

Day x - Notes

Java Training



April 1, 2023

Dpoint

[Company address]

# Polymorphism

# Method Overloading:

Methods having same name but differ in parameters. – data types or count.

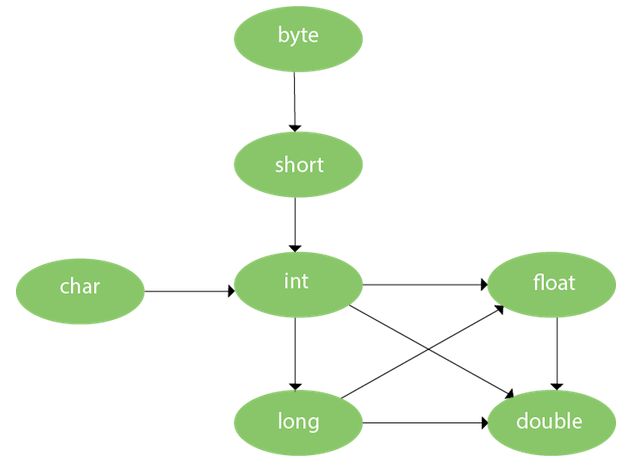
Increases the readability of the program.

Method overloading not feasible with change in return type due to ambiguity of execution.

Overloading of main method is allowed, but the method that jvm calls is the method having String[] as argument.

While calling a method which is overloaded, if no params are matched with exact data types.

Then implicit type promotion happens and matched method signature is invoked.



# Method Overriding :

If subclass has provided method implementation for method which is present in the parent.

Usage:

* Customized, specific implementation by each sub class as per req.
* Used for runtime polymorphism.

Rules:

* Subclass should have same name as parent class.
* Method must have same params as parent.
* There should be an IS-A relationship.

Notes:

Static method cannot be overridden. Because static methods are bound to the class.

Static belongs to the class area and instance belongs to the heap area.

# Covariant return type:

It helps in preventing runtime class cast exception, by allowing us to do the method overriding by changing the return type of the overriding method.

In other words, if a parent class method returns an object of type A, then the subclass method can override it and return an object of a subtype of A.

**Advantages:**

1. Avoids confusing type casting
2. Liberty to have more to the point return types
3. Prevents run time class cast exception

# Super:

It’s a keyword to refer immediate parent class. – super.prop1

Can be used to invoke parent class method. – super.method1()

Can be used to invoke parent class constructor. – super()

Super is provided by complier implicitly,

When calling default constructor of subclass, default constructor of parent class is auto called.

# Instance Initializer block:

Initialize the instance data member.

It run each time when object of the class is created.

This can be used to perform many initial works as well.

**Difference between constructor and initialization block:**

Instance Initializer Block:

* It is a block of code enclosed in curly braces that is executed whenever an instance of the class is created, before the constructor is called.
* It is useful for initializing final variables, performing common initialization tasks for multiple constructors, and initializing complex objects.
* It can access instance variables and methods of the class.
* It is executed on a per-instance basis.

Constructor:

* It is a special method that is called when an instance of the class is created.
* It is used to initialize instance variables and perform other initialization tasks.
* It can be overloaded to provide multiple ways of creating instances of the class with different initializations.
* It can access instance variables and methods of the class.
* It is executed after the instance initializer block, if present.

**Note: The java compiler copies the code of instance initializer block in every constructor.**

# Final:

Final keyword can be used for class, method, variable.

Final variable can be initialized in the constructor only, it cannot be changed.

 final int speedlimit=90;

Final method: these methods cannot be overridden.

Final method is inherited but cannot be overridden.

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

Final class: this class cannot be extended.

final class MyClass {

private int x;

public MyClass(int x) {

this.x = x;

}

public int getX() {

return x;

}

}

**Static blank final variable:**

* Uninitialized final variable is called blank final variable
* 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.
* Constructor cannot be declared as final because it is never inherited.

**class** A{

**static** **final** **int** data;//static blank final variable

**static**{ data=50;}

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

    System.out.println(A.data);

 }

}

# Runtime polymorphism:

Performing single action in different manner.

Poly = “many”

Morphs = “forms”.

**2 types of polymorphism:**

1. Compile time polymorphism
2. Runtime polymorphism

**Polymorphism in java can be implemented using:**

1. Method overloading
2. Method overriding

Runtime Polymorphism also known as Dynamic method dispatch – process in which call to overridden method is resolved at runtime rather than compile time.

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.

**For runtime polymorphism, we need to understanding UPCASETING:** reference variable of parent class refers to the Object of child class, known as upcasting.

1. **interface** I{}
2. **class** A{}
3. **class** B **extends** A **implements** I{}

relationship:

B IS-A A

B IS-A I

B IS-A Object

**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

}

}

# Static binding and Dynamic binding:

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

2 types of binding:

* 1. Static binding (Early binding) – type of object is determined at compile-time.
  2. Dynamic binding (Late binding) – type of object is determined at runtime-time.

**Static binding:**

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.

**Dynamic binding:**

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

# Java Instanceof:

 test whether the object is an instance of the specified type

**class** Simple1{

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

 Simple1 s=**new** Simple1();

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

 }

}

**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

 }

}

**Downcasting with java instanceof operator is compilation error:**

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

**downcasting by typecasting, ClassCastException is thrown at runtime:**

Dog d=(Dog)**new** Animal();

//Compiles successfully but ClassCastException is thrown at runtime