

UITS
S02
OOP

-

JAVA
Note

(use PDF/eBook Reader for Document Outline / Index Navigation)

Index

Index

- 1 - Theory + Question (OOP, Java)
- 3 - Syntax Point
- 4 - Question + Problem
- 5 - Lab - 01 - Variables (+User Input, Type Casting)
- 6 - Problem + HW
- 8 - (Loop, Array, Array methods)
- 10 - Lab - 02 - Datatypes (+Method Overloading, scope)
- 12 - Constructors
- 15 - Constructor overloading
- 16 - variable, operators
- 17 - Lab - 03 -
- 17 - Method overloading
- 18 - Type casting, "this"
- 20 - #define
- 21 - Lab - 04 : C++ Encapsulation
OOP - Inheritance
- 23 - Hierarchical inheritance, method overriding,
super keyword
- 25 - Lab - 05 :
- 26 - Encapsulation, access modifiers, getter, setter
- 27 - Lab - 06 : User Input
- 305 - Overriding

29 - **LAB** - 07 (mat 900) intro + project - 10

30 - Default - access modifiers, Protected - 8

(31 - Protected - 10 - **Do I** - 8)

32 - Polymorphism, Complete time - 10

34 - Runtime (easter) - 10

37 - **Lab** - 08 Inheritance concepts, calling parent construct through child class - 10 - **Do I** - 10

38 - Abstraction in Java - 10

Barbaryco bonelli (conservation) - 10 - **Do I** - 10

bonytail - 10

10 - **Do I** - 10

10 - **Do I** - 10

10 - **Do I** - 10

problem - 10

Index

1-

2- static keyword

4- Lab-9 - interface, multiple inheritance

5- Interface in Java

11- final keyword

12- Exception Handling

14- Lab-11 : Exception Handling

16- Multiple catch block

16- Throw and Throws

17- Thread concept in Java

19- updates after Page-12

{ 8- Lab }

- 10 : Project + Swing Polymorphism

Mid Syllabus

Tanta AKter Setu(TAS) | g class: w5Lknst
Tuesday at room 203

27.08.24

Review All Subject Khata P-7]

OOP works with Data and function

SPL "Process (and structure)"

OOP features: ① Abstraction

② Encapsulation

③ Inheritance

④ Polymorphism

Reusable functions, Slower than SPL, case sensitive

Class and Object

class > object

member variable
member object

member data
member function

Modular Approach, bottom-up design technique.

programming

Object Oriented Languages

Q. Procedural Language

⇒ PDF

HF.80.F.S

Procedural Programming:

- ① Programs divided into functions
- ② Data is not hidden, can be accessed by external functions
- ③ Top-down approach in program design
- ④ Data communicates with each other through functions
- ⑤ Emphasis on procedure rather than data

Object Oriented Programming:

- ① Programs divided into objects.
- ② Data is hidden, cannot be accessed by external functions.
- ③ Bottom-up approach in the program design.
- ④ Objects communicate with each other through functions.
- ⑤ Emphasis on data rather than procedure.

747 K =

CHAPTER

2

An Overview of Java

As in all other computer languages, the elements of Java do not exist in isolation. Rather, they work together to form the language as a whole. However, this interrelatedness can make it difficult to describe one aspect of Java without involving several others. Often a discussion of one feature implies prior knowledge of another. For this reason, this chapter presents a quick overview of several key features of Java. The material described here will give you a foothold that will allow you to write and understand simple programs. Most of the topics discussed will be examined in greater detail in the remaining chapters of Part I.

Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed. The first way is called the *process-oriented model*. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex.

To manage increasing complexity, the second approach, called *object-oriented programming*, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Abstraction

An essential element of object-oriented programming is *abstraction*. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the individual parts. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system might consist of a radio, a CD player, and/or MP3 player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to *do something*. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a

well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class*. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables*. The code that operates on that data is referred to as *member methods* or just *methods*. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a *method*, a C/C++ programmer calls a *function*.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public interface* of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by *hierarchical* (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals.

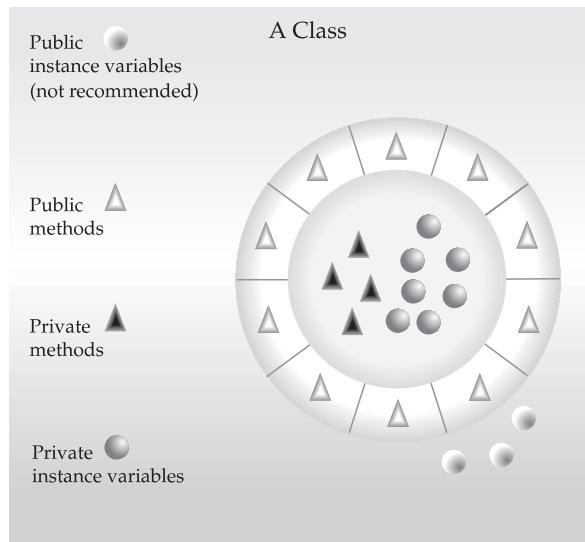
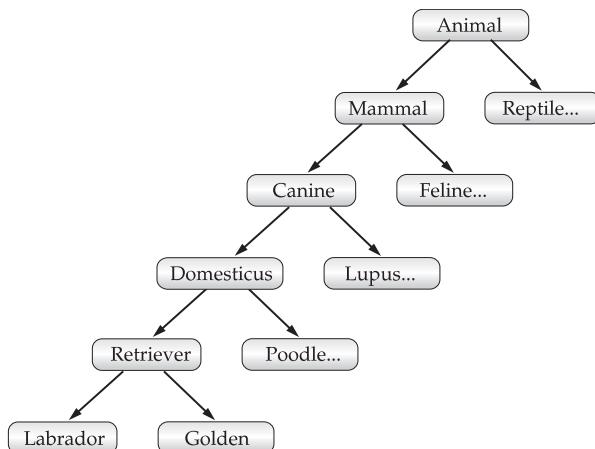


Figure 2-1 Encapsulation: public methods can be used to protect private data.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass*.

Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy*.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.



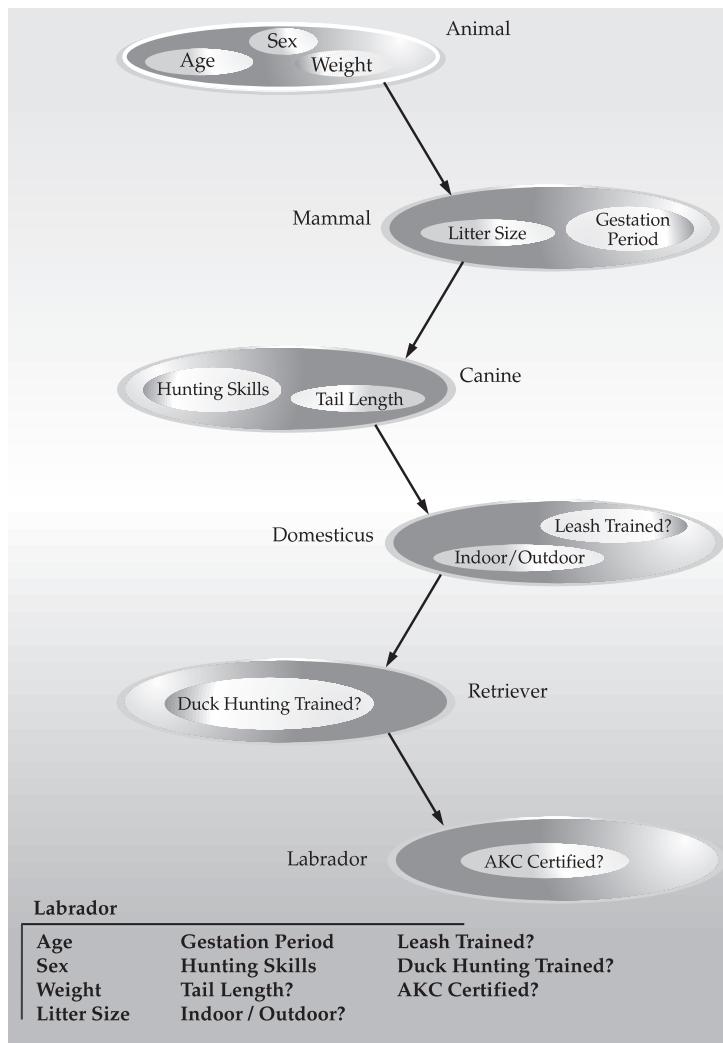


Figure 2-2 Labrador inherits the encapsulation of all its superclasses.

Polymorphism

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*. It is the compiler’s job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog’s sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog’s nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Polymorphism, Encapsulation, and Inheritance Work Together

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scalable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Of the two real-world examples, the automobile more completely illustrates the power of object-oriented design. Dogs are fun to think about from an inheritance standpoint, but cars are more like programs. All drivers rely on inheritance to drive different types (subclasses) of vehicles. Whether the vehicle is a school bus, a Mercedes sedan, a Porsche, or the family minivan, drivers can all more or less find and operate the steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people can even manage the difference between a stick shift and an automatic, because they fundamentally understand their common superclass, the transmission.

People interface with encapsulated features on cars all the time. The brake and gas pedals hide an incredible array of complexity with an interface so simple you can operate them with your feet! The implementation of the engine, the style of brakes, and the size of the tires have no effect on how you interface with the class definition of the pedals.

The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers to offer a wide array of options on basically the same vehicle. For example, you can get an antilock braking system or traditional brakes, power or rack-and-pinion steering, and a 4-, 6-, or 8-cylinder engine, or an EV. Either way, you will still press the brake pedal to stop, turn the steering wheel to change direction, and press the accelerator when you want to move. The same interface can be used to control a number of different implementations.

As you can see, it is through the application of encapsulation, inheritance, and polymorphism that the individual parts are transformed into the object known as a car. The same is also true of computer programs. By the application of object-oriented principles, the various parts of a complex program can be brought together to form a cohesive, robust, maintainable whole.

As mentioned at the start of this section, every Java program is object-oriented. Or, put more precisely, every Java program involves encapsulation, inheritance, and polymorphism. Although the short example programs shown in the rest of this chapter and in the next few chapters may not seem to exhibit all of these features, they are nevertheless present.

As you will see, many of the features supplied by Java are part of its built-in class libraries, which do make extensive use of encapsulation, inheritance, and polymorphism.

~~user~~
Class is a ~~usually~~ defined datatype
Method is function in other Languages
OOP has bigger file size.

HW → Applications of OOP

Java

Definition: Object oriented distributed → [IPvt, IPub] arranged in hierarchy / package structure

Interpreted → compiled into JVM code "Bytecode"

robust → no pointers / stack try / catch / finally sentinel

secure → real-time compilation error

architecture neutral

portable

high performance

multithreaded

dynamite language

↓
machine code (executing)

Package structure → header file

J2SE, J2EE, J2ME

Q. Why is Java a platform independent language?

⇒ .java → bytecode → .class

/machine code

↓

platform depended

conversion by Java interpreter

.java file is saved by the class name.

One IDE → Netbeans

bytecode file is created after compiling

Java Syntax

```

public class Welcome {
    public static void main(String[] args) {
        System.out.println("Hello Java");
        System.out.println("I like " + args[0] + " Java");
        String strDepartment = "CSE";
        System.out.print("We study " + strDepartment
                        + "\n");
    }
}

```

private
(access modifiers)

object independent

Review: Marks allotted

Separate khata for Java

There'll be Lab pressure just before final so

CT would be scheduled 1/2 weeks before

term exams (suggestions)

Protection (secure?) → Public, private, protected, default

reusability → Class > type > object

problem → Lab projects.

Complex Engineering

OBE, PLO, CLO

Access Issue (Encapsulation)

01.08.24

Object state, behaviour and identity

```
class Student {  
    int ID;  
    String Name;  
    void printInfo() {  
        System.out.println(ID);  
        System.out.println(Name);  
    }  
}
```

C₉ Headline → Java 9 package

utility package Scanner
class import Scanner

Exception handling → unknown cases

↓ Try/catch block

Java → Platform independent

JVM → Platform dependent

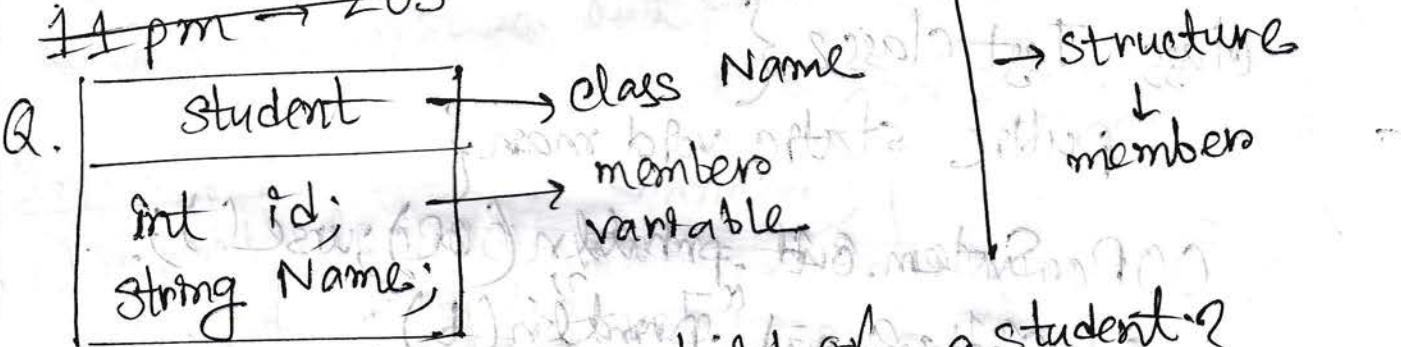
02.09.24

Q. Why is Java platform independent?

Q. Why is JVM platform dependent?

Q. What is Bytecode?

~~new (operator) = memory allocation~~



Show the name and id of a student?

