

JAVA POLYMORPHISM

POLYMORPHISM

- Poly means “many” and morphs means “forms”. So polymorphism means many forms. (Having many forms)
- If a same method / single method performs different operations with **same / different signatures**, then it is called as polymorphism.
- Signature \leftarrow (Number of arguments, Type of arguments)
- One name many forms
- One of the core object oriented programming technique

Use

- It is mainly used to implement the concept of inheritance in oops.

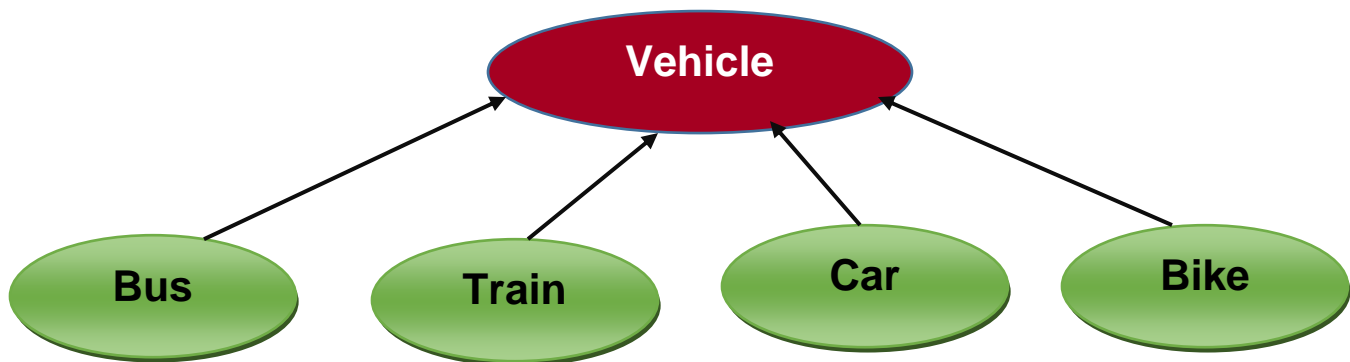


Figure: Polymorphism

Types of Polymorphism

- In java, polymorphism is classified as two types. They are
 1. Compile Time Polymorphism
 2. Run Time Polymorphism

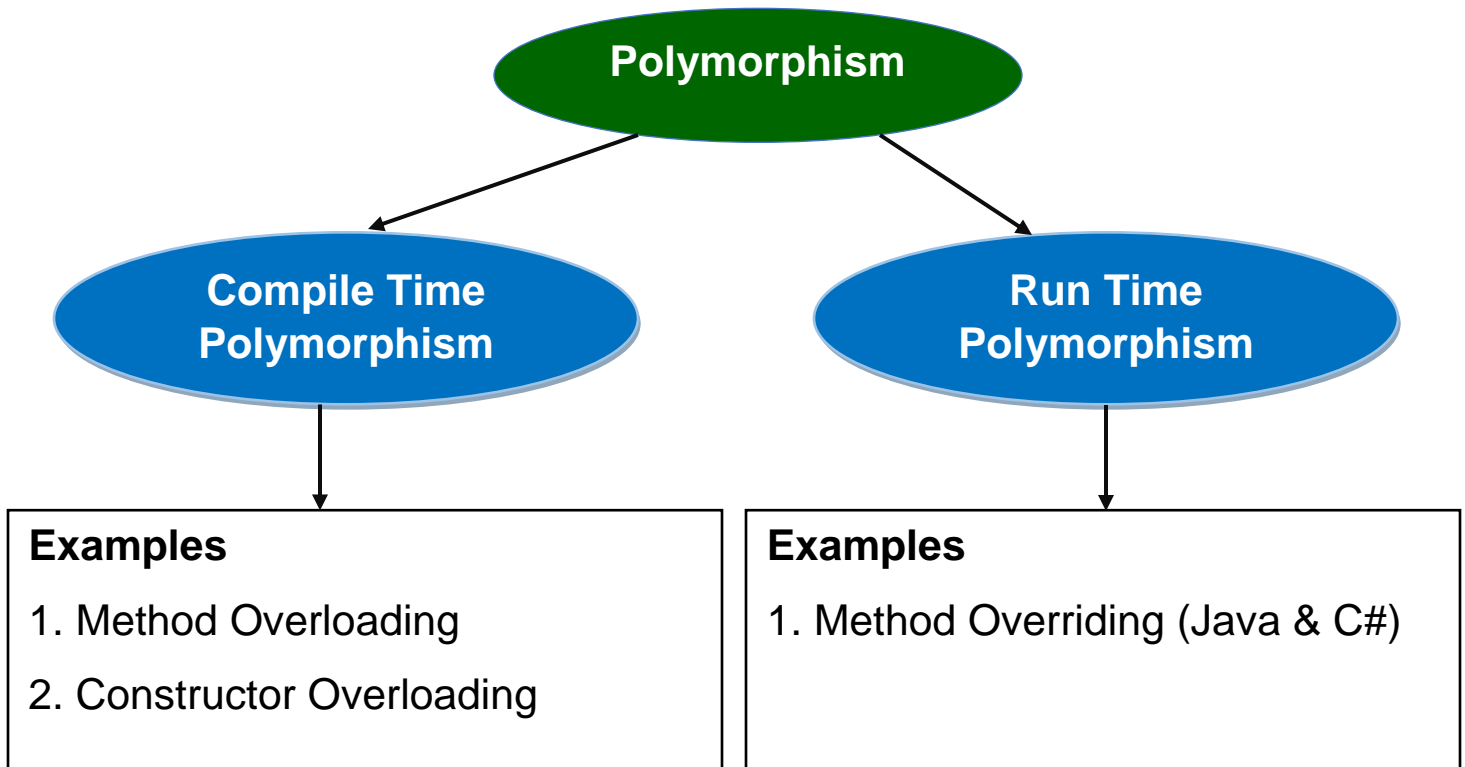


Figure: Types of Polymorphism

Note

- It is important to note that, **java does not support operator overloading.**

I. Compile Time Polymorphism

- Same function (single function) performs different operations using different signatures
- Signatures ← Number of arguments, Type of arguments
- This is called using its signatures
- Ex.
 1. Method overloading
 2. Constructor overloading

