an *interface* is similar to a class, but it contains abstract methods and static final variables only. The interface in Java is another mechanism to achieve abstraction.

We may think of an interface as a completely abstract class. None of the methods in the interface has an implementation, and all the variables in the interface are constants.

All the methods of an interface, implemented by the class that implements it.

The interface in java enables java to support multiple-inheritance. An interface may extend only one interface, but a class may implement any number of interfaces.

- An interface is a container of abstract methods and static final variables.
- An interface, implemented by a class. (**class implements interface**).
- An interface may extend another interface. (**Interface extends Interface**).
- An interface never implements another interface, or class.
- A class may implement any number of interfaces.
- We can not instantiate an interface.
- Specifying the keyword abstract for interface methods is optional, it automatically added.
- All the members of an interface are public by default.

Defining an interface in java

Defining an interface is similar to that of a class. We use the keyword interface to define an interface. All the members of an interface are public by default. The following is the syntax for defining an interface.

Syntax

```
interface InterfaceName{
    ...
    members declaration;
    ...
}
```

Let's look at an example code to define an interface.

Example

```
interface HumanInterfaceExample {

    void learn(String str);
    void work();

    int duration = 10;

}
```

In the above code defines an interface **HumanInterfaceExample** that contains two abstract methods learn(), work() and one constant duration.

Every interface in Java is auto-completed by the compiler. For example, in the above example code, no member is defined as public, but all are public automatically.

Implementing an Interface

In java, an **interface** is implemented by a class. The class that implements an interface must provide code for all the methods defined in the interface, otherwise, it must be defined as an abstract class.

The class uses a keyword ***implements*** to implement an interface. A class can implement any number of interfaces. When a class wants to implement more than one interface, we use the ***implements*** keyword is followed by a comma-separated list of the interfaces implemented by the class.

The following is the syntax for defineing a class that implements an interface.

Syntax

```
class className implements InterfaceName{
    ...
    boby-of-the-class
    ...
}
```

Let's look at an example code to define a class that implements an interface.

Example

```
interface Human {

    void learn(String str);
    void work();

    int duration = 10;

}

class Programmer implements Human{
    public void learn(String str) {
        System.out.println("Learn using " + str);
    }
    public void work() {
        System.out.println("Develop applications");
    }
}

public class HumanTest {

    public static void main(String[] args) {
        Programmer trainee = new Programmer();
        trainee.learn("coding");
        trainee.work();
    }
}
```

In the above code defines an interface **Human** that contains two abstract methods learn(), work() and one constant duration. The class Programmer implements the interface. As it implementing the Human interface it must provide the body of all the methods those defined in the Human interface.

Implementing multiple Interfaces

When a class wants to implement more than one interface, we use the *implements* keyword is followed by a comma-separated list of the interfaces implemented by the class.

