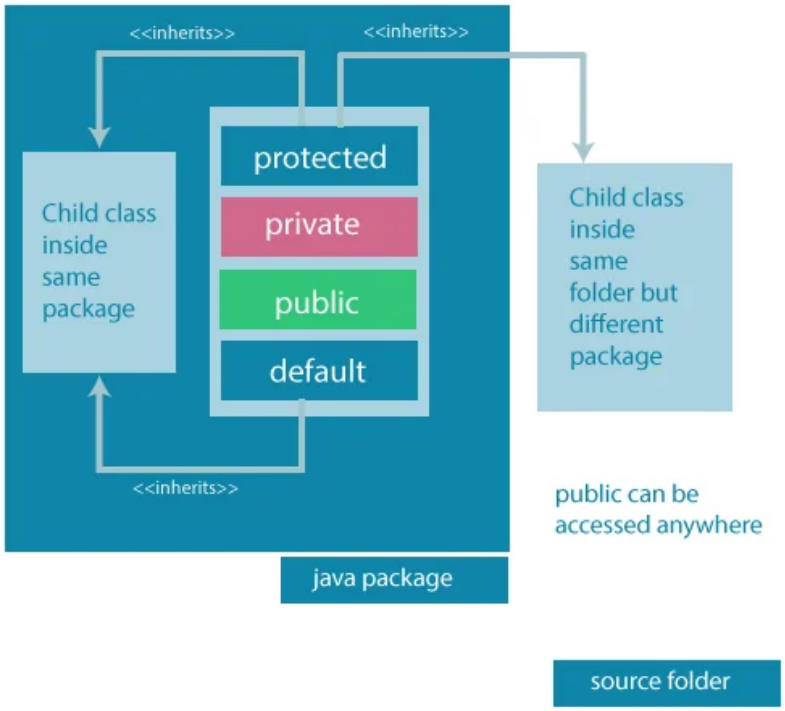
Lecture 4 - (OOP I)

Java language is a **Procedural programming**.

Access Modifiers / Visibility Modifiers.

In Java, access
modifiers are used to set

the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example,




```
class Animal {
    public void method1() {...}

   private void method2() {...}
}
```

In the above example, we have declared 2 methods: method1() and method2(). Here,

- method1 is `public` - This means it can be accessed by other classes.
- method2 is `private` - This means it can not be accessed by other classes.