Method Overloading

- If same method / single method takes one or more number of arguments that is called as method overloading
- Same method (single method) performs different operations using different set of signatures
- This is called by using its signatures
 - Types of arguments → (int, float, void, char)
 - Number of arguments → (0,1,2,3, ... n)
- **Types**
 1. Argument based function overloading
 2. Type based function overloading

1. Argument based function overloading

- If same function performs different operations using different set of arguments (0, 1, 2, etc, ...) then is called as argument based function overloading
- It gives the importance to arguments (not types)

- This is called by using its signatures. Here the signature depends on the number of arguments (not types)

Calling

- Calling is purely **based on arguments order**. It does not provide the priority to data types

2. Type based function overloading

- If same function performs different operations using **different set of types (int, float, double, etc, ...)** then it is called as type based function overloading
- It gives the **importance to data types** (not arguments)
- This is called by **using its signatures**. Here the signature depends on the data types of methods

Calling

- Calling is purely **based on types**. It does not provide the priority to number of arguments

DIFFERENCE BETWEEN OVERLOADING AND OVERRIDING

S.N	Overloading	Overriding
1.	Same method with different operations using different signatures	Same method with different operations using same signatures
2.	Compile time polymorphism (static binding or early binding)	Runtime polymorphism (dynamic binding or late binding)
3.	It does not need inheritance	It needs inheritance to achieve overriding concept
4.	Calling is based on its signatures	This is implemented using super keyword
5.	Method can use different access modifier for each method	Method should use same modifier for each method

I. TYPE BASED METHOD OVERLOADING

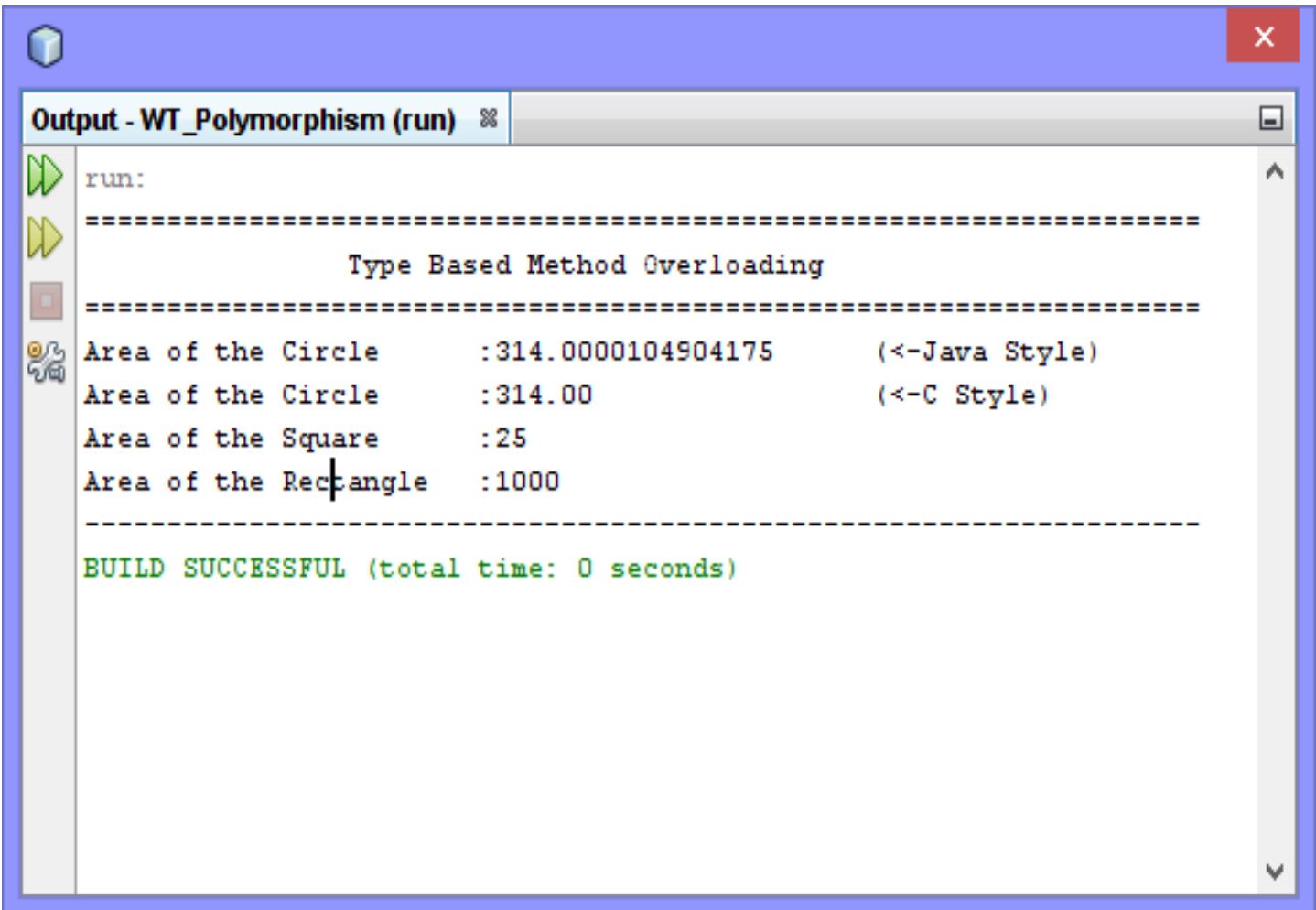
(Type_Overloading.java)

1. SOURCE CODE

```
class Moverloading
{
    final float PI=3.14f;           // constant variable
    // area of the square
    public int area(int x)
    {
        return(x*x);
    }
    // area of the circle
    public double area(double r)
    {
        return(PI*r*r);
    }
    // area of the rectangle
    public long area(int l, int b)
    {
        return(l*b);
    }
}
// main class
public class Type_Overloading
{
    // main method
    public static void main(String[] args)
    {
        System.out.println("=====");
        System.out.println("\t\tType Based Method Overloading");
        System.out.println("=====");
    }
}
// object creation using new modifier
```

```
Moverloading obj=new Moverloading();
// calling method overloading using its type based signatures
double circle=obj.area(10.0);
int square=obj.area(5);
long rect=obj.area(100, 10);
// print the result
System.out.println("Area of the Circle \t:"+circle+"\t(<-Java Style)");
System.out.printf("Area of the Circle \t:%6.2f\t\t(<-C Style)\n",circle);
System.out.println("Area of the Square \t:"+square);
System.out.println("Area of the Rectangle \t:"+square);
System.out.println("-----");
}
```