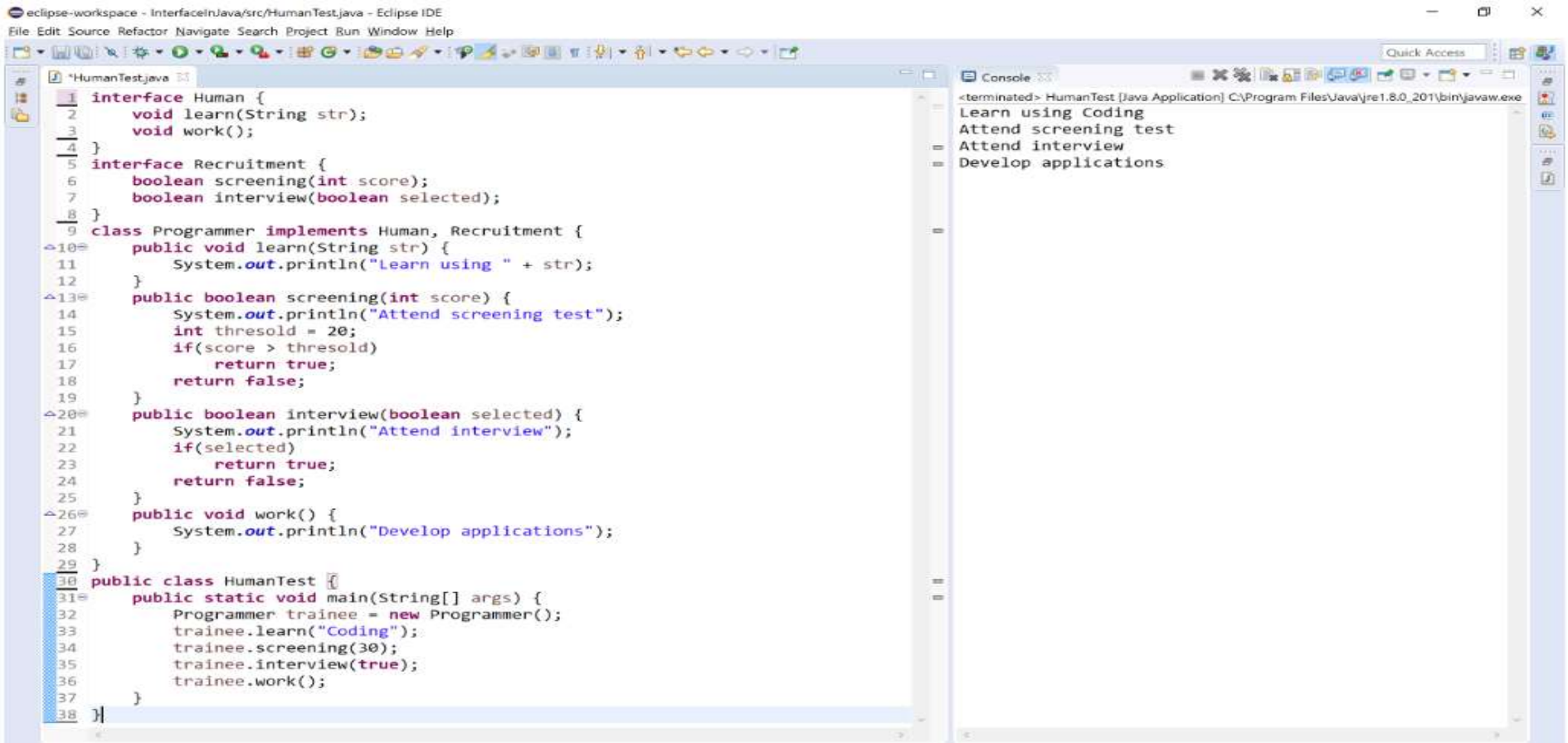
The following is the syntax for defineing a class that implements multiple interfaces.

Syntax

```
class className implements InterfaceName1, InterfaceName2, ...{
    ...
    boby-of-the-class
    ...
}
```

In the above code defines two interfaces **Human** and **Recruitment**, and a class Programmer implements both the interfaces.

When we run the above program, it produce the following output.



Nested Interfaces

In java, an interface may be defined inside another interface, and also inside a class. The interface that defined inside another interface or a class is konwn as nested interface. The nested interface is also refered as inner interface.