→ public class Student {

 int id;

 String Name; }

Class Testclass{
 public static void main{

 student s1 = new student();

 class name object name (s1.id);

 System.out.println(s1.Name);

 System.out.println();

 }}

G.

| |
|--------------|
| oop course |
| Java(String) |
| C++(String) |

 → main class name "Test class"
show the language Name

class OOP course

String a; String b; = { }

```
class TestClass2 {
```

```
public static void main{
```

```
OOPcourse c1 = new OOPcourse();
```

Statement: `c1.a = "Java";`

c1.b = "C++"; cout < "int wate

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

~~System.out.println(c*t+b);~~

61. a. "J. J." - 161 to
G. Smith, 1894, note

• *Urticaria* *caerulea* *caerulea* *caerulea*

1. *Chlorophytum comosum* L. (Liliaceae) - *Chlorophytum comosum* L. (Liliaceae)

9/1 (br. 121) 2006

(front & 2) at 4 strong-tvs. water 2

is going to try two methods

The Java Buzzwords

No discussion of Java's history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java managed to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, were kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to

create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer will often manually allocate and free dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform

solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java

How is Java platform independent?

Last Updated : 25 Jul, 2024

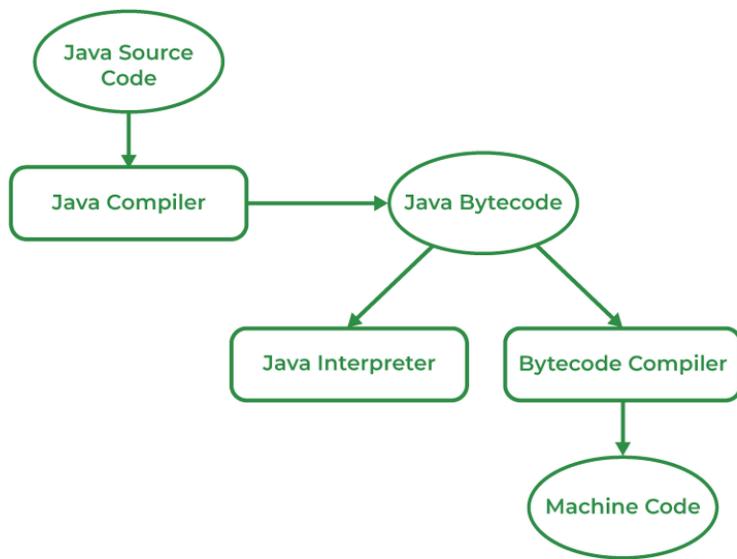


The meaning of Java platform-independent is that the Java compiled code(byte code) can run on all operating systems. A program is written in a language that is human-readable. It may contain words, phrases, etc. which the machine does not understand. For the source code to be understood by the machine, it needs to be in a language understood by machines, typically a machine-level language. So, here comes the role of a compiler. The compiler converts the high-level language (human language) into a format understood by the machines.

Therefore, a compiler is a program that translates the source code for another program from a programming language into executable code. This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the **Java Byte Code**.

Step-by-Step Execution of Java Program

- Whenever a program is written in JAVA, the java compiles it.
- The result of the JAVA compiler is the **.class file or the bytecode** and not the machine's native code (unlike the C compiler).
- The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
- And finally, the program runs to give the desired output.



In the case of C or C++ (languages that are not platform independent), the compiler generates an executable file (such as a .exe file) which is both OS-dependent and CPU-dependent. When we try to run this executable file on another OS or CPU architecture, it does not run, since it is specifically compiled for the target operating system and hardware architecture, making it incompatible with others.

Why Java is platform-independent but JVM is platform dependent?

In Java, the main point here is that the JVM depends on the operating system – so if you are running Mac OS X you will have a different JVM than if you are running Windows or some other operating system. This fact can be verified by trying to download the JVM for your particular machine – when trying to download it, you will be given a list of JVM corresponding to different operating systems, and you will obviously pick whichever JVM is targeted for the operating system that you are running. So we can conclude that JVM is platform-dependent and it is the reason why Java is able to become “Platform Independent”.

Important Points:

- In the case of Java, **it is the magic of Bytecode that makes it platform-independent.**
- This adds to an important feature in the JAVA language termed **portability**. Every system has its own JVM which gets installed automatically when the JDK

software is installed. For every operating system separate JVM is available which is capable to read the .class file or byte code.

- An important point to be noted is that while **JAVA is a platform-independent language, the JVM is platform-dependent**. Different JVM is designed for different OS and byte code is able to run on different OS.

Note: As JVM is platform dependent because of which Java is not considered completely platform-independent.

OOP

Object and Class Example: Initialization through method

a. Student

int rollno;

String name;

void Record(int r,

String n.

void DisplayInformation()

Parameterized
method

arguments
in C

(alternative)
way

→ class student {

int rollno;

String Name;

void Record(int r, String n) {

roll no = r;

name = n;

}

void DisplayInformation() { "Roll No" +

System.out.println("Roll No" + "Name" +
+ name); }

}

main member

(method?)

900

class TestStudent {
public static void main {

objects [student s1 = new Student();
student s2 = new Student();

s1.Record(111, "Kabir");

s2.Record(112, "Ans");

s1.DisplayInformation();

s2.DisplayInformation();

} } // student class

Output

Roll no = 111 Name = Kabir

Roll no = 112 Name = Ans

HW Q. Student

```

int roll;
String Name;
String Address;
Void Record(int r,
String n, String a)
Void DisplayInformation()
    
```

Mark method Name
Class Test
show the ID, Name,
Address of 3 students.

class student {

```

int roll;
String Name;
String Address; }
```

Void Record(int r, String n, String a) {

roll = r;

Name = n;

Address = a; }

Void DisplayInformation() {

System.out.println("roll = " + roll + "\nName = "
+ Name + "\nAddress = " + Address); }

class classTest

{ void main (String [] args) {

public static

student s1 = new student();

student s2 = new student();

s1.record(01, "Nabit", "Banani");

s2.record(02, "Habib", "Gulshan");

s3.record(03, "Labib", "Badda");

s1.displayInformation();

s2.displayInformation();

s3.displayInformation();

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

The **data, or variables**, defined within a **class** are called ***instance variables***. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called **members** of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the **same general form as main()**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main()** method. Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications don't require a **main()** method at all.

OOP!



17.11.24

Q. object vs structure?

→ how about a brown state w/

... brown - brown brown yellow

allison of brown past state what was she

brown brown

brown brown

What's the Difference?

Object and structure are both fundamental concepts in programming and data organization. Objects are instances of classes that encapsulate data and behavior, allowing for modular and reusable code. Structures, on the other hand, are collections of related data fields that are stored together in memory. While objects are typically used for more complex data structures and interactions, structures are often used for simpler data storage and manipulation. Both objects and structures play important roles in organizing and managing data in software development.

Photo by Florian Klauer on Unsplash

Objects are instances of classes that encapsulate data and behavior??

| Attribute | Object | Structure |
|---------------|---|---|
| Definition | Instance of a class with its own state and behavior | Arrangement of parts or elements in a complex whole |
| Composition | Can contain other objects as properties | Can contain other structures as elements |
| Encapsulation | Combines data and methods into a single unit | Can encapsulate data and functions ?? |
| Flexibility | Can be easily modified and extended | Can be rigid or flexible depending on design |
| Usage | Used in object-oriented programming | Used in data organization and representation |

Object as Parameter in Java

Similar to primitive types, Java makes it easier to give objects as parameters to methods. It is crucial to recognize that sending an object as an argument transmits merely a reference to the item—not a duplicate of it. It means that any changes made to the object inside the method will have an immediate impact on the original object. In this section, we will discuss **passing an object as a parameter in Java**.

Object Passing as Parameters

Pass-by-Value

1. Java is a pass-by-value language.
2. When forwarding objects, only the object's reference—not the actual object—is duplicated.

Encapsulation

1. Object passing supports encapsulation.
2. Methods act as interfaces, controlling access to the internal state of an object.

Immutability Considerations

1. Care is needed with immutable objects.
2. Methods should avoid modifying the state and return new instances if necessary.

Null Checks and Defensive Programming

1. Include null checks to handle potential null object parameters.
2. Defensive programming ensures robustness and prevents null pointer exceptions.

Code Reusability and Modularity:

1. Object passing enhances code reusability.
2. Promotes modularity by allowing methods to operate on different object instances.

Passing and Returning Objects in Java

Last Updated : 27 Dec, 2021



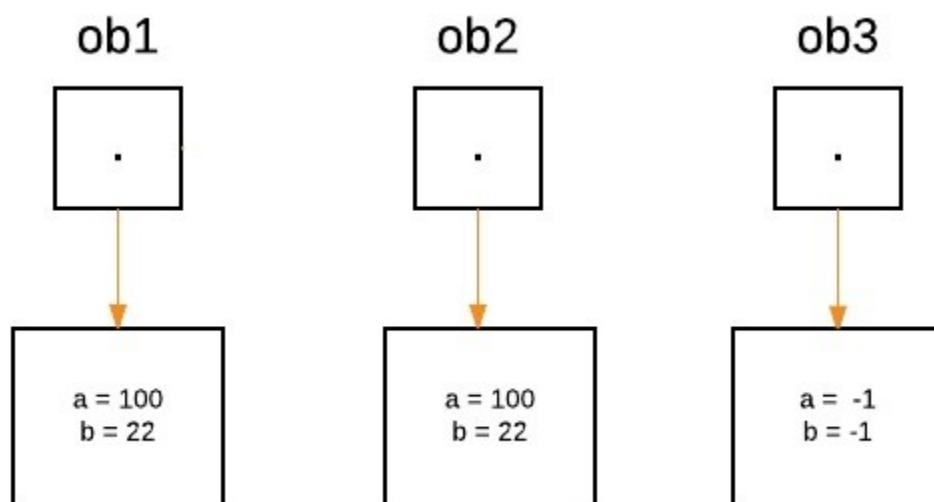
Although Java is strictly passed by value, the precise effect differs between whether a primitive type or a reference type is passed. When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.

Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect the object used as an argument.

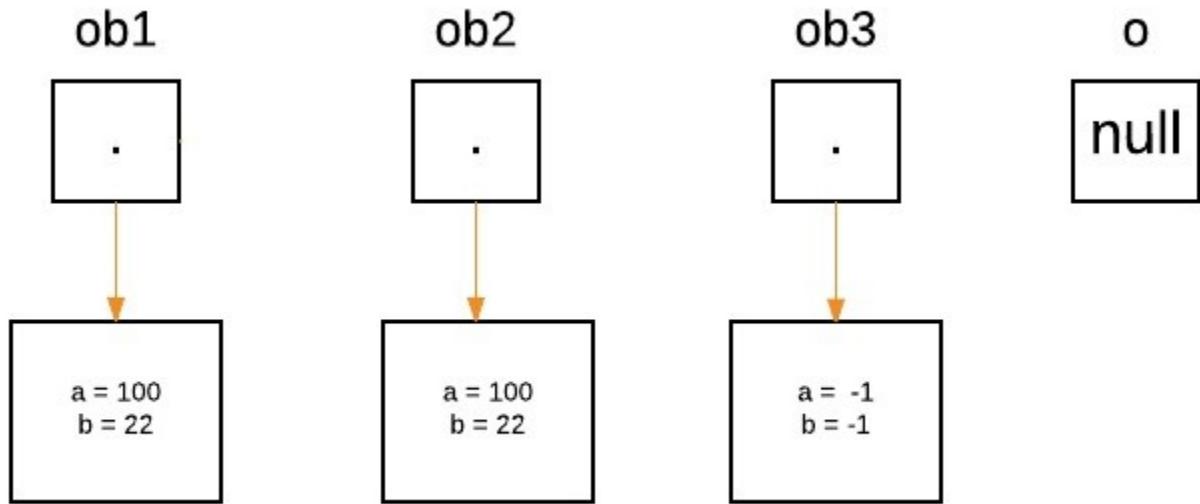
Illustration: Let us suppose three objects 'ob1' , 'ob2' and 'ob3' are created:

```
ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
```



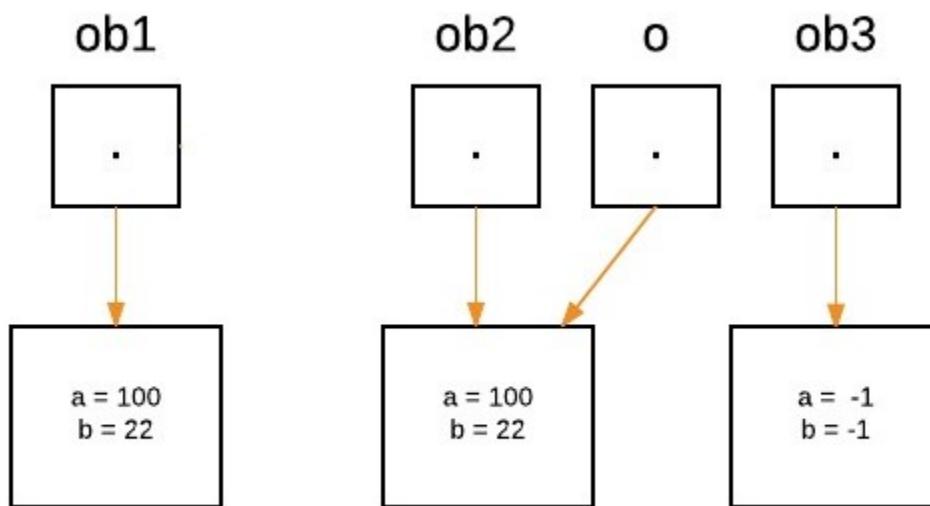
From the method side, a reference of type Foo with a name a is declared and it's initially assigned to null.

```
boolean equalTo(ObjectPassDemo o);
```