2. OUTPUT



```
run:
=====
                Type Based Method Overloading
=====
Area of the Circle      :314.0000104904175      (<-Java Style)
Area of the Circle      :314.00                (<-C Style)
Area of the Square      :25
Area of the Rectangle    :1000
-----
BUILD SUCCESSFUL (total time: 0 seconds)
```

II. ARGUMENT BASED METHOD OVERLOADING

(Argument_Overloading.java)

1. SOURCE CODE

```
class Aoverloading
{
// zero argument overloading
    void info()
    {
        System.out.println("Zero Argument");
    }
// one argument overloading
    void info(int k)
    {
        System.out.println("One Argument");
        System.out.println("\tk= "+k);
    }
// two arguments overloading
    void info(int id, String name)
    {
        System.out.println("Two Arguments");
        System.out.println("\tName= "+name);
        System.out.println("\tid= "+id);
    }
// one argument overloading
    void info(double d)
    {
        System.out.println("One Argument");
        System.out.println("\td= "+d);
    }
}
```


// main class

public class Argument_Overloading

{

// main method

public static void main(String[] args)

{

System.out.println("=====");

System.out.println("\tArgument Based Method Overloading");

System.out.println("=====");

// object creation using new modifier

Aoverloading obj=new Aoverloading();

// calling method overloading using its argument based signatures

obj.info(10.0);

System.out.println("-----");

obj.info(33,"Sachin");

System.out.println("-----");

obj.info();

System.out.println("-----");

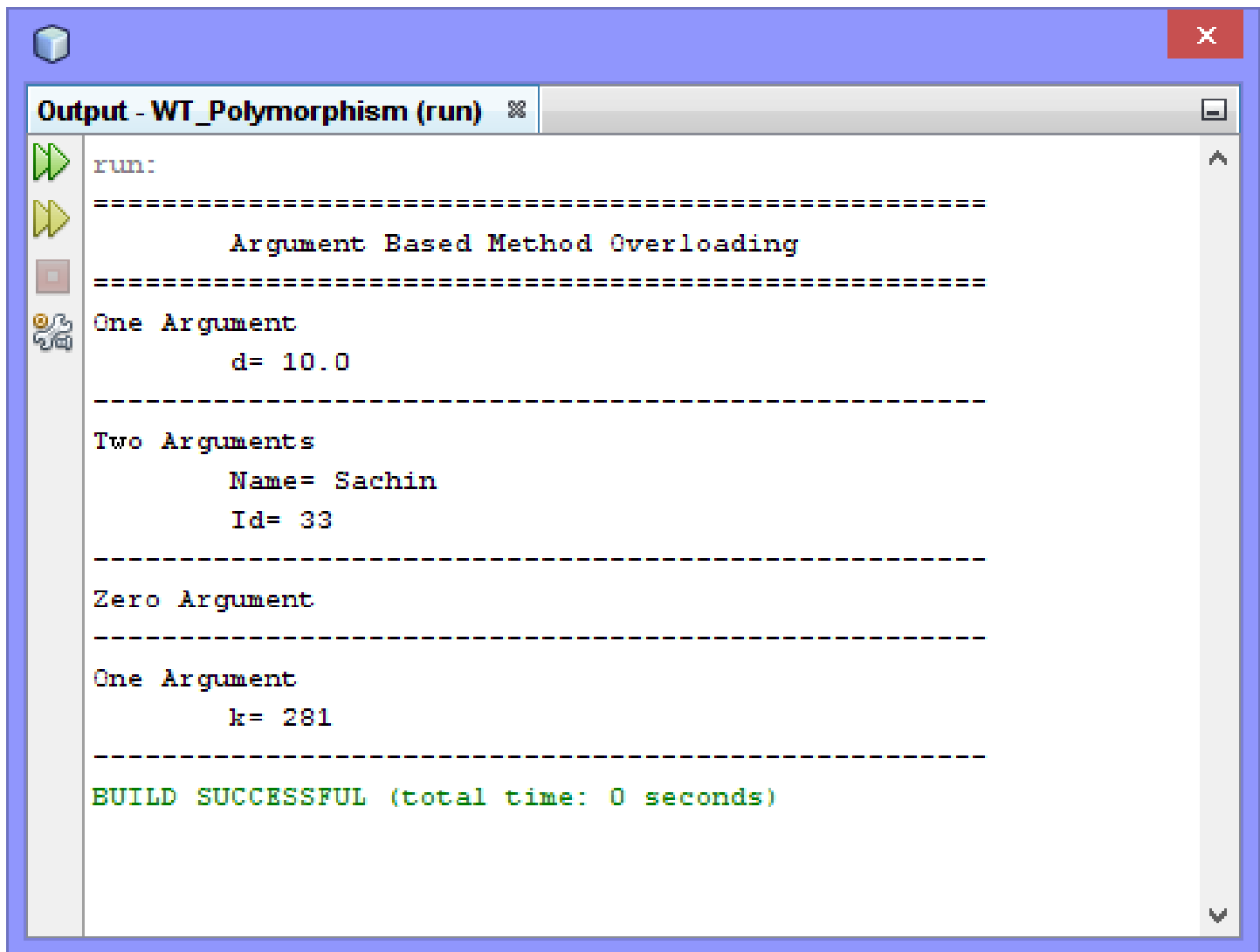
obj.info(281);

System.out.println("-----");

}

}

2. OUTPUT



```
run:
=====
      Argument Based Method Overloading
=====
One Argument
      d= 10.0
-----
Two Arguments
      Name= Sachin
      Id= 33
-----
Zero Argument
-----
One Argument
      k= 281
-----
BUILD SUCCESSFUL (total time: 0 seconds)
```

CONSTRUCTOR OVERLOADING

- If a same constructor performs different operations **with different set of arguments**, then it is called as constructor overloading.
- Constructor supports **ONLY arguments based overloading**. It does not support type based overloading. Because **it does not return any type of data**.

