**super keyword in java**

The super keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

**Usage of java super Keyword**

super is used to refer immediate parent class instance variable.

super() is used to invoke immediate parent class constructor.

super is used to invoke immediate parent class method.

**1) super is used to refer immediate parent class instance variable.**

Problem without super keyword

class Vehicle{

int speed=50;

}

class Bike3 extends Vehicle{

int speed=100;

void display(){

System.out.println(speed);//will print speed of Bike

}

public static void main(String args[]){

Bike3 b=new Bike3();

b.display();

}

}

Test it Now

Output:100

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

//example of super keyword

class Vehicle{

int speed=50;

}

class Bike4 extends Vehicle{

int speed=100;

void display(){

System.out.println(super.speed);//will print speed of Vehicle now

}

public static void main(String args[]){

Bike4 b=new Bike4();

b.display();

}

}

Test it Now

Output:50

**2) super is used to invoke parent class constructor.**

The super keyword can also be used to invoke the parent class constructor as given below:

class Vehicle{

Vehicle(){System.out.println("Vehicle is created");}

}

class Bike5 extends Vehicle{

Bike5(){

super();//will invoke parent class constructor

System.out.println("Bike is created");

}

public static void main(String args[]){

Bike5 b=new Bike5();

}

}

Test it Now

Output:Vehicle is created

Bike is created

**Note: super() is added in each class constructor automatically by compiler.**

java super

As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

Another example of super keyword where super() is provided by the compiler implicitly.

class Vehicle{

Vehicle(){System.out.println("Vehicle is created");}

}

class Bike6 extends Vehicle{

int speed;

Bike6(int speed){

this.speed=speed;

System.out.println(speed);

}

public static void main(String args[]){

Bike6 b=new Bike6(10);

}

}

Test it Now

Output:Vehicle is created

10

**3) super can be used to invoke parent class method**

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

class Person{

void message(){System.out.println("welcome");}

}

class Student16 extends Person{

void message(){System.out.println("welcome to java");}

void display(){

message();//will invoke current class message() method

super.message();//will invoke parent class message() method

}

public static void main(String args[]){

Student16 s=new Student16();

s.display();

}

}

Test it Now

Output:welcome to java

welcome

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

class Person{

void message(){System.out.println("welcome");}

}

class Student17 extends Person{

void display(){

message();//will invoke parent class message() method

}

public static void main(String args[]){

Student17 s=new Student17();

s.display()

**Final Keyword In Java**

**Final variable**

**Final method**

**Final class**

Final can be:

variable

method

class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

final keyword in java

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

class Bike9{

final int speedlimit=90;//final variable

void run(){

speedlimit=400;

}

public static void main(String args[]){

Bike9 obj=new Bike9();

obj.run();

}

}//end of class

Test it Now

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

class Bike{

final void run(){System.out.println("running");}

}

class Honda extends Bike{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda honda= new Honda();

honda.run();

}

}

Test it Now

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

final class Bike{}

class Honda1 extends Bike{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda1 honda= new Honda();

honda.run();

}

}

Test it Now

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

class Bike{

final void run(){System.out.println("running...");}

}

class Honda2 extends Bike{

public static void main(String args[]){

new Honda2().run();

}

}

Test it Now

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

class Student{

int id;

String name;

final String PAN\_CARD\_NUMBER;

...

}

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

class Bike10{

final int speedlimit;//blank final variable

Bike10(){

speedlimit=70;

System.out.println(speedlimit);

}

public static void main(String args[]){

new Bike10();

}

}

Test it Now

Output:70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

class A{

static final int data;//static blank final variable

static{ data=50;}

public static void main(String args[]){

System.out.println(A.data);

}

}

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

class Bike11{

int cube(final int n){

n=n+2;//can't be changed as n is final

n\*n\*n;

}

public static void main(String args[]){

Bike11 b=new Bike11();

b.cube(5);

}

}

Test it Now

Output:Compile Time Error

Q) Can we declare a constructor final?

No, because constructor is never inherited

**Polymorphism in Java**

Polymorphism in java is a concept by which we can perform a single action by different ways. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

**Runtime Polymorphism in Java**

Runtime polymorphism or Dynamic Method Dispatch 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 reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

**Upcasting in java**

class A{}

class B extends A{}

A a=new B();//upcasting

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

class Bike{

void run(){System.out.println("running");}

}

class Splender extends Bike{

void run(){System.out.println("running safely with 60km");}

public static void main(String args[]){

Bike b = new Splender();//upcasting

b.run();

}

}

Test it Now

Output:running safely with 60km.