As we call the method equalTo, the reference 'o' will be assigned to the object which is passed as an argument, i.e. 'o' will refer to 'ob2' as the following statement execute.

```
System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
```

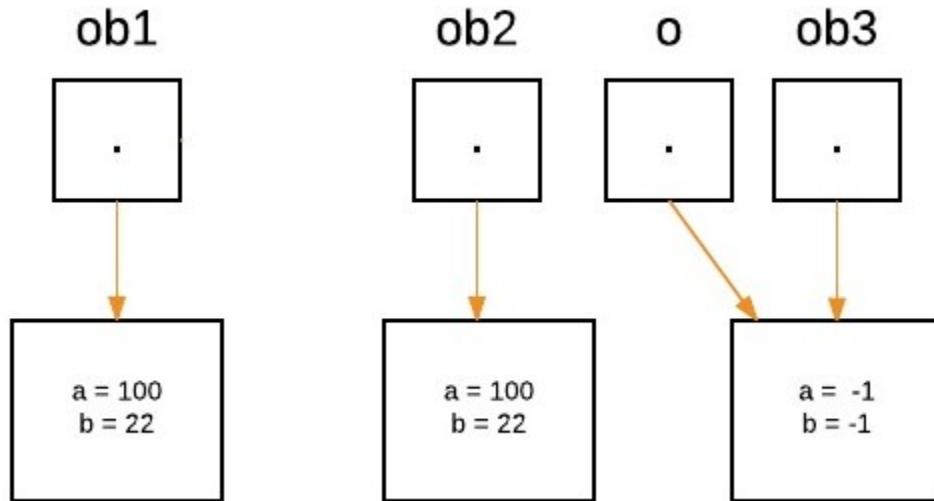


Now as we can see, equalTo method is called on 'ob1', and 'o' is referring to 'ob2'. Since values of 'a' and 'b' are same for both the references, so if(condition) is true, so boolean true will be return.

```
if(o.a == a && o.b == b)
```

Again 'o' will reassign to 'ob3' as the following statement execute.

```
System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
```



- Now as we can see, the equalTo method is called on 'ob1', and 'o' is referring to 'ob3'. Since values of 'a' and 'b' are not the same for both the references, so if(condition) is false, so else block will execute, and false will be returned.

In Java we can pass objects to methods as one can perceive from the below program as follows:

Example:

```
// Java Program to Demonstrate Objects Passing to Methods.

// Class
// Helper class
class ObjectPassDemo {
    int a, b;

    // Constructor
    ObjectPassDemo(int i, int j)
    {
        a = i;
        b = j;
    }

    // Method
```

```

boolean equalTo(ObjectPassDemo o)
{
    // Returns true if o is equal to the invoking
    // object notice an object is passed as an
    // argument to method
    return (o.a == a && o.b == b);
}

// Main class
public class GFG {
    // Main driver method
    public static void main(String args[])
    {
        // Creating object of above class inside main()
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);

        // Checking whether object are equal as custom
        // values
        // above passed and printing corresponding boolean
        // value
        System.out.println("ob1 == ob2: "
                           + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: "
                           + ob1.equalTo(ob3));
    }
}

```

Output

```

ob1 == ob2: true
ob1 == ob3: false

```

Defining a constructor that takes an object of its class as a parameter

One of the most common uses of object parameters involves constructors. Frequently, in practice, there is a need to construct a new object so that it is initially the same as some existing object. To do this, either we can use [Object.clone\(\)](#) method or define a constructor that takes an object of its class as a parameter.

Example

```
// Java program to Demonstrate One Object to
// Initialize Another

// Class 1
class Box {
    double width, height, depth;

    // Notice this constructor. It takes an
    // object of type Box. This constructor use
    // one object to initialize another
    Box(Box ob)
    {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions
    // specified
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() { return width * height * depth; }
}

// Main class
public class GFG {
    // Main driver method
    public static void main(String args[])
    {
        // Creating a box with all dimensions specified
        Box mybox = new Box(10, 20, 15);

        // Creating a copy of mybox
        Box myclone = new Box(mybox);

        double vol;

        // Get volume of mybox
```

```

        vol = mybox.volume();
        System.out.println("Volume of mybox is " + vol);

        // Get volume of myclone
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
    }
}

```

Output

```

Volume of mybox is 3000.0
Volume of myclone is 3000.0

```

Returning Objects

In java, a method can return any type of data, including objects. For example, in the following program, the **incrByTen()** method returns an object in which the value of an (an integer variable) is ten greater than it is in the invoking object.

Example

```

// Java Program to Demonstrate Returning of Objects

// Class 1
class ObjectReturnDemo {
    int a;

    // Constructor
    ObjectReturnDemo(int i) { a = i; }

    // Method returns an object
    ObjectReturnDemo incrByTen()
    {
        ObjectReturnDemo temp
            = new ObjectReturnDemo(a + 10);
        return temp;
    }
}

// Class 2

```

```
// Main class
public class GFG {

    // Main driver method
    public static void main(String args[])
    {

        // Creating object of class1 inside main() method
        ObjectReturnDemo ob1 = new ObjectReturnDemo(2);
        ObjectReturnDemo ob2;

        ob2 = ob1.incrByTen();

        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
    }
}
```

Output

```
ob1.a: 2
ob2.a: 12
```

Note: When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does. That's why we said that java is strictly_pass-by-value.

Constructor

- constructor name must match class name.
and cannot have a return type.
- constructor is called when the object is created.
- one kind of method, when an ~~class~~^{object} is created with a constructor is automatically created. When it's created by the user, we are able to set initial values for object attributes.
 - In Java, a constructor is a block of code similar to method.
 - It is called when an instance of class is created.
 - At the time of calling constructors, memory for the object is allocated in the memory.
 - Every time an object is created using the new() keyword, at least one constructor is called.

Definition

Syntax:

class name() { constructor body;

{} body starts from here
{} body ends before next method

{} body ends before next method

ex: class Bike 1 {

constructor [Bike 1() {
System.out.println("Bike is created");
}

public class void main (String args[]){

calls [Bike 1 b = new Bike(); } }

Output: Bike is created.

(Definition)

Rule-1: Constructor must have a name
the same as the class name.

Rule-2: A constructor must have no explicit return type. (No Return type)

Rule-3: It cannot be abstract, static, final.
(No access modifier)

Types: ① Parameterised (accepts parameters passed as arguments)
② Non parameterised (default)

① Default constructor

The default constructor is used to provide the default value like 0 or null → for string
for int

ex: class Student {
 int Id;
 String Name;
 void display() {
 System.out.println("Id " + "Name");
 }
}

```

public static void main(String args[])
{
    Student3 s1 = new Student3();
    Student3 s2 = new Student3();
    s1.display();
    s2.display();
}

```

Output:

| | | |
|---------|------------|-------------------------------------|
| 0 | NULL | because of constructor method |
| 0 | NULL | |
| ↑ | ↑ | |
| for int | for String | |

② Parameterised constructor :

```

class Student3{ int id;
               String name;
               }
Student3( int i, String n){ id = i;
                           name = n; }

void display(){
    System.out.println(id + " " + name);
}

```

```

public static void main(String args[])
{
    Student3 s1 = new Student3(11, "Kabir");
}

```

~~Student 3~~ S2 = new Student 3(112, "Asif");

```
s1.display();
s2.display(); }
```

Output:

111 Kabin

112 Asif

Q.

| | |
|-----------------------------------|------------------------------|
| Student 5 | int id, int age, string name |
| Student 5(int i, string n, int a) | |
| void displayInformation() | |

Show the output: 111 ASIF 18

112 KABIR 19

Code: class Student5{ int id; int age; String name;

alt: class Student5{ int id; int age; String name;

System.out.print(" "); Student5(int i, String n, int a){
 id = i; name = n; age = a; }
 void displayInformation(){ System.out.println(id + " " + name + " " + age); }}

public static void main(String args[]){

Student5 S1 = new Student5(111, "ASIF", 18);

Student5 S2 = new Student5(112, "KABIR", 19);

S1.displayInformation(); S2.displayInformation(); }}

can necessary
then

15.09.24

Constructor overloading

this keyword: (प्र०)

Q. Employee

```
Employee() { name = "Unknown";  
           id = 0; }
```

```
Employee(String empName; int empId);
```

Output:

Name = Unknown

Id = 0

Name = Ank

Id = 111

```
class Employee { String name;  
                int id; }
```

```
Employee() { name = "Unknown"; id = 0; }
```

```
Employee(String empName;  
        int empId) { name = empName;  
                    id = empId; }
```

A.S. 20.23

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

~~Employee emp1 = new Employee();~~

~~System.out.println("Name = " + name + " Id = " + id);~~

~~Employee emp2 = new Employee("Atik", 11);~~

~~System.out.println("Name = " + name + " Id = " + id);~~

{

check whether emp¹, emp²
or emp works
(new ob or same
overwritten)

- for calling different constructors, ~~new~~ different objects are needed. Can't call 2 different constructors with a single/same object.

22.09.24

[C] → next Sunday 29.09.24

slide 1, 2 + class object (slide)
(control)

[PDF] → 3. Variable types

Arithmatic operators : % = modulo

Increment, decrement operators

Prefix and post fix

Q. $x = 10;$

$x += x++ + 7 + x + x;$

$x = x + \cancel{(x++ + 7 + x + x)}$; increments but
not used

$= 10 + (10 + 12 + 12); = 44.$

(inline operation issue in C complier)

A. `int a = 5;`

`int b = 7;`

`int c = a++ + b;`

$$= 5 + 7 = 12$$

`d = ++a + b = 6 + 7`

Q. `int m = 1, n = 2;`

`int o = m++ + n + ++m;`

$$= 6$$

Instance of operators: used for object reference

variables only. Returns true or false.

Bitwise and Short hand operators: shortens (?)

$\boxed{\&, |, ^}$

$\&&, ||, !$

calculates in Binary

HW → switch-case related problems

→ same concept for constructor overloading

23.09.24

Method overloading / Parameterised overloading

person (int age, int Id, String Name)

person (int age, int Id)

person (float age, int Id)

person (float age) = person (age, 0)

person (String name) = person (0, 0, name)

person (String name, int Id) = person (name, 0, Id)

person (String name, float age) = person (name, age, 0)

person (String name, float age, int Id) = person (name, age, Id)

person (String name, float age, String address) = person (name, age, 0, address)

person (String name, float age, String address, int Id) = person (name, age, address, Id)

person (String name, float age, String address, int Id, float salary) = person (name, age, address, Id, salary)

person (String name, float age, String address, int Id, float salary, float bonus) = person (name, age, address, Id, salary, bonus)

person (String name, float age, String address, int Id, float salary, float bonus, float taxRate) = person (name, age, address, Id, salary, bonus, taxRate)

person (String name, float age, String address, int Id, float salary, float bonus, float taxRate, float deductionRate) = person (name, age, address, Id, salary, bonus, taxRate, deductionRate)

person (String name, float age, String address, int Id, float salary, float bonus, float taxRate, float deductionRate, float netSalary) = person (name, age, address, Id, salary, bonus, taxRate, deductionRate, netSalary)

ChatGPT

how does java switch case work

In Java, a `switch` statement is a control flow statement that allows you to execute one block of code from multiple possible options, based on the value of an expression. It's often used as an alternative to multiple `if-else` statements for better readability and efficiency when dealing with multiple conditions.

Here's how the `switch` statement works:

Basic Syntax:

```
java

switch (expression) {
    case value1:
        // Code to execute if expression equals value1
        break;
    case value2:
        // Code to execute if expression equals value2
        break;
    ...
    default:
        // Code to execute if none of the cases match
}
```

Key Points:

1. Expression Type:

- Starting from Java 12, the expression can be integers, characters, enumerated types (`enum`), `String`, or certain wrapper types (`Integer`, `Character`, etc.).
- From Java 14, switch expressions allow returning values.

2. `case` Labels:

- Each `case` label specifies a value to compare against the `switch` expression.
- Labels must be unique and constant expressions.

3. `break` Statement:

- It prevents fall-through to subsequent `case` blocks by exiting the `switch` statement.
- If omitted, the program continues executing subsequent `case` blocks (fall-through behavior).

4. `default` Case:

- The `default` block is executed if no `case` matches the expression. It's optional but recommended.