Calling

- Constructor overloading is called by its signature (order of arguments)
- **The arguments in the constructor call must match the constructor definition inside the class**. Else it provides the error message.

III. ARGUMENT BASED CONSTRUCTOR OVERLOADING

(Argu_Overloading.java)

1. SOURCE CODE

```
public class Argu_Overloading
{
// zero argument constructor overloading
    Argu_Overloading()
    {
        System.out.println("Zero Argument Constructor");
    }
// one argument constructor overloading
    Argu_Overloading(int a)
    {
```

```
        System.out.println("One Argument Constructor");
        System.out.println("\tint value="+a);
    }
// one argument constructor overloading
    Argu_Overloading(float a)
    {
        System.out.println("One Argument Constructor");
        System.out.println("\tfloat value="+a);
    }
// two arguments constructor overloading
    Argu_Overloading(int a, String str)
    {
        System.out.println("Two Arguments Constructor");
        System.out.println("\tid="+a);
        System.out.println("\tname="+str);
    }

// main method
public static void main(String[] args)
{
    System.out.println("=====");
    System.out.println("\tArgument Based Constructor Overloading");
    System.out.println("=====");
// calling 1-argument constructor
    Argu_Overloading a1=new Argu_Overloading(54);
    System.out.println("-----");
// calling 0-argument constructor
    Argu_Overloading a2=new Argu_Overloading();
    System.out.println("-----");
// calling 2-arguments constructor
    Argu_Overloading a3=new Argu_Overloading(23,"Sachin");
    System.out.println("-----");
```

// calling 1-argument constructor

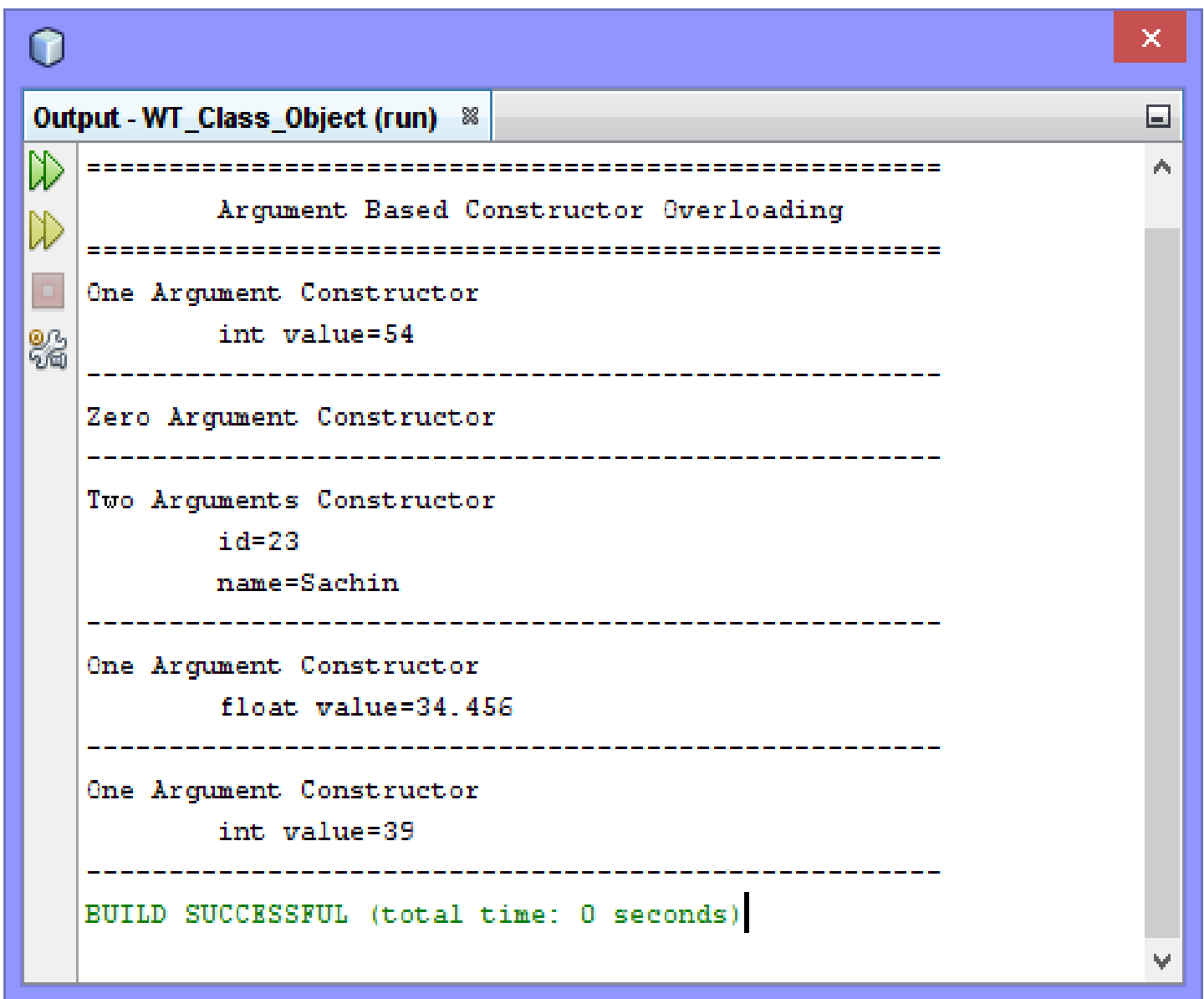
```
Argu_Overloading a4=new Argu_Overloading(34.456f);  
System.out.println("-----");
```

// calling 1-argument constructor

```
Argu_Overloading a5=new Argu_Overloading(39);  
System.out.println("-----");
```

```
    }  
}
```

2. OUTPUT



```
=====
Argument Based Constructor Overloading
=====
One Argument Constructor
    int value=54
-----
Zero Argument Constructor
-----
Two Arguments Constructor
    id=23
    name=Sachin
-----
One Argument Constructor
    float value=34.456
-----
One Argument Constructor
    int value=39
-----
BUILD SUCCESSFUL (total time: 0 seconds)
```

II. Run Time Polymorphism

- Same method (single method) performs different operations using **same set of signatures**
- Signatures ← Number of arguments, Type of arguments
- This is called using its signatures
- Ex.