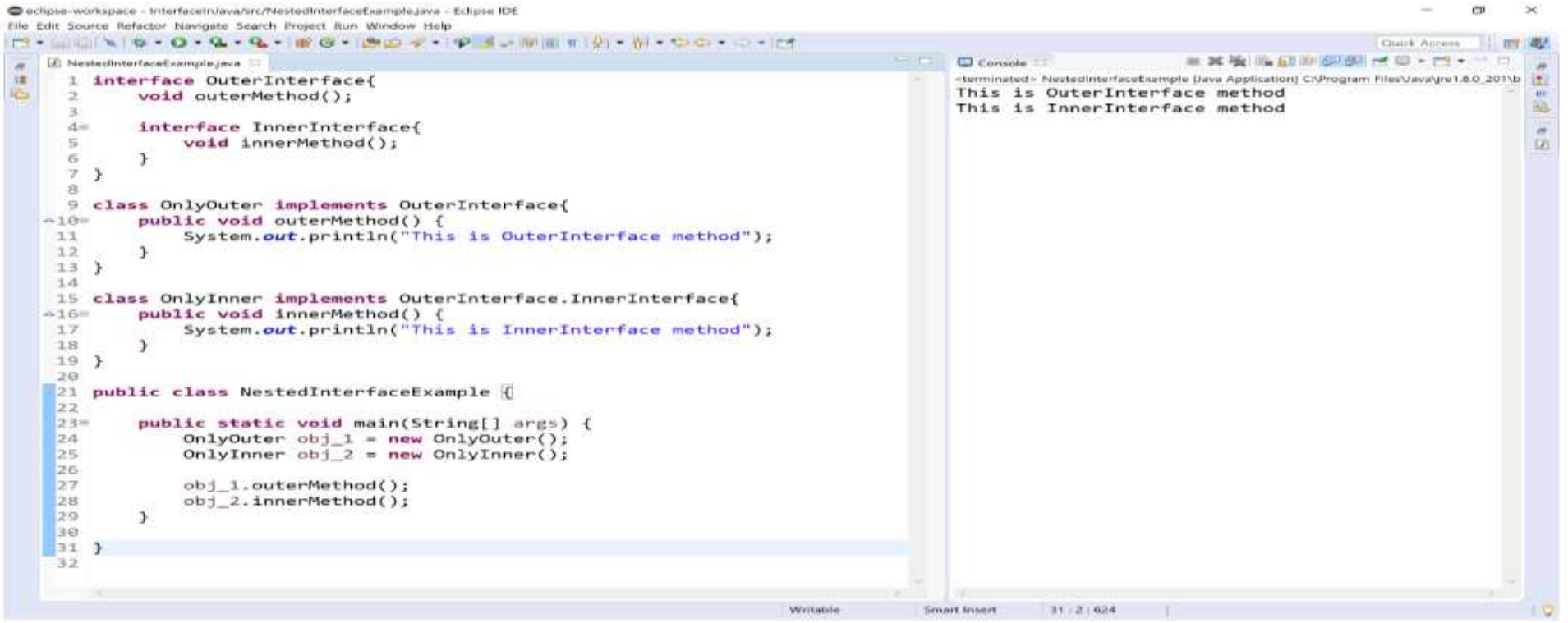
- The nested interface declared within an interface is public by default.
- The nested interface declared within a class can be with any access modifier.
- Every nested interface is static by default.

The nested interface cannot be accessed directly. We can only access the nested interface by using outer interface or outer class name followed by dot(.), followed by the nested interface name.

Nested interface inside another interface

The nested interface that defined inside another interface must be accessed as `OuterInterface.InnerInterface` .