Example 1: Simple `switch` Statement

```
java

int day = 2;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

Output:

```
mathematica
```

Tuesday

Example 2: Fall-through Behavior

```
java

int number = 2;
switch (number) {
    case 1:
        System.out.println("One");
    case 2:
        System.out.println("Two");
    case 3:
        System.out.println("Three");
    default:
        System.out.println("Default");
}
```

Output:

```
mathematica
```

```
Two
Three
Default
```

This happens because there are no `break` statements, so execution continues through the subsequent cases.

Example 3: Enhanced `switch` Expressions (Java 14+)

```
java

int day = 3;
String dayName = switch (day) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    default -> "Unknown Day";
};
System.out.println(dayName);
```

Output:

mathematica

Wednesday

Use Cases:

- Simplifying decision-making logic.
- Handling multiple possible values of a variable.
- Cleaner and more maintainable than a series of `if-else` conditions for specific scenarios.

 Copilot

Yes, in your `switch` statement, the value of `season` will be the same for cases 3, 4, and 5. For any of these cases, `season` will be assigned the value "Spring". This is because all three cases fall under the same block of code:

Java

```
case 3:  
case 4:  
case 5:  
    season = "Spring";  
    break;
```

AI-generated code. Review and use carefully. [More info on FAQ.](#)

So, if `month` is 3, 4, or 5, `season` will be "Spring". If you have any more questions or need further clarification, feel free to ask!

can multiple cases have the same statement like in the picture?

Yes, multiple `case` labels can share the same block of code in a Java `switch` statement, as shown in the image you provided. This is a common pattern and works because the code block that follows a `case` statement will execute until it encounters a `break` (or another control statement like `return`, `continue`, or `throw`).

Here's how it works:

1. If the `switch` expression matches any of the listed `case` values, the corresponding code block is executed.
 2. Once executed, the `break` statement exits the `switch` structure to prevent further code execution.
-

Example

```
java

int month = 4;
String season;

switch (month) {
    case 3:
    case 4:
    case 5:
        season = "Spring";
        break;
    case 6:
    case 7:
    case 8:
        season = "Summer";
        break;
    case 9:
    case 10:
    case 11:
        season = "Autumn";
        break;
    case 12:
    case 1:
    case 2:
        season = "Winter";
        break;
    default:
        season = "Invalid month";
}

System.out.println("The season is: " + season);
```

Output:

If `month = 4`, the output will be:

```
csharp
```

```
The season is: Spring
```

Benefits of Grouping Cases:

- **Code Reuse:** Eliminates repetition when multiple cases share identical logic.
- **Improved Readability:** Groups related cases logically, making the code more readable.
- **Efficient Logic Design:** Handles multiple conditions without unnecessary duplication.

Using a `break` ensures that the program doesn't "fall through" to subsequent `case` blocks unintentionally.

Loops

① while loop:

```
while (condition){ System.out.println(); }
```

② do/while loop:

```
do{ } while (condition);
```

③ for loop:

```
for(statement 1; statement 2; statement 3){ }
```

↓
condition

④ for-each loop: used exclusively to loop through elements in an array.

```
for(type variableName : arrayName){ }
```

Ex: `String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};`

```
for(String i : cars){ System.out.println(i); }
```

Array: → can also - String cars[];

```
String[] cars;
String[] cars = {"Volvo", "BMW", "Ford",
                 "Mezda"};
```

```
System.out.println(cars[0]);
```

```
To print specific array elements  
for(int i=0; i < cars.length; i++)  
    System.out.println(cars[i]);
```

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

```
f{  
    // Loop through an array  
    for (var i = 0; i < arr.length; i++) {  
        console.log(arr[i]);  
    }  
}
```

```
for(String i : cars){
```

```
for(String l : arr){  
    System.out.println(l);
```

System.out.println

// using for-each loop beginning pool loop-1st ①

2D Array

2D Array:
int [][] myNumbers = {{1, 2, 3, 4},
{5, 6, 7}}

int[] myArray = {5, 6, 7};

$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}} dx = \pi$ [Integrate]

{ (W)alting (Learn: 1 part) and
two-measure

- Array Declaration and Implementation:

Arrays are fundamental structures in Java that allow us to store multiple values of the same type in a single variable. They are useful for managing collections of data efficiently. Arrays in Java work differently than they do in C/C++. This article covers the basics and in-depth explanations with examples of array declaration, creation, and manipulation in Java.

i. Declaring an Array

```
// declaring array  
int intArray[];
```

The general form of array declaration is

Method 1:

```
type var-name[];
```

```
// allocating memory to array  
intArray = new int[20];
```

Method 2:

```
type[] var-name;
```

```
// combining both statements in one  
int[] intArray = new int[20];
```

3. Access an Element of an Array

We can access array elements using their index, which starts from 0:

```
// Setting the first element of the array  
numbers[0] = 10;
```

```
// Accessing the first element  
int firstElement = numbers[0];
```

Implementation:

Java

```
1 // Java program to illustrate creating an array
2 // of integers, puts some values in the array,
3 // and prints each value to standard output.
4
5 class GFG {
6     public static void main(String[] args)
7     {
8         // declares an Array of integers.
9         int[] arr;
10
11         // allocating memory for 5 integers.
12         arr = new int[5];
13
14         // initialize the elements of the array
15         // first to last(fifth) element
16         arr[0] = 10;
17         arr[1] = 20;
18         arr[2] = 30;
19         arr[3] = 40;
20         arr[4] = 50;
21
22         // accessing the elements of the specified
23         // array
24         for (int i = 0; i < arr.length; i++)
25             System.out.println("Element at index "
26                           + i + " : " + arr[i]);
27     }
}
```

Output

```
Element at index 0 : 10
Element at index 1 : 20
Element at index 2 : 30
Element at index 3 : 40
Element at index 4 : 50
```

- Arrays of Objects in Java:

Arrays of Objects in Java

An array of objects is created like an array of primitive-type data items in the following way.

Syntax:

Method 1:

`ObjectType[] arrName;`

Method 2:

`ObjectType arrName[];`

Example of Arrays of Objects

Example 1: Here we are taking a student class and creating an array of Student with five Student objects stored in the array. The Student objects have to be instantiated using the constructor of the Student class, and their references should be assigned to the array elements.

Java

```
1 // Java program to illustrate creating
2 // an array of objects
3
4 class Student {
5     public int roll_no;
```

```

6     public String name;
7
8     Student(int roll_no, String name){
9         this.roll_no = roll_no;
10        this.name = name;
11    }
12}
13
14 public class Main {
15     public static void main(String[] args){
16
17         // declares an Array of Student
18         Student[] arr;
19
20         // allocating memory for 5 objects of type
21         Student.
22         arr = new Student[5];
23
24         // initialize the elements of the array
25         arr[0] = new Student(1, "aman");
26         arr[1] = new Student(2, "vaibhav");
27         arr[2] = new Student(3, "shikar");
28         arr[3] = new Student(4, "dharmesh");
29         arr[4] = new Student(5, "mohit");
30
31         // accessing the elements of the specified
32         array
33         for (int i = 0; i < arr.length; i++)
34             System.out.println("Element at " + i + " :
35             { "
36             + arr[i].roll_no + " "
37             + arr[i].name+ " }");
38         }
39     }

```

Output

```

Element at 0 : { 1 aman }
Element at 1 : { 2 vaibhav }
Element at 2 : { 3 shikar }
Element at 3 : { 4 dharmesh }
Element at 4 : { 5 mohit }

```

Example 2: An array of objects is also created like

Java

```
1 // Java program to illustrate creating
2 // an array of objects
3
4 class Student{
5     public String name;
6
7     Student(String name){
8         this.name = name;
9     }
10
11     @Override
12     public String toString(){
13         return name;
14     }
15 }
16
17
18 public class Main{
19     public static void main (String[] args){
20
21         // declares an Array and initializing the
22         // elements of the array
23         Student[] myStudents = new Student[]{
24             new Student("Dharma"),new Student("sanvi"),
25             new Student("Rupa"),new Student("Ajay")
26         };
27
28         // accessing the elements of the specified
29         // array
30         for(Student m:myStudents){
31             System.out.println(m);
32         }
33     }
34 }
```

Output

```
Dharma
sanvi
Rupa
Ajay
```

- looping object creation:

Java

Copy code

```
class Tmp {}

public class Main {
    public static void main(String[]
args) {
        Tmp[] A = new Tmp[10]; // Declare and initialize the array

        for (int i = 0; i < 10; i++)
{
            A[i] = new Tmp(); // Create new Tmp objects for each index of the array
        }
    }
}
```

Explanation:

1. **Array Initialization:**

- `Tmp[] A = new Tmp[10];` creates an array of `Tmp` objects with 10 slots, all initially set to null.

2. **Object Instantiation:**

- Inside the loop, `A[i] = new Tmp();` assigns a new `Tmp` object to each index of the array.

3. **Accessing Array Elements:**

- After the loop, you can access or manipulate each `Tmp` object in the array `A`.

// using indexed array as reference variable for object

// creation

class Tmp {

Yelp.com

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

```
for (int i=0; i<10; i++) {
```

$A[i] = \text{new Tmp();}$ //create
tmp objects
for each index

~~↓~~ ↓
Tmp[] A=new Tmp[10]; //declare and initialize array

for (String A : a) {

System.out.println("a");

→ for-each loop

Java Array Methods:

- (i) compare(): compares 2 arrays lexicographically
- (ii) equals(): checks if 2 arrays are equal
- (iii) sort(): sorts array in ascending order
- (iv) fill(): fills an array with a specific value
- (v) length: property to return length of array
not length()

(i) Arrays.compare(array1, array2)

System.out.println(Arrays.compare(array1, array2));

↳ Returns: 0 if array1 = array2

↳ -1 if array1 < array2

↳ +1 if array1 > array2

(ii) System.out.println(Arrays.equals(array1, array2));

↳ Returns: true if the arrays are equal

↳ false if the arrays are not equal

(iii) Arrays.sort(array, [start, end]):

↳ optional. selects the index position of the first (inclusive) and last (exclusive) element to be sorted

IV value must be of the same data type
as the array, must specify start and
end position to avoid filling all
elements.

Arrays.fill(array, [start, end, value])
→ optional

V built-in Java property

array.length

String[] cars = {"Volvo", "BMW", "Ford",
"Mazda"};

System.out.println(cars.length);

08.09.24

Q.

| |
|----------------------------|
| Rectangle |
| int length, int width |
| void Insert (int l, int w) |
| void calculateArea() |

Show the Area of Rectangle where

$$\text{length} = 11$$

$$\text{width} = 5$$

Class Rectangle {

 int length;

 int width;

 void Insert (int l, int w) {

 length = l;

 width = w; }

 void calculateArea() {

 System.out.println("Area = " + length * width); }

Class Main {

 public static void main (String [] args) {

 Rectangle R1 = new Rectangle();

 R1.Insert (11, 5);

 R1.calculateArea(); } }

commonly
used

→ P. 20. 30

→ String[] args is used to write main method.

[Hw] → chapter 3, 4, 5 from book

Parameter: variables listed in the function

or method definition

Argument: values passed to the function
when it's called

{ in = return }

{ () in statements block }

{ () in statements block }
B72k

{ () in statements block }
from 2009

{ () in statements block }
adding

{ () in statements block }

{ { () in statements block } }

Type Casting! Narrowing and Widening



```
double d = 55.55;
```

```
long l = (long) d;
```

```
int i = (int) l;
```

Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size

```
byte -> short -> char -> int -> long -> float -> double
```

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type

```
double -> float -> long -> int -> char -> short -> byte
```

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class Main {
    public static void main(String[] args) {
        int myInt = 9;
        double myDouble = myInt; // Automatic casting: int to double

        System.out.println(myInt);      // Outputs 9
        System.out.println(myDouble);   // Outputs 9.0
    }
}
```

Narrowing casting must be done manually by placing the type in parentheses `()` in front of the value:

Example

```
public class Main {  
    public static void main(String[] args) {  
        double myDouble = 9.78d;  
        int myInt = (int) myDouble; // Manual casting: double to int  
  
        System.out.println(myDouble); // Outputs 9.78  
        System.out.println(myInt); // Outputs 9  
    }  
}
```

Data types are divided into two groups:

- Primitive data types - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`
- Non-primitive data types - such as `String`, `Arrays` and `Classes` (you will learn more about these in a later chapter)

Primitive Data Types

A primitive data type specifies the type of a variable and the kind of values it can hold.