1. Method Overriding

Existing Problem

- If variables or methods or combination of both are defined in both super class and sub class, then compiler may confuse to call the methods. (This situation is called **as ambiguity error**)
- So only the **methods of derived class will be executed** forever and super class methods will be hidden
- In order to call the members of base class (super class), we have to use super keyword

Solutions for Run time polymorphism

- In the run time polymorphism, **the sub class (derived class) always hides the base class functions / variables, because of same signatures**
- To overcome this problem, java provides one solution. Namely
1. Method Overriding

METHOD OVERRIDING

- If a same method (single method) performs **different operations with same set of signatures**, then it is called as method overriding.
- Unlike C++, java doesn't use the **virtual keyword in super class**. Because methods in java are **virtual by default**. So no need to mention the virtual keyword in super class.

Problem

- Whenever variables and methods are defined with **same name** in both super class and sub class, that time JVM will call only sub class members and hiding the members of super class
- This is the main drawback.

Solutions

1. Use the **super** keyword to **access the super class members**.
2. Use **up casting** approach to call both methods of super and sub class using **super class object**

1. Method Overriding using Super Keyword [Solution 1]

- The super keyword is **used to access the variables, methods and constructors of the super class** that has the same name in the sub class.
- Unlike this(), the super() expression can be placed at anywhere (first statement or last statement) in the method of sub class
- **It is important to note that, here both variables and methods of super class can be overridden.**

Method Overriding using Up Casting [Solution 2]

- It is possible to call the sub class (derived class) methods using super class object
- This is similar to c++ virtual function. Java does **not support pointers**. It uses reference instead of pointer
- Process of assigning the object of sub class (derived class) to super class object variable. Once object is assigned, then super class object can call the methods of sub class (derived class)
- Here only methods of super class can be overridden. **Variables of super class can't be overridden in sub class (derived class)**

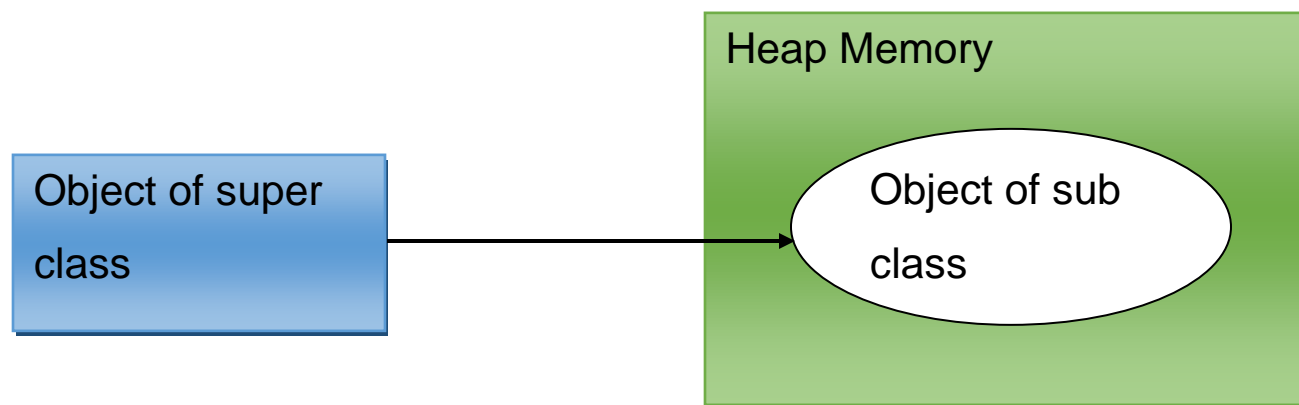


Figure: Up casting

Overriding Rules

- Applies ONLY to inherited methods related to polymorphism
- Overriding methods must have the same argument list and same return types
- Abstract method must be overridden
- **Final methods cannot be overridden**
- **Static methods cannot be overridden**
- **Constructors cannot be overridden.**

V. TRADITIONAL OVERRIDING IN INHERITANCE

(Traditional.java)

1. SOURCE CODE

// super class

```
class A
{
    int k=99;
    void test()
    {
        System.out.println("Class A is calling...");
        System.out.println("k= "+k);
    }
}
```

Super Class

// inherit the properties super class A

```
class B extends A
{
    int k=248;
    void test()
    {
        System.out.println("Class B is calling...");
        System.out.println("k= "+k);
    }
}
```