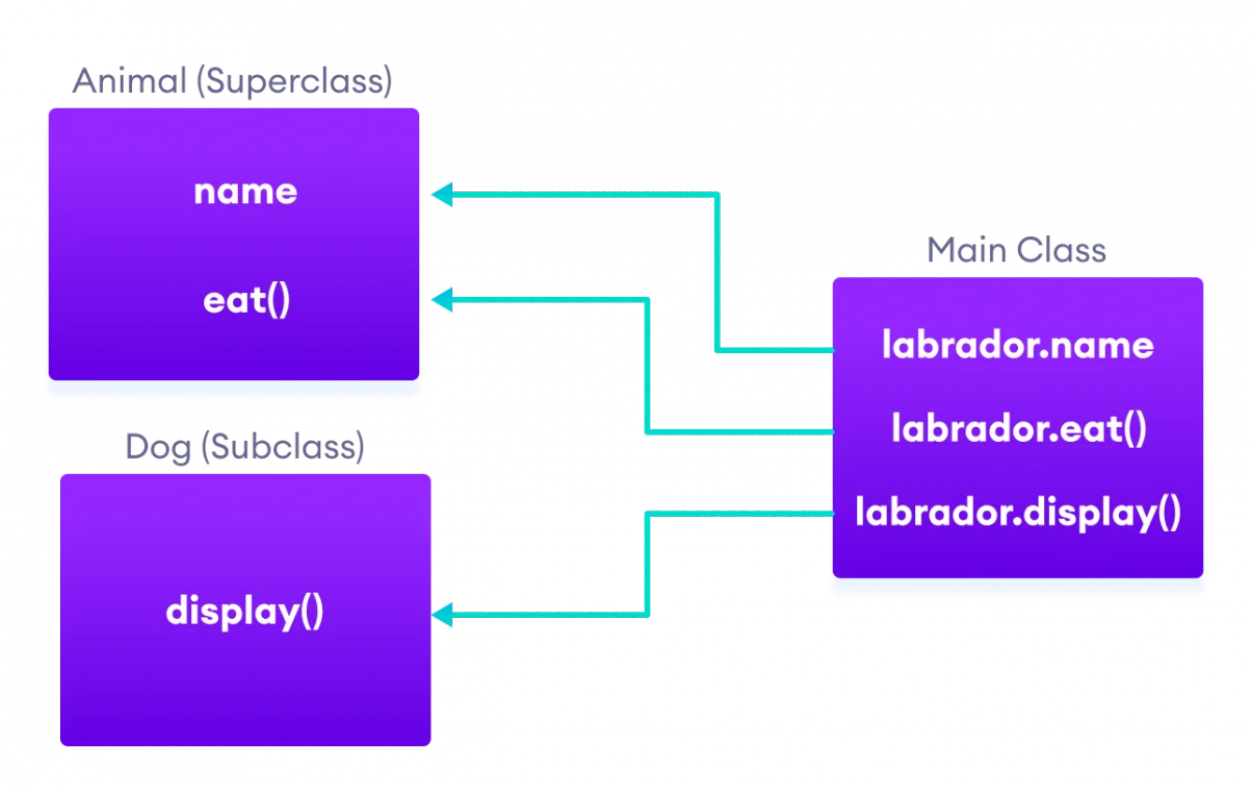
Types

Types of Modifier

Aa Modifier	Description
Default	declarations are visible only within the package (package private)

Aa Modifier	Description
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Inheritance



is-a relationship

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

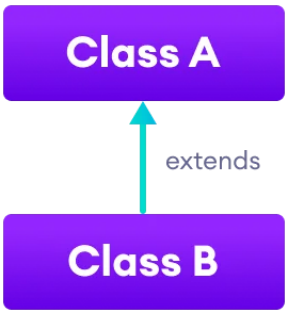
Why use Inheritance?

Reusability

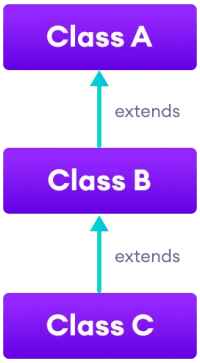
Method Overriding

Types of Inheritance

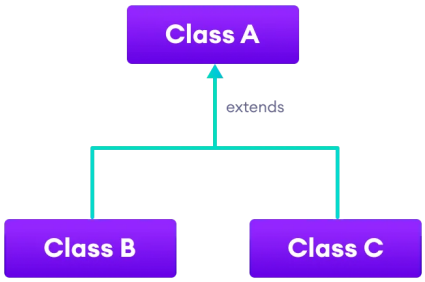
1. Single Inheritance



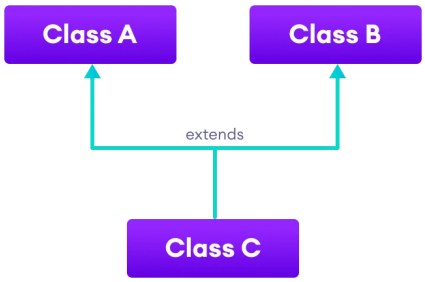
2. Multilevel Inheritance



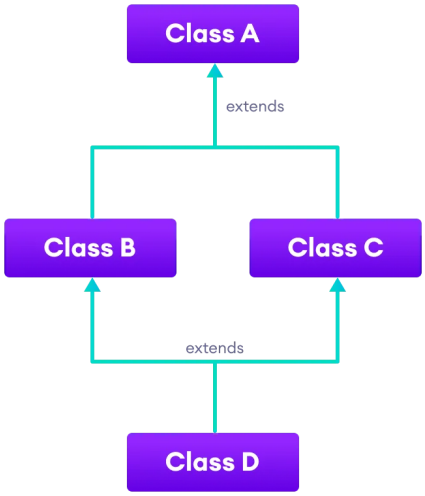
3. Hierarchical Inheritance



4. Multiple Inheritance



5. **Hybrid** Inheritance



Method Overloading

Defining methods having same name but different signatures

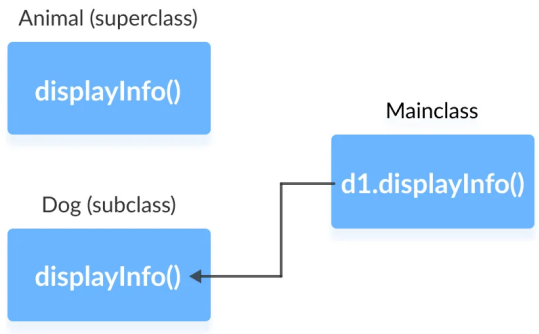
- Signature : Name + types of parameters, return type

```
class A{
  int a;
  int b;

  void hey(){
    sysout("Blah blah");
  }

  void hey(int a){
    sysout("Professor is a bit moody today");
  }
  int hey (int a, int b){
    sysout("Blah blah blah");
    if (prof == moody)
      return 1;
    return 0;
  }

}
```

Method Overriding

```
class Animal{
  public void displayInfo(){
    sysout("I am an Animal");
  }
}
class Dog extends Animal{
  @Override
  public void displayInfo(){
    sysout("I am a Dog");
  }
}

class Main{
  public static void main (String[] args){
  Dog d1 = new Dog();
  d1.displayInfo();
  }
}


Result --
I am a Dog.
```

When we call displayInfo() using the d1 object (object of the subclass), the method inside the subclass Dog is called. The displayInfo() method of the subclass overrides the same method of the superclass.