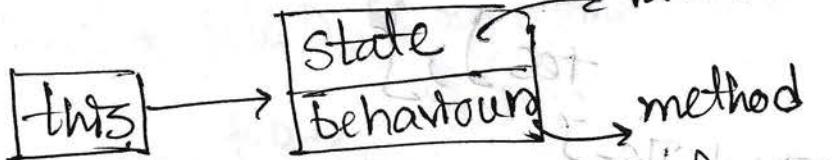
There are eight primitive data types in Java:

| Data Type | Description |
|----------------------|---|
| <code>byte</code> | Stores whole numbers from -128 to 127 |
| <code>short</code> | Stores whole numbers from -32,768 to 32,767 |
| <code>int</code> | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| <code>long</code> | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| <code>float</code> | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| <code>double</code> | Stores fractional numbers. Sufficient for storing 15 to 16 decimal digits |
| <code>boolean</code> | Stores true or false values |
| <code>char</code> | Stores a single character/letter or ASCII values |

| Type | Description | Default | Size | Example Literals | Range of values |
|----------------|-------------------------|---------|---------|---|---|
| boolean | true or false | false | 8 bits | true, false | true, false |
| byte | twos-complement integer | 0 | 8 bits | (none) | -128 to 127 |
| char | Unicode character | \u0000 | 16 bits | 'a', '\u0041', '\101', '\\", '\', '\n', '\beta' | characters representation of ASCII values 0 to 255 |
| short | twos-complement integer | 0 | 16 bits | (none) | -32,768 to 32,767 |
| int | twos-complement intger | 0 | 32 bits | -2,-1,0,1,2 | -2,147,483,648 to 2,147,483,647 |
| long | twos-complement integer | 0 | 64 bits | -2L,-1L,0L,1L,2L | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | IEEE 754 floating point | 0.0 | 32 bits | 1.23e100f , -1.23e-100f , .3f , .3.14F | upto 7 decimal digits |
| double | IEEE 754 floating point | 0.0 | 64 bits | 1.23456e300d , -123456e-300d , .1e1d | upto 16 decimal digits |

"this" keyword: eliminate confusion between class attributes and parameters with the same name

- (3) instance variable → virtual copy of a class variable
→ declared without "static" keyword
In Java, "this" is a reference variable that refers to the current object.



use: ① this can be used to refer current class instance → call to something

② this can be used to invoke current class method

③ this() can be used to invoke current class constructor / method

Syntax: → this(); → should call the constructor of the class in which it is written

- Class is defined with state and behaviour

If "this" is not used:

```
class Student { int rollno; String name;
    float fee;
```

```
Student (int rollno, string name,
        float fee) { roll no = roll no;
```

this.name = name;

this.fee = fee; }

```
void display () { System.out.println (
    roll no + " " + name +
    fee); }
```

```
class TestThis {
```

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

```
{ Student s1 = new Student (111, "Ankit", 5000 f),
  Student s2 = new Student (112, "Sumit", 6000 f); }
```

```
s1.display ();
```

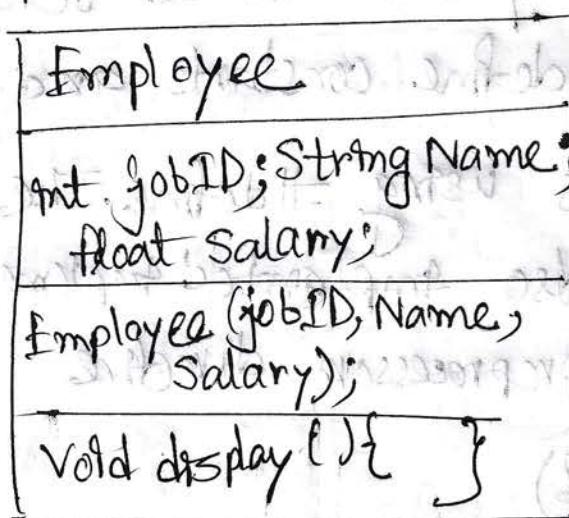
```
s2.display (); }
```

Output without this:

| | | |
|---|------|-----|
| 0 | Null | 0.0 |
| 0 | Null | 0.0 |

→ default value for float

Q1



Output:

905, Attk, 50000.0

class Employee { int jobID; String Name; float Salary; }

Employee (int jobID, String Name, float Salary) {

this.jobID = jobID;

this.^{Name} = Name;

this.Salary = Salary;

void display() { System.out.println("jobID+", " + Name + ", " + Salary); }

Class EmpThis { public static void main (String args[]) {

Employee E1 = new Employee(905, "Attk", 50000.0);

E1.display(); }}

② This to invoke current class Method

```
class A{ void m(){  
    System.out.println("Hello m");}  
    void n(){System.out.println("Hello n");}
```

1/m();

This.m();};}

these two are the same

```
.class TestThis{  
    public static void main(String args[]){  
        A a = new A();  
        a.n();}}}
```

‘this’ reference in Java

Last Updated : 08 Jan, 2024



In Java, ‘this’ is a reference variable that refers to the current object, or can be said “this” in Java is a keyword that refers to the current object instance. It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables. Using “this” reference can improve code readability and reduce naming conflicts.

Java this reference Example

In Java, this is a reference variable that refers to the current object on which the method or constructor is being invoked. It can be used to access instance variables and methods of the current object.

Below is the implementation of this reference:

```
// Java Program to implement
// Java this reference

// Driver Class
public class Person {

    // Fields Declared
    String name;
    int age;

    // Constructor
    Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Getter for name
    public String get_name() { return name; }

    // Setter for name
    public void change_name(String name)
    {
```

```
this.name = name;
}

// Method to Print the Details of
// the person
public void printDetails()
{
    System.out.println("Name: " + this.name);
    System.out.println("Age: " + this.age);
    System.out.println();
}

// main function
public static void main(String[] args)
{
    // Objects Declared
    Person first = new Person("ABC", 18);
    Person second = new Person("XYZ", 22);

    first.printDetails();
    second.printDetails();

    first.change_name("PQR");
    System.out.println("Name has been changed to: "
        + first.get_name());
}
}
```

Output

Name: ABC

Age: 18

Name: XYZ

Age: 22

Name has been changed to: PQR

Explanation

In the above code, we have defined a Person class with two private fields name and age. We have defined the Person class constructor to initialize these fields using this keyword. We have also defined getter and setter methods for these fields which use this keyword to refer to the current object instance.

In the printDetails() method, we have used this keyword to refer to the current object instance and print its name, age, and object reference.

In the Main class, we have created two Person objects and called the printDetails() method on each object. The output shows the name, age, and object reference of each object instance.

Methods to use ‘this’ in Java

Following are the ways to use the ‘this’ keyword in Java mentioned below:

- Using the ‘this’ keyword to refer to current class instance variables.
- Using this() to invoke the current class constructor
- Using ‘this’ keyword to return the current class instance
- Using ‘this’ keyword as the method parameter
- Using ‘this’ keyword to invoke the current class method
- Using ‘this’ keyword as an argument in the constructor call

1. Using ‘this’ keyword to refer to current class instance variables

```
// Java code for using 'this' keyword to
// refer current class instance variables
class Test {
    int a;
    int b;

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    void display()
    {
        // Displaying value of variables a and b
    }
}
```

```

        System.out.println("a = " + a + " b = " + b);
    }

public static void main(String[] args)
{
    Test object = new Test(10, 20);
    object.display();
}
}

```

Output

a = 10 b = 20

2. Using this() to invoke current class constructor

```

// Java code for using this() to
// invoke current class constructor
class Test {
    int a;
    int b;

    // Default constructor
    Test()
    {
        this(10, 20);
        System.out.println(
            "Inside default constructor \n");
    }

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println(
            "Inside parameterized constructor");
    }

public static void main(String[] args)

```

```
{  
    Test object = new Test();  
}  
}
```

Output

```
Inside parameterized constructor  
Inside default constructor
```

3. Using 'this' keyword to return the current class instance

```
// Java code for using 'this' keyword  
// to return the current class instance  
class Test {  
    int a;  
    int b;  
  
    // Default constructor  
    Test()  
    {  
        a = 10;  
        b = 20;  
    }  
  
    // Method that returns current class instance  
    Test get() { return this; }  
  
    // Displaying value of variables a and b  
    void display()  
    {  
        System.out.println("a = " + a + " b = " + b);  
    }  
  
    public static void main(String[] args)  
    {  
        Test object = new Test();  
        object.get().display();  
    }  
}
```

```
}
```

Output

```
a = 10  b = 20
```

4. Using 'this' keyword as a method parameter

```
// Java code for using 'this'  
// keyword as method parameter  
class Test {  
    int a;  
    int b;  
  
    // Default constructor  
    Test()  
    {  
        a = 10;  
        b = 20;  
    }  
  
    // Method that receives 'this' keyword as parameter  
    void display(Test obj)  
    {  
        System.out.println("a = " + obj.a  
                           + "  b = " + obj.b);  
    }  
  
    // Method that returns current class instance  
    void get() { display(this); }  
  
    // main function  
    public static void main(String[] args)  
    {  
        Test object = new Test();  
        object.get();  
    }  
}
```

Output

```
a = 10  b = 20
```

5. Using 'this' keyword to invoke the current class method

```
// Java code for using this to invoke current
// class method
class Test {

    void display()
    {
        // calling function show()
        this.show();

        System.out.println("Inside display function");
    }

    void show()
    {
        System.out.println("Inside show function");
    }

    public static void main(String args[])
    {
        Test t1 = new Test();
        t1.display();
    }
}
```

Output

```
Inside show function
Inside display function
```

6. Using 'this' keyword as an argument in the constructor call

```
// Java code for using this as an argument in constructor
// call
// Class with object of Class B as its data member
class A {
    B obj;

    // Parameterized constructor with object of B
    // as a parameter
    A(B obj)
    {
        this.obj = obj;

        // calling display method of class B
        obj.display();
    }
}

class B {
    int x = 5;

    // Default Constructor that create an object of A
    // with passing this as an argument in the
    // constructor
    B() { A obj = new A(this); }

    // method to show value of x
    void display()
    {
        System.out.println("Value of x in Class B : " + x);
    }
}

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

Output

Value of x in Class B : 5

Advantages of using ‘this’ reference

There are certain advantages of using ‘this’ reference in Java as mentioned below:

1. It helps to distinguish between instance variables and local variables with the same name.
2. It can be used to pass the current object as an argument to another method.
3. It can be used to return the current object from a method.
4. It can be used to invoke a constructor from another overloaded constructor in the same class.

Disadvantages of using ‘this’ reference

Although ‘this’ reference comes with many advantages there are certain disadvantages of also:

1. Overuse of this can make the code harder to read and understand.
2. Using this unnecessarily can add unnecessary overhead to the program.
3. Using this in a static context results in a compile-time error.
4. Overall, this keyword is a useful tool for working with objects in Java, but it should be used judiciously and only when necessary.

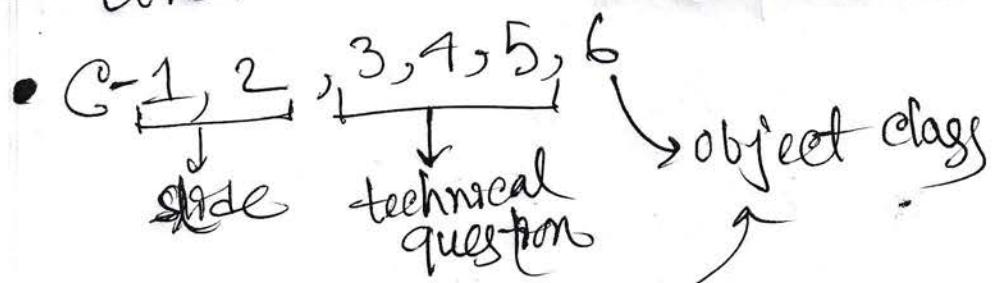
This article is contributed by **Mehak Narang** and **Amit Kumar**.

Mid discuss :

increment decrement - 105

type casting - 87

conditional statement → switch-case - 90



- Example from book

Final Syllabus

Super Keyword

① Super use in instance variable

② Super use in instance method

③ super() use in constructors (parent class)

`new Superclassname();`

`obj1.superclassname();`

calling obj1's class
constructor

`super();` calls the constructor

`classname() { }`

(doesn't include any resemblance with
the class)

`if (this == null) { }
(super.b)`

`(super.b)`

Super Keyword in Java

Last Updated : 15 Dec, 2023



The **super keyword in Java** is a reference variable that is used to refer to parent class when we're working with objects. You need to know the basics of [Inheritance](#) and [Polymorphism](#) to understand the Java super keyword.

The Keyword “super” came into the picture with the concept of Inheritance. In this article, we gonna covers all about super in Java including definitions, examples, Uses, Syntax, and more.

Table of Content

- [Characteristics of Super Keyword in Java](#)
- [Use of super keyword in Java](#)
- [1. Use of super with Variables](#)
- [2. Use of super with Methods](#)
- [3. Use of super with constructors](#)
- [Advantages of Using Java Super Keyword](#)

Characteristics of Super Keyword in Java

In Java, super keyword is used to refer to the parent class of a subclass. Here are some of its key characteristics:

- **super is used to call a superclass constructor:** When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.
- **super is used to call a superclass method:** A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class's implementation of the method in addition to its own.
- **super is used to access a superclass field:** A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class's version of a field.
- **super must be the first statement in a constructor:** When calling a superclass constructor, the super() statement must be the first statement in the constructor

of the subclass.

- **super cannot be used in a static context:** The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.
- **super is not required to call a superclass method:** While it is possible to use the super keyword to call a method in the parent class, it is not required. If a method is not overridden in the subclass, then calling it without the super keyword will invoke the parent class's implementation.

*Overall, the **super keyword** is a powerful tool for subclassing in Java, allowing subclasses to inherit and build upon the functionality of their parent classes.*

Use of super keyword in Java

It is majorly used in the following contexts as mentioned below:

- **Use of super with Variables**
- **Use of super with Methods**
- **Use of super with Constructors**

1. Use of super with Variables

This scenario occurs when a derived class and base class have the same data members. In that case, there is a possibility of ambiguity r the [JVM](#).

We can understand it more clearly using the following example:

Example

```
// super keyword in java example

// Base class vehicle
class Vehicle {
    int maxSpeed = 120;
}

// sub class Car extending vehicle
class Car extends Vehicle {
    int maxSpeed = 180;
```

```

void display()
{
    // print maxSpeed of base class (vehicle)
    System.out.println("Maximum Speed: "
                        + super.maxSpeed);
}
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}

```

Output

Maximum Speed: 120

In the above example, both the base class and subclass have a member maxSpeed. We could access the maxSpeed of the base class in subclass using super keyword.

2. Use of super with Methods

This is used when we want to call the parent class method. So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

This code snippet helps to understand the said usage of the super keyword.

Example

```

// super keyword in java example

// superclass Person
class Person {
    void message()
    {

```

```

        System.out.println("This is person class\n");
    }
}

// Subclass Student
class Student extends Person {
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is
    // only in Student class
    void display()
    {
        // will invoke or call current
        // class message() method
        message();

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

// Driver Program
class Test {
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}

```

Output

This is student class
 This is person class

In the above example, we have seen that if we only call method **message()** then, the current class **message()** is invoked but with the use of the **super** keyword, **message()** of the superclass could also be invoked.

3. Use of super with constructors

The super [keyword](#) can also be used to access the parent class constructor. One more important thing is that 'super' can call both parametric as well as non-parametric constructors depending on the situation.

Following is the code snippet to explain the above concept:

Example 1

```
// Java Code to show use of
// super keyword with constructor

// superclass Person
class Person {
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

// subclass Student extending the Person class
class Student extends Person {
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output

```
Person class Constructor
Student class Constructor
```

In the above example, we have called the superclass constructor using the keyword ‘super’ via subclass constructor.