Sub Class

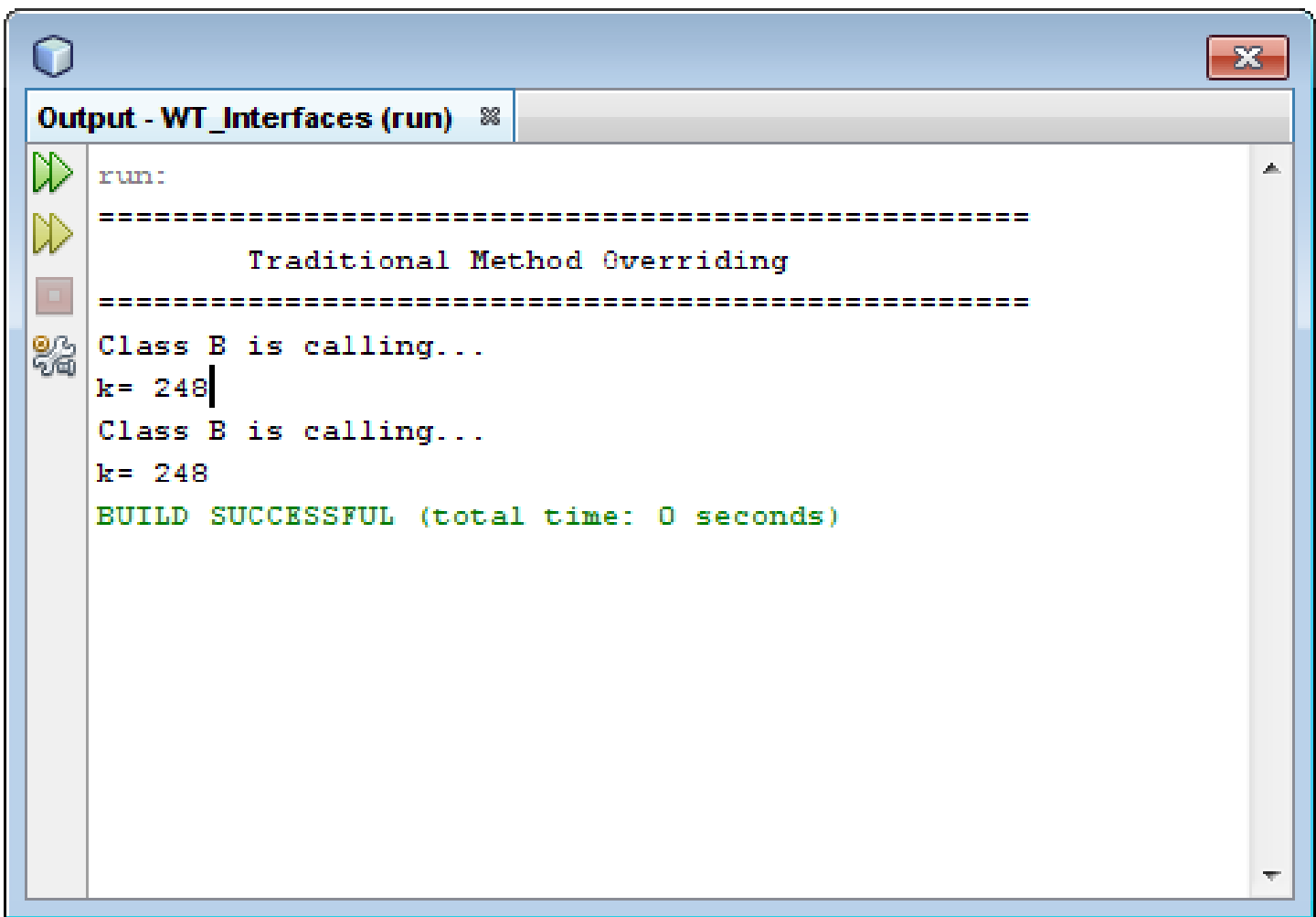
// main class

```
public class Traditional
{
    public static void main(String[] args)
    {
        System.out.println("=====");
        System.out.println("\tTraditional Method Overriding ");
        System.out.println("=====");
    }
}
```

```
// create an object for sub class
    B obj=new B();
// call the methods of super & sub classes using sub class object
    obj.test();
    obj.test();

}
}
```

2. OUTPUT



VI. SOLVING METHOD OVERRIDING

(THROUGH INHERITANCE USING **SUPER KEYWORD**)

(Traditional.java)

Used Inheritance : **Single Inheritance**

1. SOURCE CODE

// super class

```
class A
{
    int k=99;
    void test()
    {
        System.out.println("Class A is calling...");
        System.out.println("k= "+k);
    }
}
```

// Inherit the properties of super class A

class B **extends** A

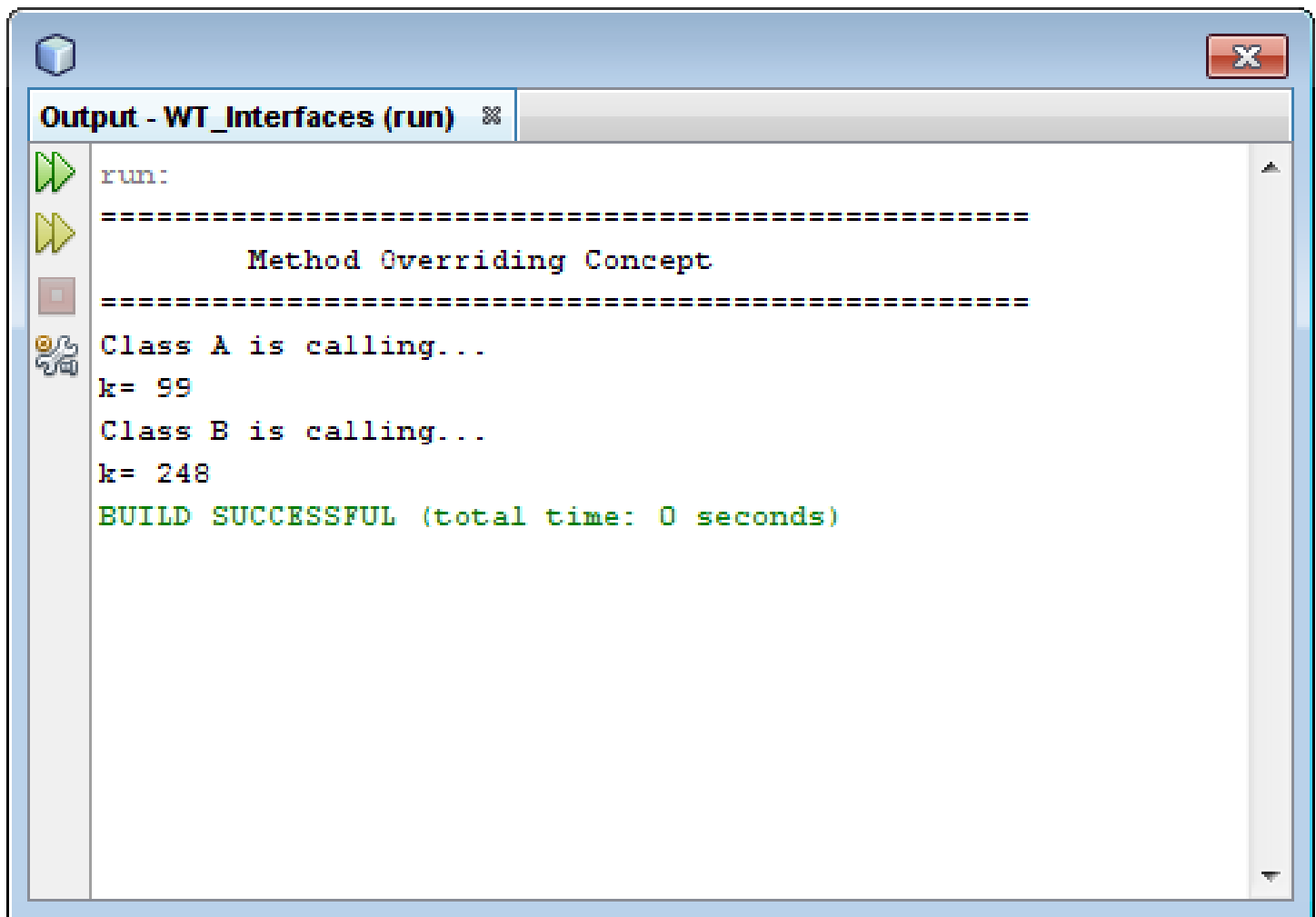
```
{
    int k=248;
    void test()
    {
        super.test();
        System.out.println("Class B is calling...");
        System.out.println("k= "+k);
    }
}
```

