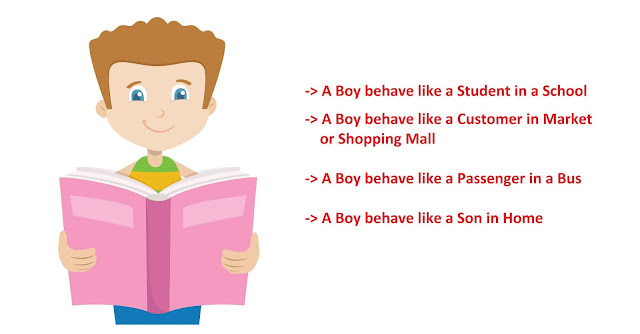
**Different definitions of Polymorphism are:**

1. *Polymorphism*lets us perform a single action in different ways.
2. *Polymorphism*allows you to define one interface and have multiple implementations
3. We can create functions or reference variables that behave differently in a different programmatic context.
4. *Polymorphism*means many forms.

**Polymorphism Real-World Example**

Suppose if you are in a classroom that time you behave like a **student**, when you are in the market at that time you behave like a **customer**, when you at your home at that time you behave like a **son** or **daughter**, Here one person present in different-different behaviors.

****

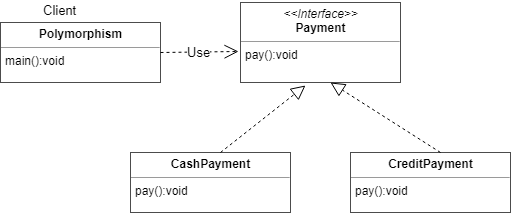
Let's understand the polymorphism with examples.

**2. Implementation with Examples**

**Example: Payment Processing Example**

In this **Payment Processing Example**, applying runtime polymorphism results in many forms at runtime.

In the below source code, the single payment "p" instance can be used to pay by cash and credit card, payment *p* instance takes many forms here.

****

public class Polymorphism {

public static void main(String[] args) {

// Here the runtime polymorphism fundamental is not applied,

// as it is of single CashPayment form

CashPayment c = new CashPayment();

c.pay();

// Now the initialization is done using runtime polymorphism and

// it can have many forms at runtime

// Single payment "p" instance can be used to pay by cash and credit card

Payment p = new CashPayment();

p.pay(); // Pay by cash

p = new CreditPayment();

p.pay(); // Pay by creditcard

}

}

/\*\*

\* This represents payment interface

\*/

interface Payment {

public void pay();

}

/\*\*

\* Cash IS-A Payment type

\*

\* @author tirthalp

\*

\*/

class CashPayment implements Payment {

// method overriding

@Override

public void pay() {

System.out.println("This is cash payment");

}

}

/\*\*

\* Creditcard IS-A Payment type

\*/

class CreditPayment implements Payment {

// method overriding

@Override

public void pay() {

System.out.println("This is credit card payment");

}

}

**3. Types of Polymorphism in Java**

1. Compile time polymorphism or method overloading or static banding
2. Runtime polymorphism or method overriding or dynamic binding

When a type of object is determined at a compiled time(by the compiler), it is known as *static binding*.

When a type of object is determined at run-time, it is known as *dynamic binding*.

**3.1 Compile-time Polymorphism**

If the class contains two or more methods having the same name and different arguments then it is *method overloading*.

The compiler will resolve the call to a correct method depending on the actual number and/or types of the passed parameters The advantage of method overloading is to increases the readability of the program.