Example 2

```
class ParentClass {  
    public boolean isTrue() { return true; }  
}  
  
class ChildClass extends ParentClass {  
    public boolean isTrue()  
    {  
        // calls parent implementation of isTrue()  
        boolean parentResult = super.isTrue();  
        // negates the parent result  
        return !parentResult;  
    }  
}  
  
public class Main {  
    public static void main(String[] args)  
    {  
        ChildClass child = new ChildClass();  
        // calls child implementation  
        // of isTrue()  
        boolean result = child.isTrue();  
  
        // prints "false"  
        System.out.println(result);  
    }  
}
```

Output

```
false
```

Advantages of Using Java Super Keyword

The **super keyword in Java** provides many advantages in object-oriented programming are as follows:

- **Enables reuse of code:** Using the super keyword allows subclasses to inherit functionality from their parent classes, which promotes the reuse of code and reduces duplication.
- **Supports polymorphism:** Because subclasses can override methods and access fields from their parent classes using super, polymorphism is possible. This allows for more flexible and extensible code.
- **Provides access to parent class behaviour:** Subclasses can access and use methods and fields defined in their parent classes through the super keyword, which allows them to take advantage of existing behaviour without having to reimplement it.
- **Allows for customization of behaviour:** By overriding methods and using super to call the parent implementation, subclasses can customize and extend the behaviour of their parent classes.
- **Facilitates abstraction and encapsulation:** The use of super promotes encapsulation and abstraction by allowing subclasses to focus on their behaviour while relying on the parent class to handle lower-level details.

Overall, the super keyword is a key feature of [inheritance](#) and [polymorphism](#) in Java, and it provides several benefits for developers seeking to write reusable, extensible, and well-organized code.

Important Points to Remember While Using “Java Super Keyword”

Here are some Important points that you need to take care of during using super keywords in Java:

- **Call to super() must be the first statement in the Derived(Student) Class constructor** because if you think about it, it makes sense that the superclass has no knowledge of any subclass, so any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it needs to complete its execution first.
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the

superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. The object *does* have such a constructor, so if the Object is the only superclass, there is no problem.



- If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors is called, all the way back to the constructor of Object. This, in fact, is the case. It is called [constructor chaining](#).

FAQs – Java super Keyword

Q1. What is super () and super keyword in Java?

Super() is a Java keyword used to call a superclass constructor. Super accesses superclass members and maintains inheritance hierarchies.

Q2. Which is the super class of Java?

The **Object class aka super class** is at the top of the class hierarchy in Java's **java.lang package**. Every class, whether predefined or user-defined, is a subclass of the Object class.

Q3. Why is Super important in Java?

super is essential in Java as it facilitates the access, initialization, and management of relationships between superclasses and subclasses, thereby promoting code reusability.

(Shakil scribbled)

Java String Methods

[« Previous](#)[Next »](#)

All String Methods

The String class has a set of built-in methods that you can use on strings.

| Method | Description | Return Type |
|--|--|-------------|
| <u>charAt()</u> | Returns the character at the specified index (position) | char |
| <u>codePointAt()</u> | Returns the Unicode of the character at the specified index | int |
| <u>codePointBefore()</u> | Returns the Unicode of the character before the specified index | int |
| <u>codePointCount()</u> | Returns the number of Unicode values found in a string. | int |
| <u>compareTo()</u> | Compares two strings lexicographically | int |
| <u>compareToIgnoreCase()</u> | Compares two strings lexicographically, ignoring case differences | int |
| <u>concat()</u> | Appends a string to the end of another string | String |
| <u>contains()</u> | Checks whether a string contains a sequence of characters | boolean |
| <u>contentEquals()</u> | Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer | boolean |
| <u>copyValueOf()</u> | Returns a String that represents the characters of the character array | String |
| <u>endsWith()</u> | Checks whether a string ends with the specified character(s) | boolean |

| | | |
|---|---|---------|
| <u>equals()</u> | C.compares two strings. Returns true if the strings are equal, and false if not | boolean |
| <u>equalsIgnoreCase()</u> | C.compares two strings, ignoring case considerations | boolean |
| <u>format()</u> | C.returns a formatted string using the specified locale, format string, and arguments | String |
| <u>getBytes()</u> | C.converts a string into an array of bytes | byte[] |
| <u>getChars()</u> | C.copies characters from a string to an array of chars | void |
| <u>hashCode()</u> | C.returns the hash code of a string | int |
| <u>indexOf()</u> | C.returns the position of the first found occurrence of specified characters in a string | int |
| <u>intern()</u> | C.returns the canonical representation for the string object | String |
| <u>isEmpty()</u> | C.checks whether a string is empty or not | boolean |
| <u>join()</u> | C.joins one or more strings with a specified separator | String |
| <u>lastIndexOf()</u> | C.returns the position of the last found occurrence of specified characters in a string | int |
| <u>length()</u> | C.returns the length of a specified string | int |
| <u>matches()</u> | C.searches a string for a match against a regular expression, and returns the matches | boolean |
| <u>offsetByCodePoints()</u> | C.returns the index within this String that is offset from the given index by codePointOffset code points | int |
| <u>regionMatches()</u> | C.tests if two string regions are equal | boolean |
| <u>replace()</u> | C.searches a string for a specified value, and returns a new string where the specified values are replaced | String |
| <u>replaceAll()</u> | C.replaces each substring of this string that matches the given regular expression with the given replacement | String |

| | | |
|-----------------------|---|--------------|
| <u>replaceFirst()</u> | Replaces the first occurrence of a substring that matches the given regular expression with the given replacement | String |
| <u>split()</u> | Splits a string into an array of substrings | String[] |
| <u>startsWith()</u> | Checks whether a string starts with specified characters | boolean |
| <u>subSequence()</u> | Returns a new character sequence that is a subsequence of this sequence | CharSequence |
| <u>substring()</u> | Returns a new string which is the substring of a specified string | String |
| <u>toCharArray()</u> | Converts this string to a new character array | char[] |
| <u>toLowerCase()</u> | Converts a string to lower case letters | String |
| <u>toString()</u> | Returns the value of a String object | String |
| <u>toUpperCase()</u> | Converts a string to upper case letters | String |
| <u>trim()</u> | Removes whitespace from both ends of a string | String |
| <u>valueOf()</u> | Returns the string representation of the specified value | String |

what are java built in classes

Java includes a rich set of built-in classes provided by its standard library, commonly referred to as the **Java API**. These classes are organized into **packages** such as `java.lang`, `java.util`, `java.io`, and more. Below is an overview of some key packages and their commonly used classes:

1. `java.lang` Package

The `java.lang` package is automatically imported in every Java program. It contains fundamental classes.

- `Object` : The root of the class hierarchy; every class in Java inherits from `Object`.
 - `String` : Represents sequences of characters.
 - `StringBuilder` / `StringBuffer` : For mutable strings.
 - `Math` : Provides mathematical operations and constants.
 - `System` : Contains system utilities like input/output and garbage collection.
 - `Thread` : For multithreading.
 - `Runtime` : Deals with the Java runtime environment.
-

2. `java.util` Package

Contains utility classes for data structures, collections, and date/time manipulation.

- **Collection Framework:**
 - `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`, `PriorityQueue`
- **Date/Time API:**
 - `Date`, `Calendar`, `LocalDate`, `LocalTime`, `LocalDateTime`, `ZoneId`
- **Other Utilities:**

- `Scanner` : For input parsing.
 - `Random` : For generating random numbers.
 - `Arrays` : For array-related operations.
 - `Collections` : For collection-related utilities.
-

3. `java.io` Package

Contains classes for input/output operations.

- **File Operations:**
 - `File` : Represents file and directory paths.
 - `FileReader` / `FileWriter` : For reading/writing text files.
 - **Stream API:**
 - `InputStream` , `OutputStream` : For byte-stream operations.
 - `BufferedReader` , `BufferedWriter` : For efficient reading/writing.
 - `ObjectInputStream` , `ObjectOutputStream` : For object serialization.
-

4. `java.nio` Package

Used for non-blocking I/O and advanced file operations.

- `Paths` and `Files` : Modern ways to work with file systems.
 - `ByteBuffer` , `CharBuffer` : For buffer management.
-

5. `java.net` Package

Contains classes for networking.

- `URL` : Represents a Uniform Resource Locator.

- `HttpURLConnection` : For HTTP connections.
 - `Socket` , `ServerSocket` : For socket programming.
-

6. `java.sql` Package

Classes for database interaction.

- `Connection` : Represents a connection to a database.
 - `Statement` , `PreparedStatement` : Execute SQL queries.
 - `ResultSet` : Retrieve and manipulate query results.
-

7. `java.time` Package (Java 8 and later)

A modern date/time API.

- `LocalDate` , `LocalTime` , `LocalDateTime` : Immutable date-time objects.
 - `Instant` , `Duration` , `Period` : Represent time intervals.
 - `DateTimeFormatter` : For formatting dates and times.
-

8. `java.util.concurrent` Package

For concurrent programming and multi-threading.

- `Executor` , `ExecutorService` : Manage threads.
 - `Future` : Handle asynchronous results.
 - `ConcurrentHashMap` , `CopyOnWriteArrayList` : Thread-safe collections.
-

9. `javax` Packages

Various packages beginning with `javax` include:

- `javax.swing` : For creating GUI applications.
 - `javax.servlet` : For web applications.
 - `javax.xml` : For XML parsing and processing.
-

Example Usage

```
java

import java.util.ArrayList;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();

        ArrayList<Integer> list = new ArrayList<>();
        list.add(num);
        System.out.println("List: " + list);
    }
}
```

Java's built-in classes help you handle tasks ranging from basic string manipulation to advanced concurrency and networking. You can refer to the [Java Documentation](#) for detailed information on these classes.

Java Arrays Class

[« Previous](#)[Next »](#)

The Java `Arrays` class (found in `java.util`), has methods that allow you to manipulate arrays.

Arrays Methods

A list of popular methods of the Arrays Class can be found in the table below:

| Method | Description |
|---------------------------|---|
| <code>compare()</code> | Compares two arrays |
| <code>copyOf()</code> | Creates a copy of an array with a new length |
| <code>deepEquals()</code> | Compares two multidimensional arrays to check whether they are deeply equal to each other |
| <code>equals()</code> | Checks if two arrays are equal |
| <code>fill()</code> | Fills an array with a specified value |
| <code>mismatch()</code> | Returns the index position of the first mismatch/conflict between two arrays |
| <code>sort()</code> | Sorts an array in ascending order |

Properties

| Property | Description |
|---------------------|--------------------------------|
| <code>length</code> | Returns the length of an array |

The `length` property is a built-in Java property, and does not belong to the Arrays class.

Related Pages

[Java Arrays Tutorial](#)

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes

(Book lines align with Setu ma'am)

30.09.24

Inheritance: "extends" keyword

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. The Parent class idea behind inheritance in Java is that you can create new classes that are built upon existing class. When you inherit from an existing class you can reuse methods and field of the parent class.

Inheritance represents the "IS-A"

relationship which is known as a parent-child relationship.

• Sub class / child class / Derived Class

• Super class / parent class / Base Class

Is-A Inheritance in Java

explain is-a relation of inheritance in java

The **is-a** relationship in Java refers to a concept in object-oriented programming where one class is a specialized version of another class. This relationship is established through **inheritance**.

Key Points:

1. Inheritance:

- When a class inherits another class using the `extends` keyword, it establishes an is-a relationship.
- The subclass (child class) inherits the properties and methods of the superclass (parent class).

2. Meaning of Is-A:

- A subclass can be considered as an instance of its superclass because it possesses the characteristics (properties and behaviors) of the superclass.
- This aligns with the idea of generalization and specialization:
 - The superclass represents a general category.
 - The subclass represents a more specific category.