The super() can be placed anywhere in the derived class method

// call the super class method

```
    }  
}  
  
// main class  
public class Traditional  
{  
    // main method  
    public static void main(String[] args)  
    {  
        System.out.println("=====");  
        System.out.println("\tMethod Overriding Concept");  
        System.out.println("=====");  
        // create an object for sub class  
        B obj=new B();  
        // call the methods of super & sub classes using sub class object  
        obj.test();  
    }  
}
```

2. OUTPUT



```
run:
=====
      Method Overriding Concept
=====
Class A is calling...
k= 99
Class B is calling...
k= 248
BUILD SUCCESSFUL (total time: 0 seconds)
```

VII. SOLVING METHOD OVERRIDING (THROUGH INHERITANCE USING **UP CASTING**)

(Traditional.java)

Used Inheritance : **Single Inheritance**

1. SOURCE CODE

// super class

class S

```
{  
    int i=25;  
    void disp()  
    {  
        System.out.println("Super class is calling ...");  
    }  
}
```

// sub class

public class N **extends** S

```
{  
    int i=50;  
    void disp()  
    {  
        System.out.println("Sub class is calling ...");  
    }  
    public static void main(String[] args)  
    {
```

Up Casting

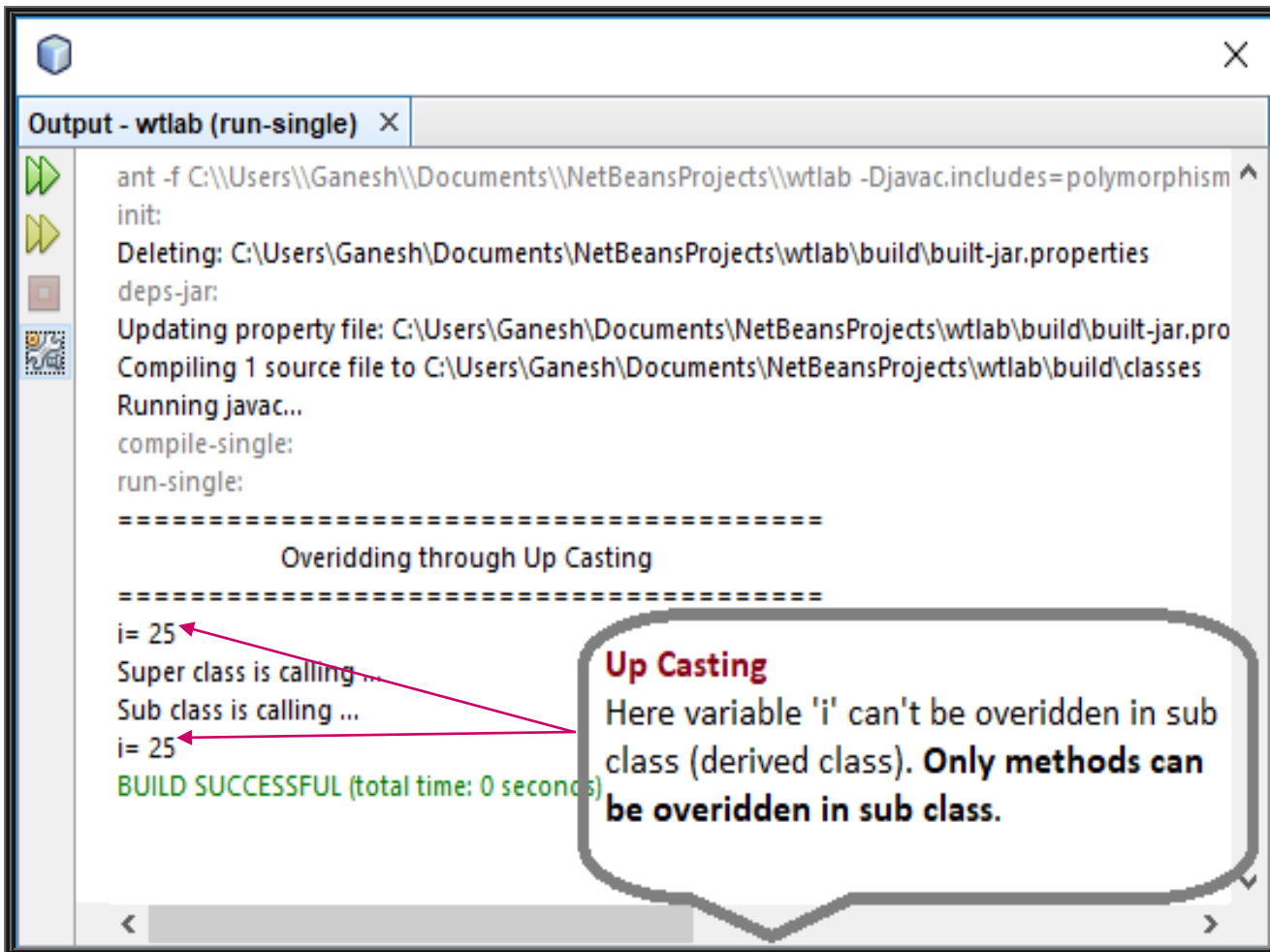
Super class variable **cannot be overridden** in sub class.