Overriding Rules

- Both superclass and subclass must have the same
  - method name,
  - return type
  - parameter list

- Can't override the method declared as **final** and **static.**
- We should always override abstract methods of the superclass
- Constructors in Java are not inherited. Hence, there is no such thing as constructor overriding in Java.

Incase of Abstract Method

In Java, abstract classes are created to be the superclass of other classes. And, if a class contains an abstract method, it is mandatory to override it.

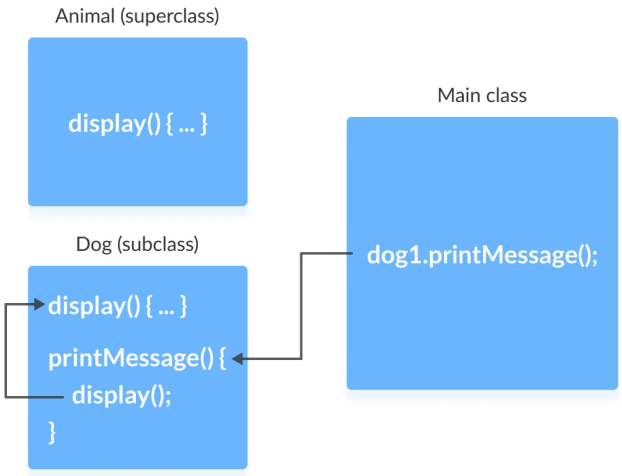
Super keyword

We can access the methods and fields of the superclass using super keyword

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      super.displayInfo();
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}

Result --
  I am an animal.
  I am a dog.
```

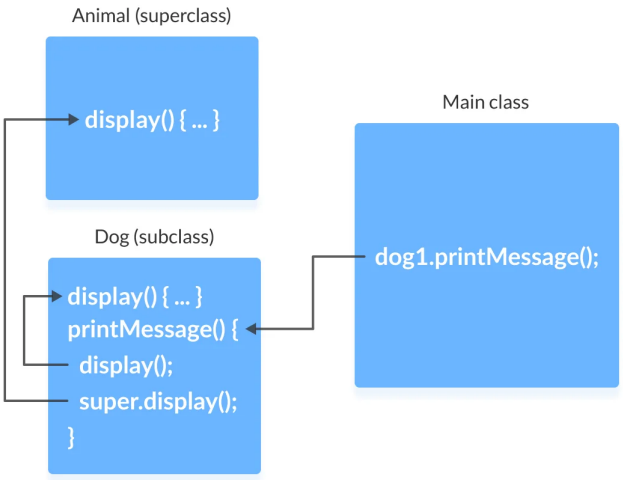
Uses of super keyword

- To call methods of superclass (i.e to override)




```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}

Result --
  I am a dog
```

- To access attributes (fields) of superclass




```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
Result --
```

```
  I am a dog
  I am an animal
```

- To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor

```
class Animal {

  // default or no-arg constructor of class Animal
  Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // default or no-arg constructor of class Dog
  Dog() {

    // calling default constructor of the superclass
    // Always the 1st line
    super();

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}


Result --
  I am a dog
  I am an animal
```

This Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,

```
class Main {
    int instVar;
```

```
    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }
}
```

Using this in Constructor Overloading

While working with constructor overloading, we might have to invoke one constructor from another constructor. In such a case, we cannot call the constructor explicitly. Instead, we have to use `this` keyword.

Here, we use a different form of this keyword. That is, `this()`. Let's take an example,

```
class Complex {

    private int a, b;

    // constructor with 2 parameters
    private Complex( int i, int j ){
        this.a = i;
        this.b = j;
    }

    // constructor with single parameter
    private Complex(int i){
        // invokes the constructor with 2 parameters
        this(i, i);
    }

    // constructor with no parameter
    private Complex(){
        // invokes the constructor with single parameter
        this(0);
    }

    @Override
    public String toString(){
        return this.a + " + " + this.b + "i";
    }