**Method Overloading: changing no. of arguments**

In this example, we have created two methods, first, add() method performs the addition of two numbers and a second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create an instance for calling methods.

class Adder {

static int add(int a, int b) {

return a + b;

}

static int add(int a, int b, int c) {

return a + b + c;

}

}

class TestOverloading1 {

public static void main(String[] args) {

System.out.println(Adder.add(11, 11));

System.out.println(Adder.add(11, 11, 11));

}

}

**Method Overloading: changing the data type of arguments**

In this example, we have created two methods that differ in data type. The first add method receives two integer arguments and the second add method receives two double arguments.

class Adder {

static int add(int a, int b) {

return a + b;

}

static double add(double a, double b) {

return a + b;

}

}

class TestOverloading2 {

public static void main(String[] args) {

System.out.println(Adder.add(11, 11));

System.out.println(Adder.add(12.3, 12.6));

}

}

**3.2 Runtime Polymorphism**

*Runtime polymorphism* is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before *Runtime Polymorphism.*

**Upcasting:**

****

When the reference variable of *Parent*class refers to the object of *Child*class, it is known as upcasting.

**For example:**

class A {}

class B extends A {}

class Demo {

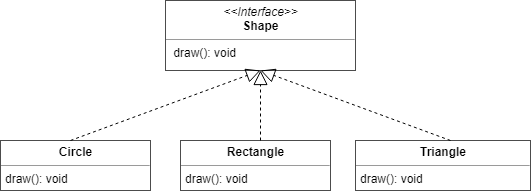
public static void main(String[] args) {

A a = new B(); //upcasting

}

}

**Java Runtime Polymorphism Example: Shape Example**

****

class Shape {

void draw() {

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

}

}

class Rectangle extends Shape {

void draw() {

System.out.println("drawing rectangle...");

}

}

class Circle extends Shape {

void draw() {

System.out.println("drawing circle...");

}

}

class Triangle extends Shape {

void draw() {

System.out.println("drawing triangle...");

}

}

class TestPolymorphism2 {

public static void main(String args[]) {

Shape s;

s = new Rectangle();

s.draw();

s = new Circle();

s.draw();

s = new Triangle();

s.draw();

}

}

Output:

drawing rectangle...

drawing circle...

drawing triangle...

**Java Runtime Polymorphism with Data Member**

The method is overridden by not applicable data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member *speedlimit*, we are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the *Parent*class always.

Runtime polymorphism can't be achieved by data members

class Bike {

int speedlimit = 90;

}

class Honda extends Bike {

int speedlimit = 150;

public static void main(String args[]) {

Bike obj = new Honda();

System.out.println(obj.speedlimit); //90

}

}

Output:

90

Let's try the below scenario: Here the *BabyDog*is not overriding the *eat()* method, so *eat()* method of Dog class is invoked. Note that if we are not using @Override annotation in this example.

class Animal {

void eat() {

System.out.println("animal is eating...");

}

}

class Dog extends Animal {

void eat() {

System.out.println("dog is eating...");

}

}

class BabyDog1 extends Dog {

public static void main(String args[]) {

Animal a = new BabyDog1();

a.eat();

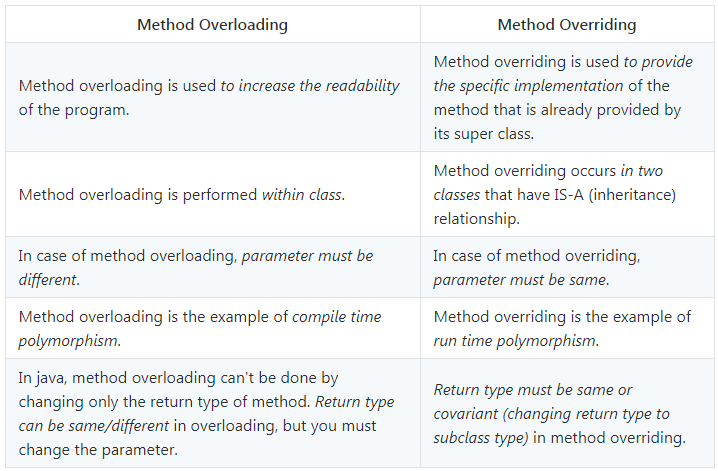
}

}

Output:

dog is eating...

**4. Polymorphism - Method Overloading vs Method Overriding**

****

**GitHub Repository**