3. Polymorphism:

- A subclass object can be referred to by a superclass reference due to the is-a relationship.
- For example:

```
java

class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

```

    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog(); // Polymorphism
        a.eat(); // Valid: Dog is-an Animal
        // a.bark(); // Not allowed with an Animal reference
    }
}

```

- Here, **Dog** is-an **Animal**, so you can assign a **Dog** object to an **Animal** reference.

4. Real-life Analogy:

- **Superclass:** Vehicle
- **Subclass:** Car, Truck, Motorcycle
- All cars, trucks, and motorcycles **are** vehicles, which means they inherit the characteristics of the Vehicle class but might also have their specific features.

Advantages of the Is-A Relationship:

- Promotes code reuse by inheriting **fields and methods** from the parent class.
- Supports polymorphism, enabling flexibility and extensibility in code design.
- Helps in implementing a hierarchical structure, making the codebase easier to understand and manage.

Limitations:

- Should not be confused with a **Has-A relationship (composition)**, where one class has a reference to another.
- Overusing inheritance can lead to tightly coupled classes, which can make maintenance harder. Favor composition over inheritance when appropriate.

give me a class diagram of it

A class diagram for demonstrating the **is-a** relationship in Java inheritance would show:

1. A **parent class (Animal)**.
2. Two **child classes (Dog and Cat)** that extend the **Animal** class.

Here's the diagram represented textually, as I cannot directly draw here:

lua

```
+-----+  
|      Animal      | <-- Superclass  
+-----+  
| - name          |  
| - age           |  
+-----+  
| + eat()         |  
+-----+  
          ^  
          |  
+-----+-----+  
|          |  
+-----+-----+  
|      Dog       |      Cat      | <-- Subclasses  
+-----+-----+  
| - breed        |      | - color     |  
+-----+-----+  
| + bark()       |      | + meow()    |  
+-----+-----+
```

Description of Diagram Elements:

1. Superclass (Animal):

- Attributes: `name`, `age` (fields shared by all animals).
- Method: `eat()` (shared behavior).

2. Subclass (Dog):

- Attribute: `breed` (specific to Dog).
- Method: `bark()` (unique behavior for dogs).

3. Subclass (Cat):

- Attribute: `color` (specific to Cat).
- Method: `meow()` (unique behavior for cats).

Code Representation:

Here's how the diagram translates into Java code:

```
java

// Superclass
class Animal {
    String name;
    int age;

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

// Subclass Dog
class Dog extends Animal {
    String breed;

    void bark() {
        System.out.println(name + " is barking.");
    }
}

// Subclass Cat
class Cat extends Animal {
    String color;

    void meow() {
        System.out.println(name + " is meowing.");
    }
}

// Main Program
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
```

```
dog.name = "Buddy";
dog.age = 5;
dog.breed = "Labrador";
dog.eat(); // Inherited method
dog.bark(); // Dog-specific method

Cat cat = new Cat();
cat.name = "Whiskers";
cat.age = 3;
cat.color = "White";
cat.eat(); // Inherited method
cat.meow(); // Cat-specific method
}

}
```

Let me know if you'd like a visual representation (e.g., generated as an image).



what is a Has-A relation in java?



A **Has-A** relationship in Java refers to **composition**, where one class contains a reference to another class as an attribute. This indicates a "part-of" or "owns" relationship, where one object "has" another as its property or dependency.

Key Features of Has-A Relationship

1. Uses Composition:

- A class contains an instance (or reference) of another class.
- This is implemented using instance variables.

2. Meaning of Has-A:

- "Class A has a Class B" implies that objects of Class A include (or depend on) objects of Class B as part of their structure or functionality.

3. Flexibility:

- Composition allows objects to be reused in multiple contexts.
- It promotes loose coupling, as the classes are independent but work together.

4. Comparison with Is-A:

- **Is-A** involves inheritance (``extends`` keyword), meaning one class is a specialized version of another.
- **Has-A** involves composition, meaning one class uses or owns another without inheriting from it.

Example of Has-A Relationship

Consider a `Car` that **has-a** `Engine`.

```
java

// Engine Class
class Engine {
    void start() {
        System.out.println("Engine starts.");
    }
}

// Car Class
class Car {
    // Car has-a Engine
    private Engine engine;

    // Constructor to initialize the engine
    Car(Engine engine) {
        this.engine = engine;
    }

    void drive() {
        engine.start(); // Using the Engine
        System.out.println("Car is driving.");
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        Engine engine = new Engine(); // Create an Engine
        Car car = new Car(engine); // Composition: Car has-a Engine
        car.drive();
    }
}
```

Output

csharp

```
Engine starts.
Car is driving.
```

Explanation:

1. The `Engine` **class** is an independent class with its own functionality.
2. The `Car` **class** contains a **reference to `Engine` as an instance variable**. This signifies that a car "has-a" engine.
3. Using the `Car` object calls methods of the `Engine` class, demonstrating composition.

Benefits of Has-A:

- **Reusability:** You can reuse the `Engine` class in different vehicles (e.g., bikes, trucks) without modifying its functionality.
- **Loose Coupling:** Changes to the `Engine` class won't affect the `Car` class much, as they are only connected through composition.
- **Flexibility:** It provides better control compared to inheritance, as objects can be dynamically associated during runtime.

Real-World Analogy

1. A **Person** has-a **Heart**.

- The **Person** is independent of the **Heart**, but the **Person** contains and depends on the **Heart** for functioning.

2. Code Example:

```
java

class Heart {
    void beat() {
        System.out.println("Heart is beating...");
    }
}

class Person {
    private Heart heart;

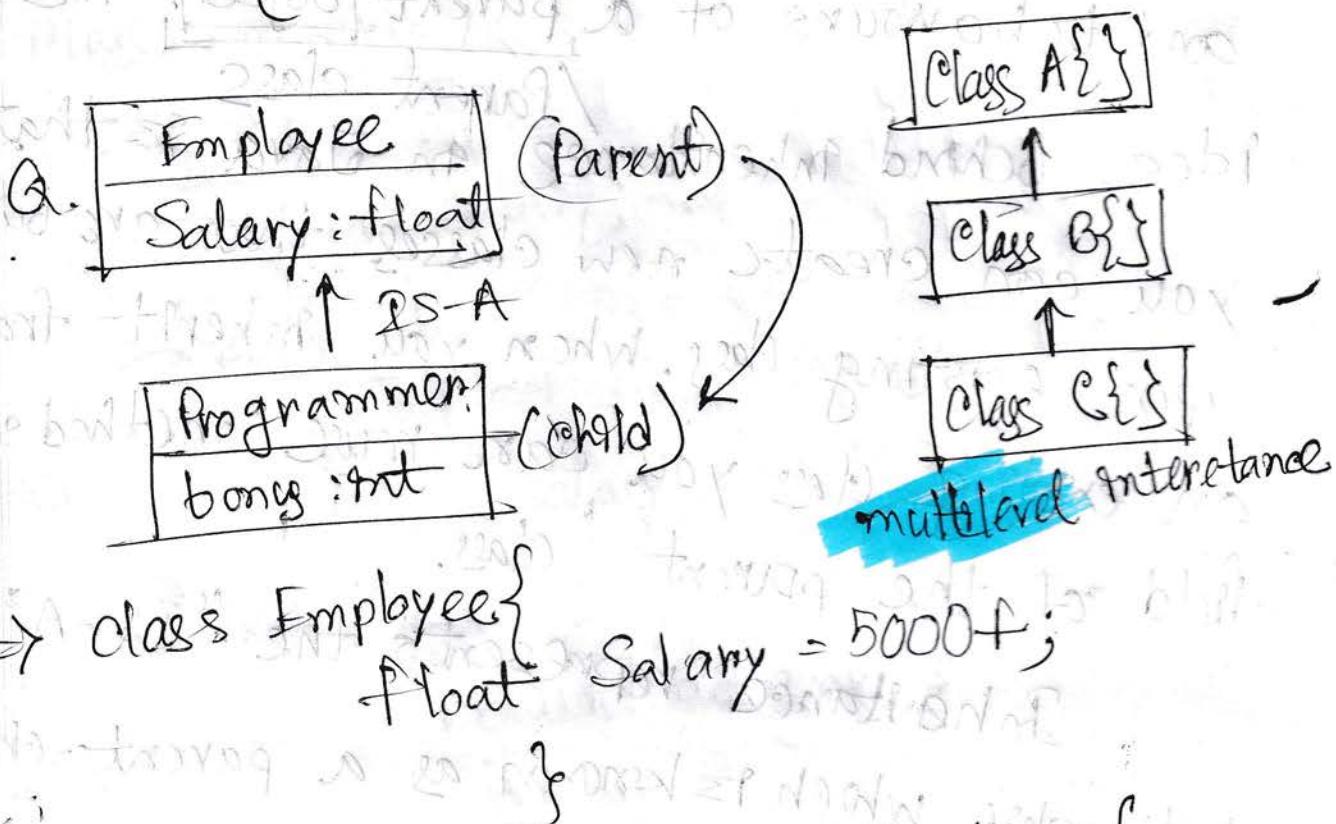
    Person(Heart heart) {
        this.heart = heart;
    }

    void live() {
        heart.beat();
        System.out.println("Person is alive.");
    }
}
```

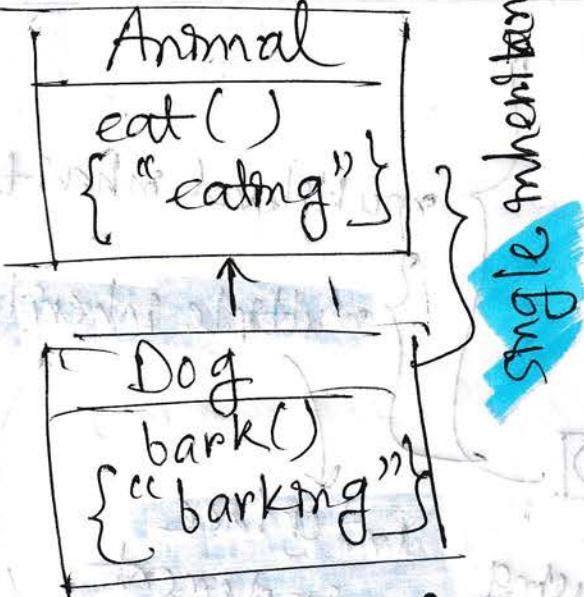
This is a practical example of Has-A, where **`Person`** has-a **`Heart`**.

Syntax: of Inheritance:

class Subclass_name extends Superclass_name
{ methods and fields }



Q.



single inheritance

single

Access modifier:

```

public class Animal {
}
  
```

```

class Animal {
}
  
```

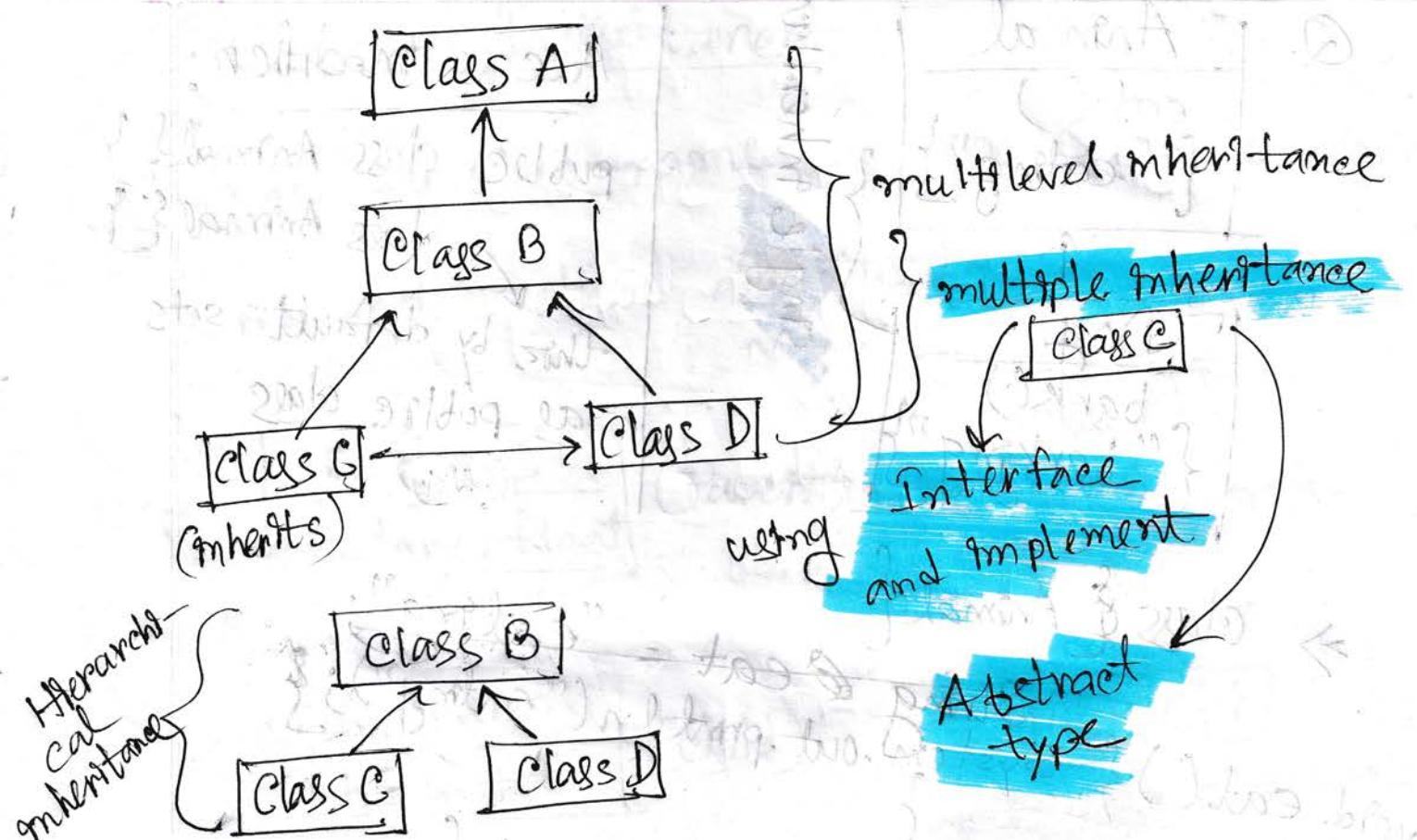
this by default sets
as public class