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

```
        // creating object of Complex class
        // calls the constructor with 2 parameters
        Complex c1 = new Complex(2, 3);

        // calls the constructor with a single parameter
        Complex c2 = new Complex(3);

        // calls the constructor with no parameters
        Complex c3 = new Complex();

        // print objects
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
    }
}
```

Output:

```
2 + 3i
3 + 3i
0 + 0i
```

In the above example, we have used `this` keyword,

- to call the constructor `Complex(int i, int j)` from the constructor `Complex(int i)`
- to call the constructor `Complex(int i)` from the constructor `Complex()`

Notice the line,

```
System.out.println(c1);
```

Here, when we print the object c1, the object is converted into a string. In this process, the `toString()` is called. Since we override the `toString()` method inside our class, we get the output according to that method.

One of the huge advantages of `this()` is to reduce the amount of duplicate code. However, we should be always careful while using `this()`.

This is because calling constructor from another constructor adds overhead and it is a slow process. Another huge advantage of using `this()` is to reduce the amount of duplicate code.

this with Getters and Setters

Another common use of `this` keyword is in *setters and getters methods* of a class. For example:

```
class Main {
   String name;

   // setter method
   void setName( String name ) {
       this.name = name;
   }

   // getter method
   String getName(){
       return this.name;
   }

   public static void main( String[] args ) {
       Main obj = new Main();

       // calling the setter and the getter method
       obj.setName("Toshiba");
       System.out.println("obj.name: "+obj.getName());
   }
}
```

Output:

```
obj.name: Toshiba
```

Final Keyword

Once any entity (variable, method or class) is declared `final`, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

1. Java final Variable

In Java, **we cannot change the value of a final variable.** For example,

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

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

2. Java final Method

Before you learn about final methods and final classes, make sure you know about the Java Inheritance.

In Java, the `final` **method cannot be overridden by the child class**. For example,

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

3. Java final Class

In Java, the **final class cannot be inherited by another class.** For example,

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class.

When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

Casting

Casting objects and the *instanceof* operator

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting)
  - E.g. **Person p = new Employee();**
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used
  - E.g. **Person p = new Employee(); Employee e = (Employee)p;**
  - If the superclass object is not an instance of the subclass, a runtime error occurs
  - It is a good practice to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the ***instanceof*** operator
- Casting an object reference does not create a new object

```
Person p = new Person();
Employee e = (Employee)p;

//run time Error
Result --
    Exception
```

```
Professor f = new Professor();
Person w = new Person();
sysout(f instanceof Professor)  // True

sysout(w instanceof Professor)  // True

sysout(f instanceof Person)  // False
```

The Object class

Every Java class has **object** as superclass

It has methods that are usually overridden:

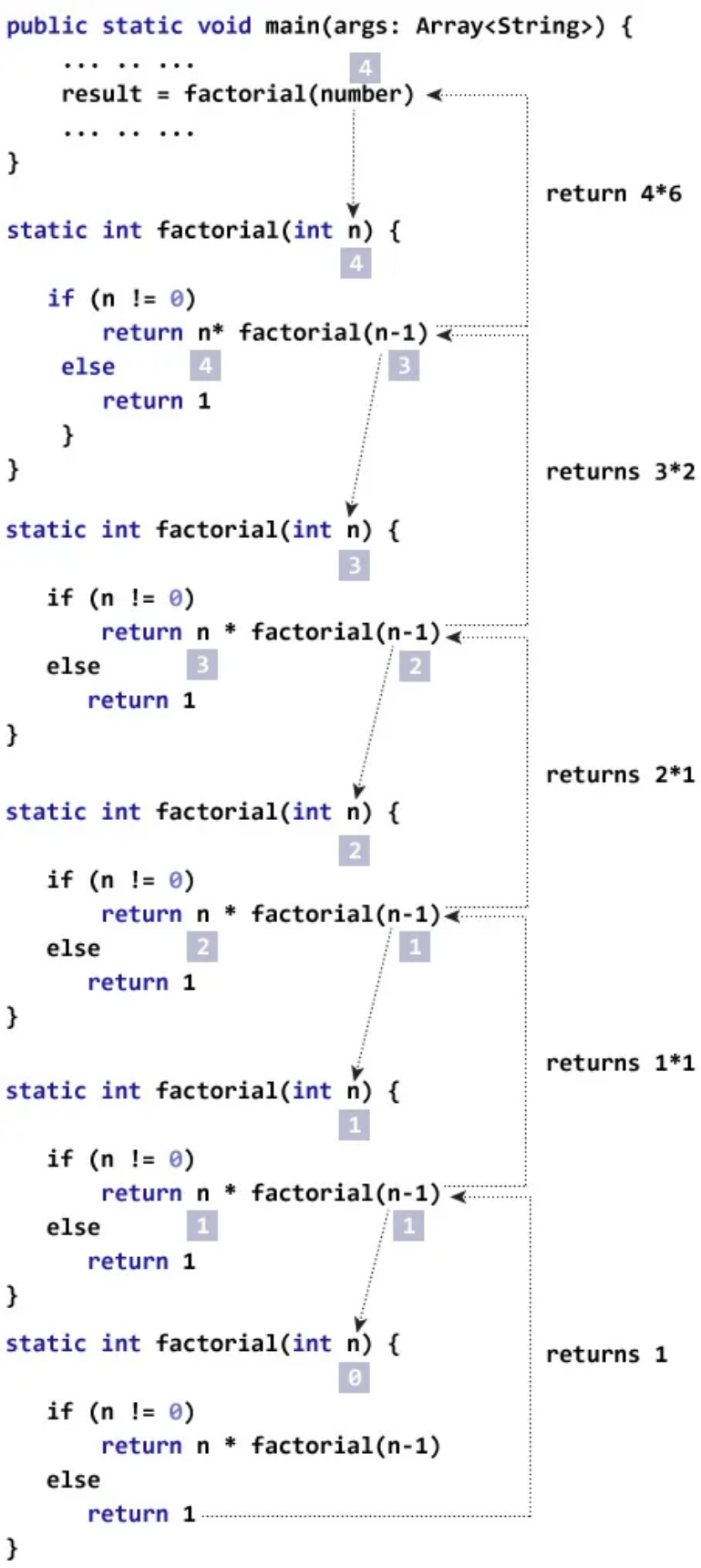
- **equals**

```
Person p1 = new Person('Yashank', 'Toronto');
Person p2 = new Person('Yashank', 'Toronto');
Person p3 = p1;
 // Person is a subclass of object
sysout(p1.equals(p2))
//Returns false, cause their reference is not the same.
// since, their addresses are not the same.
sysout(p1.equals(p3))     // True Cause their ref are same.
```

```
class Person{
String name;
String address;

  public boolean equals(Object obj){
    if (obj == null)
      return false;

    if (getClass != obj.getClass())
      return false;
    Person other = (Person)obj;
    if (name.equals(other.name) && address.equals(other.address))
      return true;
    return false;
    }
}
```

- **hashCode**
- **toString**

Recursion


public static void main(args: Array<String>) {
... .
result = factorial(number)
... .
}
4
static int factorial(int n) {
if (n != 0)
return n* factorial(n-1)
else
return 1
}
4
3
return 4*6
static int factorial(int n) {
if (n != 0)
return n * factorial(n-1)
else
return 1
}
3
2
returns 3*2
static int factorial(int n) {
if (n != 0)
return n * factorial(n-1)
else
return 1
}
2
1
returns 2*1
static int factorial(int n) {
if (n != 0)
return n * factorial(n-1)
else
return 1
}
1
1
returns 1*1
static int factorial(int n) {
if (n != 0)
return n * factorial(n-1)
else
return 1
}
0
returns 1

Advantages and Disadvantages of Recursion

When a recursive call is made, new storage locations for variables are allocated on the stack. As, each recursive call returns, the old variables and parameters are removed from the stack. Hence, recursion generally uses more memory and is generally slow.

On the other hand, a recursive solution is much simpler and takes less time to write, debug and maintain.

InstanceOf Operator

The `instanceof` operator in Java is used to check whether an object is an instance of a particular class or not.

Its syntax is

```
objectName instanceOf className;
```

Here, if objectName is an instance of className, the operator returns `true`. Otherwise, it returns `false`.

Example: Java instanceof

```
class Main {

  public static void main(String[] args) {

    // create a variable of string type
    String name = "Programiz";

    // checks if name is instance of String
      boolean result1 = name instanceof String; // prints true


    // create an object of Main
    Main obj = new Main();

    // checks if obj is an instance of Main
    boolean result2 = obj instanceof Main;  // prints true

  }
}
```

Java instanceof during Inheritance

We can use the `instanceof` operator to check if objects of the subclass is also an instance of the superclass. For example,

```
// Java Program to check if an object of the subclass
// is also an instance of the superclass

// superclass
class Animal {
}

// subclass
class Dog extends Animal {
}

class Main {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog d1 = new Dog();

    // checks if d1 is an instance of the subclass
    System.out.println(d1 instanceof Dog);        // prints true

    // checks if d1 is an instance of the superclass
    System.out.println(d1 instanceof Animal);     // prints true
  }
}
```