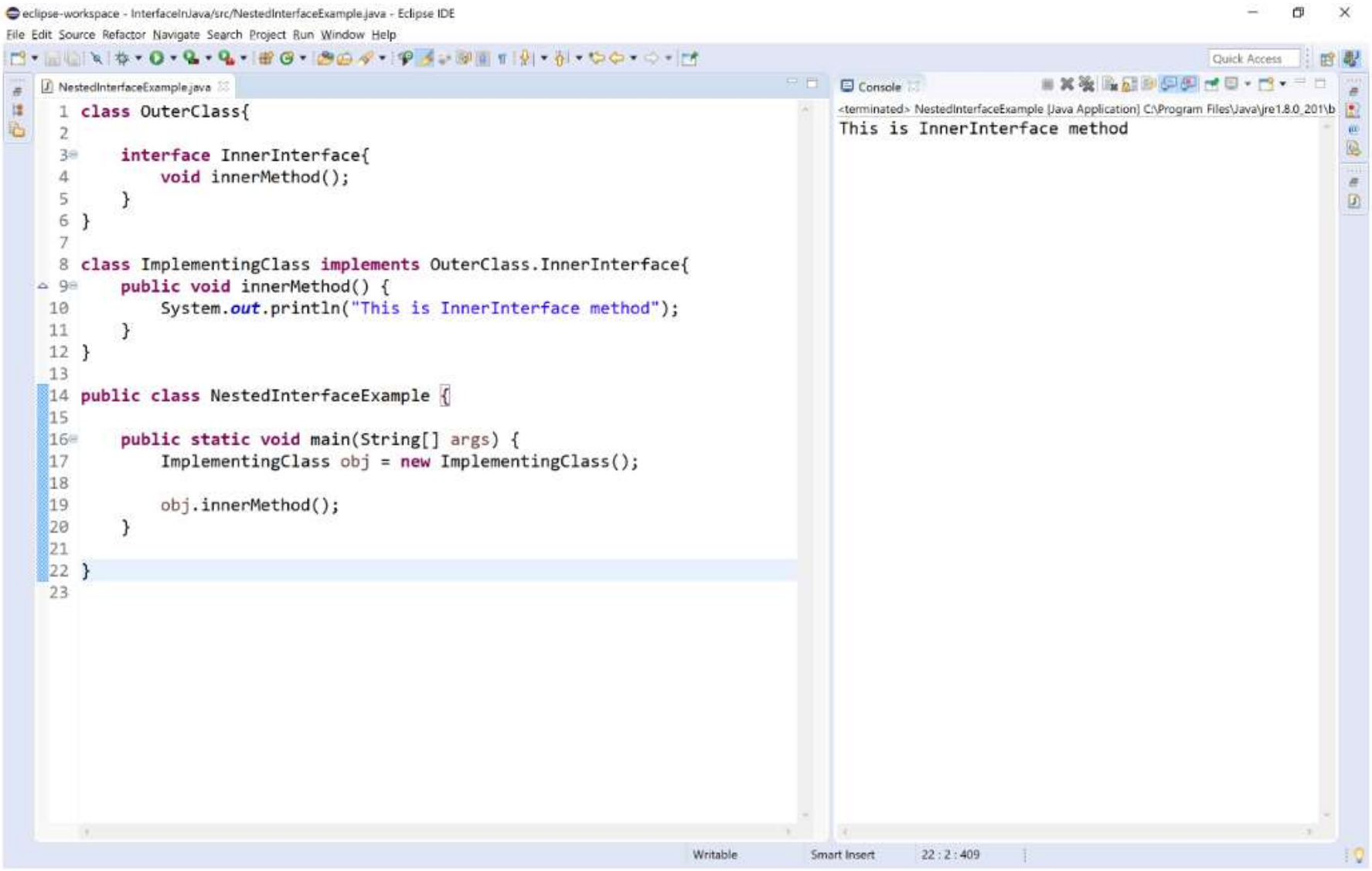
Let's look at an example code to illustrate nested interfaces inside another interface.



Nested interface inside a class

The nested interface that defined inside a class must be accessed as `ClassName.InnerInterface` .

