# Polymorphism

Polymorphism means many forms; it occurs when we have many classes related to each other by inheritance.

We have 2 types of polymorphism.

* Compile time polymorphism
* Runtime polymorphism

## Compile-time polymorphism

Binding a method call with a method definition is done at compile time is called compile-time polymorphism.

Compile-time polymorphism is implemented through method overloading.

Compile-time polymorphism is also known as static polymorphism or early binding.

Compile-time polymorphism is achieved by method overloading or operator overloading.

### Method Overloading:

We can have one or more methods with the same name that is distinguished by argument number, type, or order.

void gfg() { ... }

void gfg(int num1 ) { ... }

void gfg(float num1) { ... }

void gfg(int num1 , float num2 ) { ... }

Refer program - MethodOverloadingExample1\_Class

|  |
| --- |
| **package** polymorphism;  **public** **class** MethodOverloadingExample1\_Class  {  **public** **static** **void** main(String[] args)  {  Demo d = **new** Demo();  **int** x = d.add(10, 20);  **float** y = d.add(10.0f, 20);  **float** z = d.add(10, 20.12f);    System.***out***.println(x+ " - "+y+" - "+z);  }  }  **class** Demo  {  **public** **int** add(**int** a,**int** b)  {  **return** a+b;  }  **public** **float** add(**float** a,**int** b)  {  **return** a+b;  }  **public** **float** add(**int** a,**float** b)  {  **return** a+b;  }  } |

### Operator Overloading

An operator is said to be overloaded if it can perform more than one function. In java, the + operator is overloaded. It works as an arithmetic addition operator, and it also can be used to join the strings.

Java does not allow user-defined operator overloading.

### Advantages of compile-time polymorphism

1. It improves code clarity and allows for the use of a single name for similar procedures.
2. It has a faster execution time since it is discovered early in the compilation process.

## Run-time polymorphism

Binding a method call with method definition at run time is called run-time polymorphism.

It is late binding.

It is also called dynamic method dispatch.

It can be achieved using method overriding.

If two or more methods with the same parameter list, then it is said to be method overriding. Methods cannot be overridden in the same class because of the ambiguity of to call.

Methods can be overridden in inheritance only.

Refer program: MethodOverriddingExample1\_Class

|  |
| --- |
| **package** polymorphism;  **public** **class** MethodOverriddingExample1\_Class  {  **public** **static** **void** main(String[] args)  {  B obj = **new** B();  **int** x = obj.add(10, 20);  System.***out***.println(x);    obj.show();  }  }  **class** A  {  **public** **int** add(**int** a,**int** b)  {  **return** a+b;  }    **public** **void** show()  {  System.***out***.println("Class A");  }  }  **class** B **extends** A  {  **public** **int** add(**int** a,**int** b)  {  **return** a+b;  }  **public** **void** show()  {  System.***out***.println("Class B");  }  } |

In the above example, A class add and show methods are overridden by B class add and show methods. Here B class methods are referred to as overriding methods and A-class methods are referred to as Overridden methods.

Rules of Runtime Polymorphism

* Methods of child and parent class must have the same name.
* Methods of child and parent class must have the same parameter.
* IS-A relationship is mandatory (inheritance).

Limitations of Runtime Polymorphism

* One cannot override the private methods of a parent class.
* One cannot override Final methods.
* One cannot override static methods.

Example 2 – Refer Program: RuntimePolymorphimsExample2

|  |
| --- |
| **package** polymorphism;  **public** **class** RuntimePolymorphimsExample2  {  **public** **static** **void** main(String[] args)  {  parent p = **new** parent();  p.show();    child c = **new** child();  c.show();    parent p1 = **new** child();  p1.show(); // ==========> this is runtime polymorphism  }  }  **class** parent  {  **public** **void** show()  {  System.***out***.println("\nParent class");  }  }  **class** child **extends** parent  {  @Override  **public** **void** show()  {  System.***out***.println("\nChild class");  }  } |

Example 3: Refer Program: RuntimepolymorphishmExample3

|  |
| --- |
| //Example with multilevel inheritenc  **package** polymorphism;  **public** **class** RuntimepolymorphishmExample3  {  **public** **static** **void** main(String[] args)  {  Human superObject=**new** Human();  Human subObject=**new** Man(); // // upcasting : first level of heritance  Human babyObject=**new** Baby(); // // upcasting : second level of heritance  superObject.sip();  subObject.sip(); //run time polymorphism happening in first level of heritance  babyObject.sip(); //run time polymorphism happening in second level of heritance  }  }  **class** Human  {  **void** sip()  {  System.***out***.println("Human is sipping");  }  }  **class** Man **extends** Human  {  **void** sip()  {  System.***out***.println("Man is sipping soup");  }  }  **class** Baby **extends** Man  {  **void** sip()  {  System.***out***.println("Baby is sipping milk");  }  } |

Example 4: Refer Program: staticprivatefinalExample

1. Private methods cannot be Overridden
2. Final Methods cannot be Overridden
3. Static Methods cannot be Overridden

|  |
| --- |
| **package** polymorphism;  **public** **class** abc {  **public** **static** **void** main(String[] args) {  def parent = **new** ghi();  parent.*show*();  parent.print();  parent.sample();  }  }  **class** def  {  **public** **static** **void** show()  {  System.***out***.println("def - Static method - Parent");  }  **private** **void** display()  {  System.***out***.println("def - private method - Parent");  }    **final** **public** **void** print()  {  System.***out***.println("def - final method - parent");  }  **public** **void** sample()  {  System.***out***.println("def - Instance method - parent");  }  }  **class** ghi **extends** def  {  @Override  **public** **static** **void** show() ====🡺 Compiler Error because it is Static method  {  System.***out***.println("def - Static method - Child");  }  @Override  **private** **void** display() ==🡺 Compiler Error because it is Private method  {  System.***out***.println("def - private method - Child");  }    @Override  final public void print() ==🡺 Compiler error because it is final method  {  System.out.println("def - final method - Child");  }    @Override  **public** **void** sample()  {  System.***out***.println("def - Instance method - Child");  }  } |