Real example of Java Runtime Polymorphism

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.

Java Runtime Polymorphism example of bank

Note: It is also given in method overriding but there was no upcasting.

class Bank{

int getRateOfInterest(){return 0;}

}

class SBI extends Bank{

int getRateOfInterest(){return 8;}

}

class ICICI extends Bank{

int getRateOfInterest(){return 7;}

}

class AXIS extends Bank{

int getRateOfInterest(){return 9;}

}

class Test3{

public static void main(String args[]){

Bank b1=new SBI();

Bank b2=new ICICI();

Bank b3=new AXIS();

System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());

}

}

Test it Now

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

Java Runtime Polymorphism with data member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.

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

Rule: Runtime polymorphism can't be achieved by data members.

class Bike{

int speedlimit=90;

}

class Honda3 extends Bike{

int speedlimit=150;

public static void main(String args[]){

Bike obj=new Honda3();

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

}

Test it Now

Output:90

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

class Animal{

void eat(){System.out.println("eating");}

}

class Dog extends Animal{

void eat(){System.out.println("eating fruits");}

}

class BabyDog extends Dog{

void eat(){System.out.println("drinking milk");}

public static void main(String args[]){

Animal a1,a2,a3;

a1=new Animal();

a2=new Dog();

a3=new BabyDog();

a1.eat();

a2.eat();

a3.eat();

}

}

Test it Now

Output: eating

eating fruits

drinking Milk

Try for Output

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();

}}

Test it Now

Output: Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked

Static Binding and Dynamic Binding

static binding and dynamic binding in java

Connecting a method call to the method body is known as binding.

There are two types of binding

static binding (also known as early binding).

dynamic binding (also known as late binding).

Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

int data=30;

Here data variable is a type of int.

2) References have a type

class Dog{

public static void main(String args[]){

Dog d1;//Here d1 is a type of Dog

}

}

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

class Animal{}

class Dog extends Animal{

public static void main(String args[]){

Dog d1=new Dog();

}

}

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

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

If there is any private, final or static method in a class, there is static binding.

Example of static binding

class Dog{

private void eat(){System.out.println("dog is eating...");}

public static void main(String args[]){

Dog d1=new Dog();

d1.eat();

}

}

Dynamic binding

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

Example of dynamic binding

class Animal{

void eat(){System.out.println("animal is eating...");}

}

class Dog extends Animal{

void eat(){System.out.println("dog is eating...");}

public static void main(String args[]){

Animal a=new Dog();

a.eat();

}

}

Test it Now

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

Java instanceof

java instanceof

Example of instanceof operator

Applying the instanceof operator with a variable the have null value

Downcasting with instanceof operator

Downcasting without instanceof operator

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

class Simple1{

public static void main(String args[]){

Simple1 s=new Simple1();

System.out.println(s instanceof Simple1);//true

}

}

Test it Now

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

class Animal{}

class Dog1 extends Animal{//Dog inherits Animal

public static void main(String args[]){

Dog1 d=new Dog1();

System.out.println(d instanceof Animal);//true

}

}

Test it Now

Output:true

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

class Dog2{

public static void main(String args[]){

Dog2 d=null;

System.out.println(d instanceof Dog2);//false

}

}

Test it Now

Output:false

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

Dog d=new Animal();//Compilation error

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

Dog d=(Dog)new Animal();

//Compiles successfully but ClassCastException is thrown at runtime

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

class Animal { }

class Dog3 extends Animal {

static void method(Animal a) {

if(a instanceof Dog3){

Dog3 d=(Dog3)a;//downcasting

System.out.println("ok downcasting performed");

}

}

public static void main (String [] args) {

Animal a=new Dog3();

Dog3.method(a);

}

}

Test it Now

Output:ok downcasting performed

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

class Animal { }

class Dog4 extends Animal {

static void method(Animal a) {

Dog4 d=(Dog4)a;//downcasting

System.out.println("ok downcasting performed");

}

public static void main (String [] args) {

Animal a=new Dog4();

Dog4.method(a);

}

}

Test it Now

Output:ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

Animal a=new Animal();

Dog.method(a);