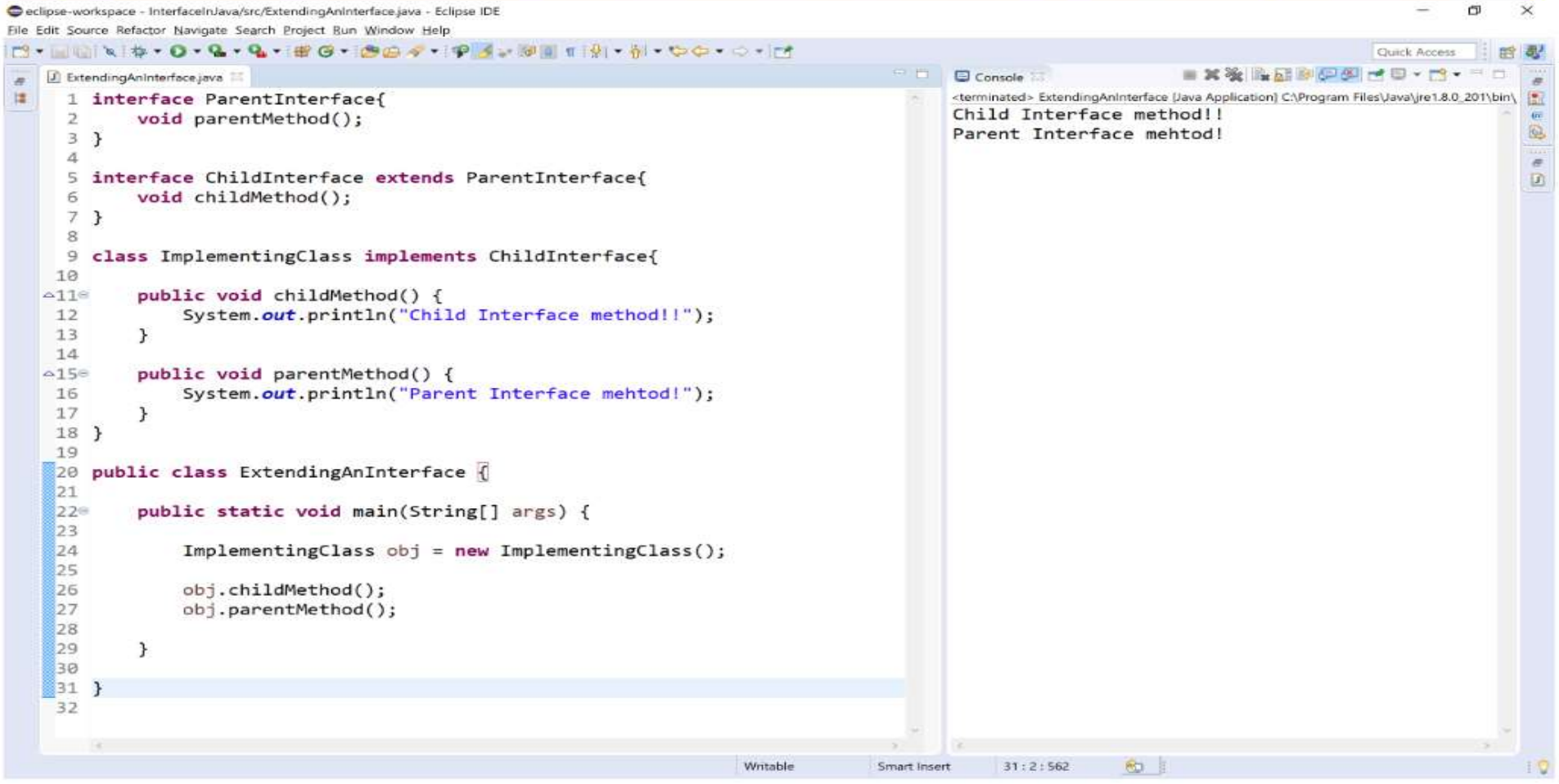
Let's look at an example code to illustrate nested interfaces inside a class.



Extending an Interface

In java, an interface can extend another interface. When an interface wants to extend another interface, it uses the keyword **extends**. The interface that extends another interface has its own members and all the members defined in its parent interface too. The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces, otherwise, it needs to be defined as abstract class.

- An interface can extend another interface.
- An interface can not extend multiple interfaces.
- An interface can implement neither an interface nor a class.
- The class that implements child interface needs to provide code for all the methods defined in both child and parent interfaces.



Packages

In java, a package is a container of classes, interfaces, and sub-packages. We may think of it as a folder in a file directory.

We use the packages to avoid naming conflicts and to organize project-related classes, interfaces, and sub-packages into a bundle.