Up Casting

Super class method can be overridden in sub class.

```
        System.out.println("=====");
        System.out.println("\tOverriding through Up Casting");
        System.out.println("=====");
// object creation for super class
        S obj=new S();
        System.out.println("i= "+obj.i);
// calling super class method using super class object
        obj.disp();
// object creation for sub class
        N sub=new N();
// up casting: assigning sub class object to super class
        obj=sub;
// calling sub class method using super class object
        obj.disp();
        System.out.println("i= "+obj.i);
    }
}
```

2. OUTPUT



```
Output - wtlab (run-single) X
ant -f C:\\Users\\Ganesh\\Documents\\NetBeansProjects\\wtlab -Djavac.includes=polymorphism
init:
Deleting: C:\\Users\\Ganesh\\Documents\\NetBeansProjects\\wtlab\\build\\built-jar.properties
deps-jar:
Updating property file: C:\\Users\\Ganesh\\Documents\\NetBeansProjects\\wtlab\\build\\built-jar.pro
Compiling 1 source file to C:\\Users\\Ganesh\\Documents\\NetBeansProjects\\wtlab\\build\\classes
Running javac...
compile-single:
run-single:
=====
                Overriding through Up Casting
=====
i= 25
Super class is calling ...
Sub class is calling ...
i= 25
BUILD SUCCESSFUL (total time: 0 seconds)
```

Up Casting
Here variable 'i' can't be overridden in sub class (derived class). **Only methods can be overridden in sub class.**

VII. OVERLOADING THROUGH INTERFACE

(loverloading.java)

Used Inheritance : **Hierarchical Inheritance**

1. SOURCE CODE

// super interface

interface Shape

```
{  
    void area(int v);  
}
```

// empty method

Super Interface

// sub class 1

class Square implements Shape

```
{  
    public void area(int a)  
    {  
        int rs=a*a;  
        System.out.println("Area of the Square : "+rs);  
    }  
}
```

Sub Class 1

// sub class 2

class Circle implements Shape

```
{  
    final float PI=3.14f;  
    public void area(int r)  
    {  
        float rs=PI*r*r;  
        System.out.println("Area of the Circle : "+rs);  
    }  
}
```

Sub Class 2

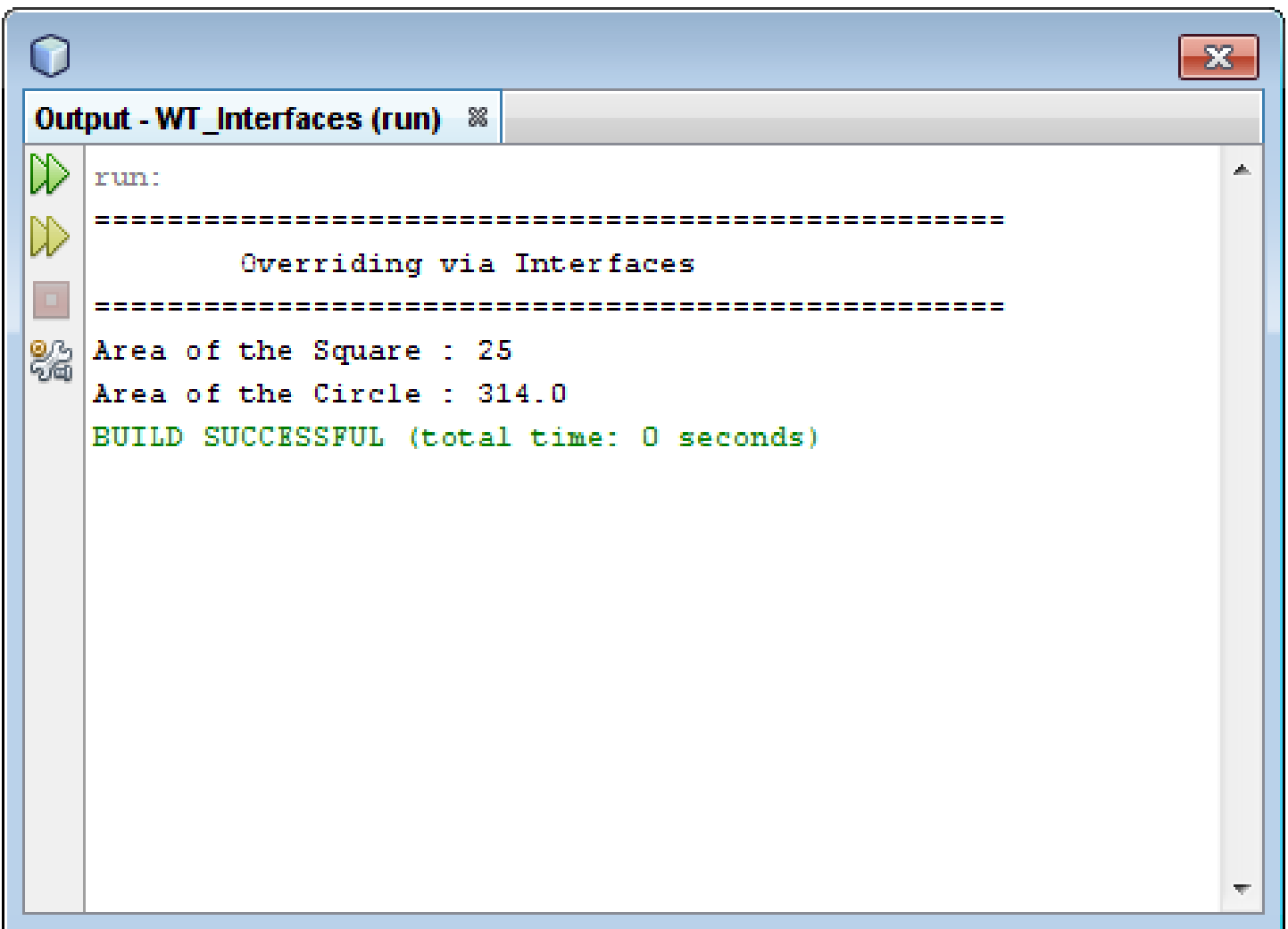
// main class

public class loverloading

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

```
{  
    System.out.println("=====");  
    System.out.println("\tOverriding via Interfaces");  
    System.out.println("=====");  
    // create an object for square sub class  
    Square sq=new Square();  
    sq.area(5);  
    // create an object for circle class  
    Circle cr=new Circle();  
    cr.area(10);  
}  
}
```

2. OUTPUT



```
run:  
=====  
    Overriding via Interfaces  
=====  
Area of the Square : 25  
Area of the Circle : 314.0  
BUILD SUCCESSFUL (total time: 0 seconds)
```