//Now ClassCastException but not in case of instanceof operator

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

interface Printable{}

class A implements Printable{

public void a(){System.out.println("a method");}

}

class B implements Printable{

public void b(){System.out.println("b method");}

}

class Call{

void invoke(Printable p){//upcasting

if(p instanceof A){

A a=(A)p;//Downcasting

a.a();

}

if(p instanceof B){

B b=(B)p;//Downcasting

b.b();

}

}

}//end of Call class

class Test4{

public static void main(String args[]){

Printable p=new B();

Call c=new Call();

c.invoke(p);

}

}

Test it Now

Output: b method

**Abstract class in Java**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

Abstract class (0 to 100%)

Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

abstract class A{}

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

abstract void printStatus();//no body and abstract

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

abstract class Bike{

abstract void run();

}

class Honda4 extends Bike{

void run(){System.out.println("running safely..");}

public static void main(String args[]){

Bike obj = new Honda4();

obj.run();

}

}

Test it Now

running safely..

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.

A factory method is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

abstract class Shape{

abstract void draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

class Rectangle extends Shape{

void draw(){System.out.println("drawing rectangle");}

}

class Circle1 extends Shape{

void draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

class TestAbstraction1{

public static void main(String args[]){

Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method

s.draw();

}

}

Test it Now

drawing circle

Another example of abstract class in java

File: TestBank.java

abstract class Bank{

abstract int getRateOfInterest();

}

class SBI extends Bank{

int getRateOfInterest(){return 7;}

}

class PNB extends Bank{

int getRateOfInterest(){return 7;}

}

class TestBank{

public static void main(String args[]){

Bank b=new SBI();//if object is PNB, method of PNB will be invoked

int interest=b.getRateOfInterest();

System.out.println("Rate of Interest is: "+interest+" %");

}}

Test it Now

Rate of Interest is: 7 %

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

//example of abstract class that have method body

abstract class Bike{

Bike(){System.out.println("bike is created");}

abstract void run();

void changeGear(){System.out.println("gear changed");}

}

class Honda extends Bike{

void run(){System.out.println("running safely..");}

}

class TestAbstraction2{

public static void main(String args[]){

Bike obj = new Honda();

obj.run();

obj.changeGear();

}

}

Test it Now

bike is created

running safely..

gear changed

Rule: If there is any abstract method in a class, that class must be abstract.

class Bike12{

abstract void run();

}

Test it Now

compile time error

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

interface A{

void a();

void b();

void c();

void d();

}

abstract class B implements A{

public void c(){System.out.println("I am C");}

}

class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

public void d(){System.out.println("I am d");}

}

class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();

}}

Test it Now

Output:I am a

I am b

I am c

I am d

**Interface in Java**

**Interface**

**An interface in java is a blueprint of a class. It has static constants and abstract methods only.**

The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

**Java Interface also represents IS-A relationship.**

It cannot be instantiated just like abstract class.

**Why use Java interface?**

There are mainly three reasons to use interface. They are given below.

It is used to achieve fully abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.

**interface**

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.

relationship between class and interface

Simple example of Java interface

In this example, Printable interface have only one method, its implementation is provided in the A class.

interface printable{

void print();

}

class A6 implements printable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

A6 obj = new A6();

obj.print();

}

}

Test it Now

Output:Hello

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

multiple inheritance in java

interface Printable{

void print();

}

interface Showable{

void show();

}

class A7 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

A7 obj = new A7();

obj.print();

obj.show();

}

}

Test it Now

Output:Hello

Welcome

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

interface Printable{

void print();

}

interface Showable{

void print();

}

class TestTnterface1 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestTnterface1 obj = new TestTnterface1();

obj.print();

}

}

Test it Now

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

interface Printable{

void print();

}

interface Showable extends Printable{

void show();

}

class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

Testinterface2 obj = new Testinterface2();

obj.print();

obj.show();

}

}

Test it Now

Hello

Welcome

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

public interface Serializable{

}

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

interface printable{

void print();

interface MessagePrintable{

void msg();

}

}

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract**methods. | Interface can have **only abstract** methods. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or constructor**. |
| 5) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

//Creating interface that has 4 methods

interface A{

void a();//bydefault, public and abstract

void b();

void c();

void d();

}

//Creating abstract class that provides the implementation of one method of A interface

abstract class B implements A{

public void c(){System.out.println("I am C");}

}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

public void d(){System.out.println("I am d");}

}

//Creating a test class that calls the methods of A interface

class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();

}}

Test it Now

Output:

I am a

I am b

I am c

I am d