In java, the packages have divided into two types.

- Built-in Packages
- User-defined Packages

Built-in Packages

The built-in packages are the packages from java API. The Java API is a library of pre-defined classes, interfaces, and sub-packages. The built-in packages were included in the JDK.

There are many built-in packages in java, few of them are as java, lang, io, util, awt, javax, swing, net, sql, etc.

We need to import the built-in packages to use them in our program. To import a package, we use the import statement.

User-defined Packages

The user-defined packages are the packages created by the user. User is free to create their own packages.

Definig a Package in java

We use the *package* keyword to create or define a package in java programming language.

Syntax

```
package packageName;
```

- The package statement must be the first statement in the program.
- The package name must be a single word.
- The package name must use Camel case notation.

Let's consider the following code to create a user-defined package myPackage.

Example

```
package myPackage;

public class DefiningPackage {

    public static void main(String[] args) {

        System.out.println("This class belongs to myPackage.");

    }

}
```

Now, save the above code in a file **DefiningPackage.java**, and compile it using the following command.

```
javac -d . DefiningPackage.java
```

The above command creates a directory with the package name *myPackage*, and the *DefiningPackage.class* is saved into it.

Run the program use the following command.

```
java myPackage.DefiningPackage
```

- When we use IDE like Eclipse, Netbeans, etc. the package structure is created automatically.

Access protection in java packages

In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public.

In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.

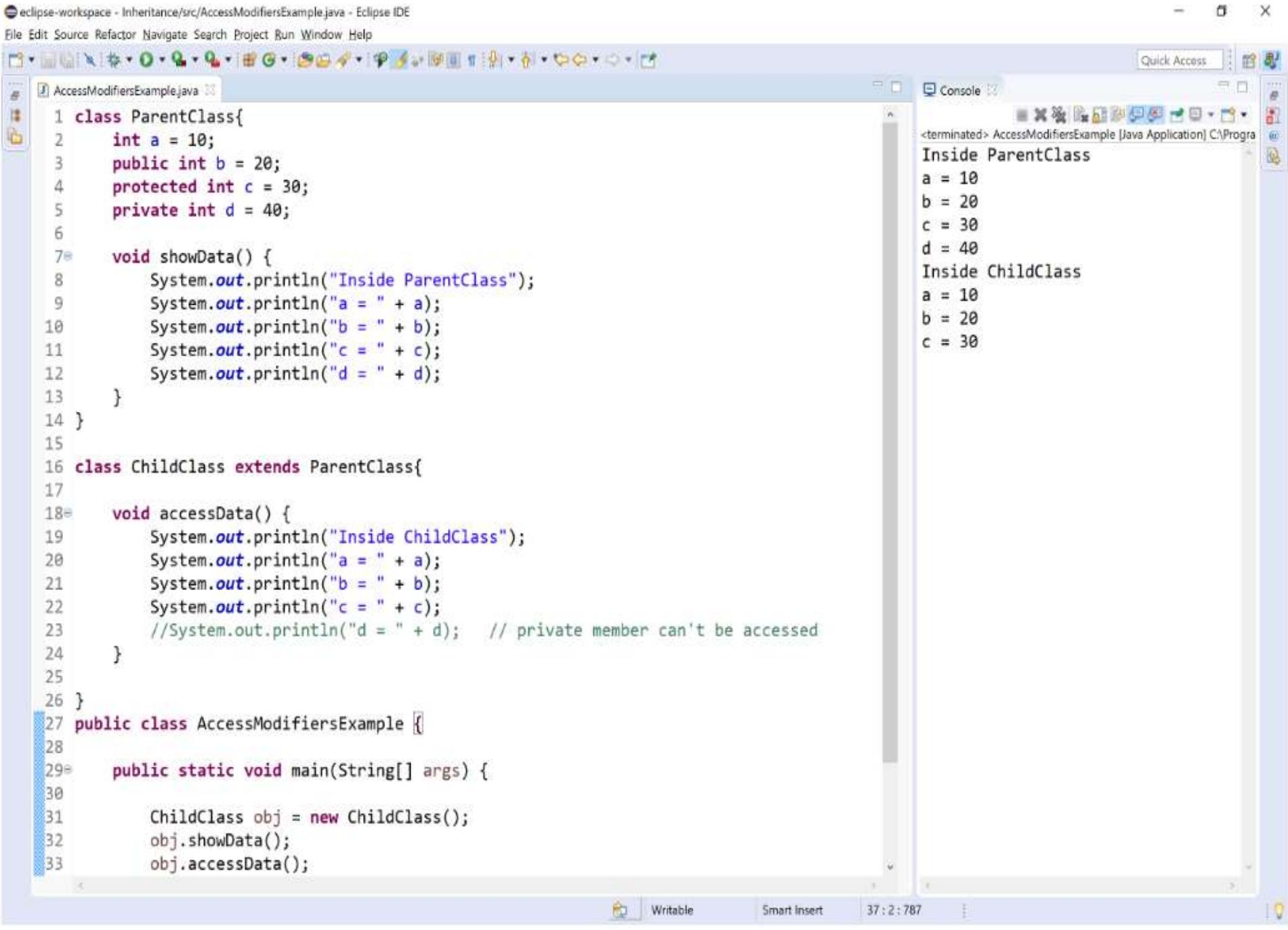
In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

Access control for members of class and interface in java

Accessibility / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

- The `public` members can be accessed everywhere.
- The `private` members can be accessed only inside the same class.
- The `protected` members are accessible to every child class (same package or other packages).
- The `default` members are accessible within the same package but not outside the package.

When we run this code, it produce the following output.




```java
class ParentClass{
    int a = 10;
    public int b = 20;
    protected int c = 30;
    private int d = 40;

    void showData() {
        System.out.println("Inside ParentClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}

class ChildClass extends ParentClass{

    void accessData() {
        System.out.println("Inside ChildClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        //System.out.println("d = " + d);   // private member can't be accessed
    }

}
public class AccessModifiersExample {

    public static void main(String[] args) {

        ChildClass obj = new ChildClass();
        obj.showData();
        obj.accessData();
```

Importing Packages

In java, the *import* keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly.

The import statement must be after the package statement, and before any other statement.

Using an import statement, we may import a specific class or all the classes from a package.

- Using one import statement, we may import only one package or a class.
- Using an import statement, we can not import a class directly, but it must be a part of a package.
- A program may contain any number of import statements.

Importing specific class

Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.

Syntax

```
import packageName.ClassName;
```

Let's look at an import statement to import a built-in package and Scanner class.

Example

```
package myPackage;

import java.util.Scanner;

public class ImportingExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);

        int i = read.nextInt();

        System.out.println("You have entered a number " + i);
    }
}
```

Importing all the classes

Using an importing statement, we can import all the classes of a package. To import all the classes of the package, we use * symbol. The following syntax is employed to import all the classes of a package.

Syntax

```
import packageName.*;
```

Let's look at an import statement to import a built-in package.

Example

```
package myPackage;

import java.util.*;

public class ImportingExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);

        int i = read.nextInt();

        System.out.println("You have entered a number " + i);

        Random rand = new Random();

        int num = rand.nextInt(100);

        System.out.println("Randomly generated number " + num);
    }
}
```

In the above code, the class **ImportingExample** belongs to **myPackage** package, and it also importing all the classes like Scanner, Random, Stack, Vector, ArrayList, HashSet, etc. from the **java.util** package.

- The import statement imports only classes of the package, but not sub-packages and its classes.
- We may also import sub-packages by using a symbol '.' (dot) to separate parent package and sub-package.

Consider the following import statement.

```
import java.util.*;
```

The above import statement `util` is a sub-package of `java` package. It imports all the classes of `util` package only, but not classes of `java` package.