```

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

class Dog extends Animal {
    bark()
    {
        System.out.println("barking");
    }
}

public static void main(String args[])
{
    Dog b1 = new Dog();
    b1.eat();
    b1.bark();
}
  
```



Mod syllabus:-

Constructor overloading
(this & I)

21 October 2024

603 - (11.30-12.30) → Mid

Hierarchy:

Hierarchical Inheritance:

"When two or more classes inherits a single class, it is known as hierarchical inheritance"

[class Animal]

[class Cat]

[class Dog]

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

class Dog extends Animal{
 void bark()
 {System.out.println("barking...");}}

class Cat extends Animal{
 void meow()
 {System.out.println("meowing");}}

class Cat extends Dog { }

↳ creates hybrid inheritance which doesn't work without interface

class TestInheritance03 {

creates ambiguity as to which cat is called

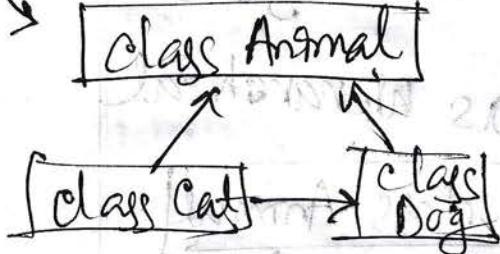
public static void main (static args[]).

{ cat c = new Cat();

c.meow();

c.eat();

c.bark(); } }



Output: meowing
eating

~~Method overriding (prioritizes child class method)~~

```
class Animal{  
    void eat() {  
        System.out.println("eating...");  
    }  
  
    class Dog extends Animal{  
        void eat() {  
            System.out.println("barking...");  
        }  
  
        public static void main(String args[]){  
            Dog d = new Dog();  
            d.eat();  
        }  
    }  
}
```

```
Dog d = new Dog(j);
d.eat(); } }
```

Output: barking (Default → child class)

```
• void display()
{   eat(); super.eat(); } }
```

d.display();

#Encapsulation

07.10.24

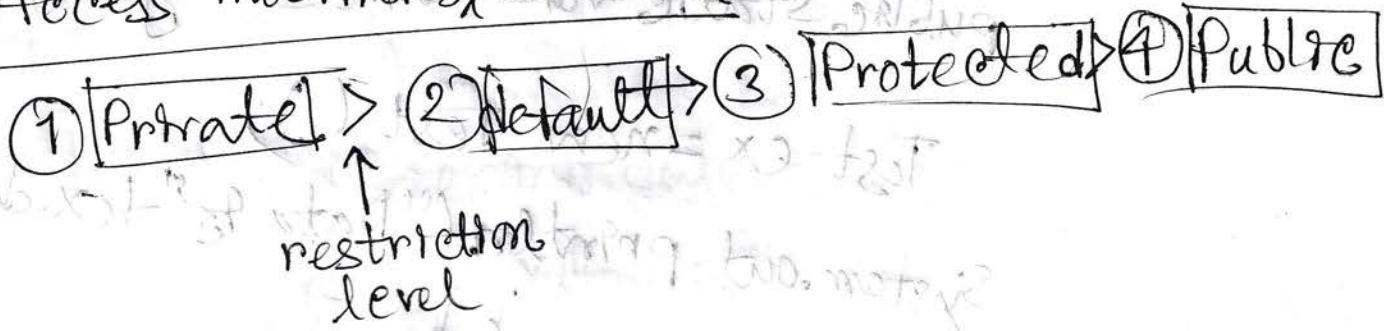
objects as array? → much like structure members

as arrays in C?

this keyword Java and

specifiers

Access modifiers in Java:



Syntax:

```
① public class Test{
    private int data = 40;
    public static void main(String args[]){
        Test ex = new Test();
        System.out.println("Data is "+ex.data);
    }
}
```

Output: Data is **40** since it's in the same class

code:

```
public class Test{  
    private int data = 40;}
```

```
class Test1{
```

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

```
        Test ex = new Test();
```

```
        System.out.println("Data is " + ex.data);  
    }  
}
```

Output: Complete Time errors.

→ Run time errors

Access Modifiers in Java

Last Updated : 15 Dec, 2024



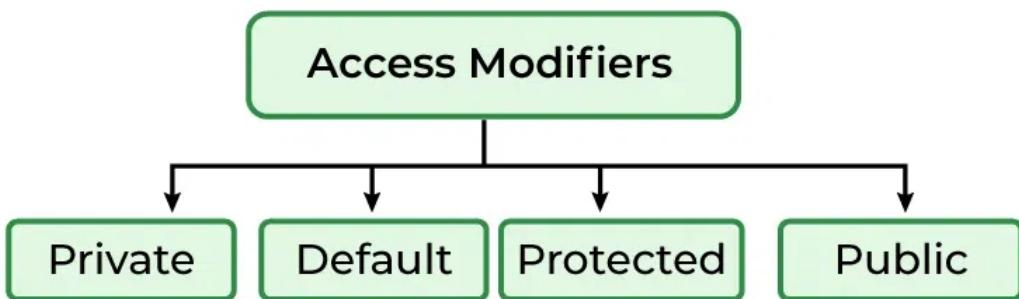
In Java, Access modifiers helps to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc. to the user depending upon the access modifier used with the element. In this article, let us learn about Java Access Modifiers, their types, and the uses of access modifiers.

Types of Access Modifiers

There are 4 types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

Access Modifiers in Java



1. Default Access Modifier

When no access modifier is specified for a class, method, or data member, it is said to be having the **default** access modifier by default. The default access modifiers are accessible **only within the same package**.

Example 1: Demonstrating Default Access Modifier Within the Same Package. In this example, we will create two packages and the classes in the packages will be

having the default access modifiers and we will try to access a class from one package from a class of the second package.

Java

```
1 // default access modifier
2 package p1;
3
4 // Class Geek is having
5 // Default access modifier
6 class Geek
7 {
8     void display()
9     {
10         System.out.println("Hello World!");
11     }
12 }
```

Example 2: Error when Accessing Default Modifier Class across Packages. In this example, the program will show the **compile-time error** when we try to access a default modifier class from a different package.

Java

```
1 // error while using class from different
2 // package with default modifier
3 package p2;
4 import p1.*;      // importing package p1
5
6 // This class is having
7 // default access modifier
8 class GeekNew {
9     public static void main(String args[]) {
10
11         // Accessing class Geek from package p1
12         Geek o = new Geek();
13
14         o.display();
```

```
15 }  
16 }
```

2. Private Access Modifier

The **private access modifier** is specified using the keyword **private**. The methods or data members declared as private are accessible **only within the class in which they are declared**.

- Any other class of the **same package** will not be able to access these members.
- **Top-level classes or interfaces** can not be declared as private because,
 - **private** means “**only visible within the enclosing class**“.
 - **protected** means “**only visible within the enclosing class and any subclasses**“.

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes.

Example: In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

Java

```
1 // error while using class from different package with  
2 // private access modifier  
3 package p1;  
4  
5 // Class A  
6 class A {  
7     private void display() {  
8         System.out.println("GeeksforGeeks");  
9     }  
10 }  
11  
12 // Class B  
13 class B {
```

```
14     public static void main(String args[]) {  
15         A obj = new A();  
16  
17         // Trying to access private method  
18         // of another class  
19         obj.display();  
20     }  
21 }
```

Explanation: The above code will show a compile-time error when trying to access a private method from class B, even within the same package.

3. Protected Access Modifier

The **protected access modifier** is specified using the keyword **protected**. The methods or data members declared as protected are **accessible within the same package or subclasses in different packages**.

Example 1: In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

Java

```
1 // protected access modifier  
2 package p1;  
3  
4 // Class A  
5 public class A {  
6     protected void display() {  
7         System.out.println("GeeksforGeeks");  
8     }  
9 }
```

So, it demonstrates that a protected method is accessible within the same package.

Example 2: In this example, we will create two packages, p1 and p2. Class A in p1 has a protected method display. Class B in p2 extends A and accesses the protected method through [inheritance](#) by creating an object of class B.

Java

```
1 // protected modifier
2 package p2;
3
4 // importing all classes
5 // in package p1
6 import p1.*;
7
8 // Class B is subclass of A
9 class B extends A {
10     public static void main(String args[]) {
11         B obj = new B();
12         obj.display();
13     }
14 }
```

Explanation: The above example demonstrates that a [protected method](#) is accessible in a subclass from a different package using inheritance.

4. Public Access Modifier

The **public access modifier** is specified using the keyword **public**.

- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

Example 1: Here, the code shows that a **public method** is **accessible within the same package**.

Java



```
1 // public modifier
2 package p1;
3
4 public class A {
5
6     public void display() {
7         System.out.println("GeeksforGeeks");
8     }
9 }
```

Example 2: Here, the example shows that a **public method** is *accessible across packages*.

Java



```
1 // public access modifier
2 package p2;
3
4 import p1.*;
5
6 class B {
7     public static void main(String args[]) {
8
9         A obj = new A();
10        obj.display();
11    }
12 }
```

Important Points:

- If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants.

Comparison Table of Access Modifiers in Java

| | Default | Private | Protected | Public |
|--------------------------------|---------|---------|-----------|--------|
| Same Class | Yes | Yes | Yes | Yes |
| Same Package Subclass | Yes | No | Yes | Yes |
| Same Package Non-Subclass | Yes | No | Yes | Yes |
| Different Package Subclass | No | No | Yes | Yes |
| Different Package Non-Subclass | No | No | No | Yes |

Algorithm to Use Access Modifier in Java

Here's a basic algorithm for using access modifiers in Java:

- **Define a class:** Create a class to represent the object you want to manage.
- **Define instance variables:** Inside the class, define variables for the data you want to manage.
- **Set an access modifier:**
 - Use **private** for variables *only accessible within the class*.
 - Use **protected** for variables *accessible within the class and its subclasses*.
 - Use **public** for variables *accessible from anywhere*.
- **Use getter and setter methods:** To access or modify variables, use **getter (accessor)** and **setter (mutator) methods**, even for public variables, to maintain encapsulation.

FAQs – Access Modifiers in Java

What are access modifiers in Java?

Access modifiers in Java are the keywords that are used for controlling the use of the methods, constructors, fields, and methods in a class.

What is the default access modifier in Java?

The default access modifier allows access within the same package only.

What are the 12 modifiers in Java?

12 Modifiers in Java are public, private, protected, default, static, final, synchronized, abstract, native, strictfp, transient, and volatile.

Can a private method be accessed outside its class?

No, a private method is only accessible within its own class.

What does the protected access modifier do?

The protected modifier allows access within the same package and by subclasses.

Can public variables be accessed from any class?

Yes, public variables are accessible from any class, regardless of the package.

* Q. class A

```
private int data = 40
private void msg(){
    "Hello java";}
```

write the main method
in Test class 1 and
print the following
output:

10
Hello java

Code: class A {

```
private int data = 40;
private void msg(){
```

```
System.out.println("Hello java");}
```

public class TestClass1

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

```
A obj = new A();
```

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

```
obj.msg();
```

```
}
```

Output: compile time error

→ getter-setter method

How to get access?

⇒ If we want to make this variable available to the outside world/publicly, we have to define the two public method;

① getter: which returns the value of the variable.

② setter: which sets the value of the variable.

Syntax:

code: class A { private int data;
public void set A (int data) {
this.data = data;
public int get A () {
return this.data; } }

convention

public class Test {

public static void main (String args[]) {
System.out.println ("Hello World"); }

A obj = new A();
obj.setA(12);
System.out.println("Data is "+
obj.getA());}}

Getter and Setter in Java

Last Updated : 22 Jun, 2023



In Java, Getter and Setter are methods used to protect your data and make your code more secure. Getter and Setter make the programmer convenient in setting and getting the value for a particular data type.

Getter in Java: Getter returns the value (accessors), it returns the value of data type int, String, double, float, etc. For the program's convenience, the getter starts with the word "get" followed by the variable name.

Setter in Java: While Setter sets or updates the value (mutators). It sets the value for any variable used in a class's programs. and starts with the word "set" followed by the variable name.

Syntax

```
class ABC{
    private variable;

    public void setVariable(int x){
        this.variable=x;
    }

    public int getVariable{
        return variable;
    }
}
```

Note: In both getter and setter, the first letter of the variable should be capital.

Examples of Getter and Setter in Java

Example 1:

```
// Java Program to Illustrate Getter and Setter

// Importing input output classes
import java.io.*;

// Class 1
// Helper class
class GetSet {

    // Member variable of this class
    private String name;

    // Method 1 - Getter
    public String getName() { return name; }

    // Method 2 - Setter
    public void setName(String N)
    {

        // This keyword refers to current instance itself
        this.name = N;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of class 1 in main() method
        GetSet obj = new GetSet();

        // Setting the name by calling setter method
        obj.setName("Geeks for Geeks");
        // Getting the name by calling getter method
        System.out.println(obj.getName());
    }
}
```

Output

Geeks for Geeks

Getter and Setter give you the convenience of entering the value of the variables of any data type by the requirement of the code. Getters and setters let you manage how crucial variables in your code are accessed and altered. It can be seen in the program discussed below as follows:

Example 2

```
// Java Program to Illustrate Getter and Setter

// Importing input output classes
import java.io.*;

class GetSet {
    // Member variable of this class
    private int num;

    // Method 1 - Setter
    public void setNumber(int number)
    {
        // Checking if number is between 1 to 10 exclusive
        if (number < 1 || number > 10) {

            throw new IllegalArgumentException();
        }
        num = number;
    }

    // Method 2 - Getter
    public int getNumber() { return num; }
}

// Class 2
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        GetSet obj = new GetSet();
```

```
// Calling method 1 inside main() method  
obj.setNumber(5);  
  
// Printing the number as setter above  
System.out.println(obj.getNumber());  
}  
}
```

Output

5

Explanation of the above program:

Here we can see that if we take a value greater than 10 then it shows an error, By using the setNumber() method, one can be sure the value of a number is always between 1 and 10. This is much better than updating the number variable directly.

Note: *This could be avoided by making the number a private variable and utilizing the setNumber method. Using a getter method, on the other hand, is the sole way to read a number's value.*

04.11.24

Default: access specifier

If you don't use any modifier, it is treated as default by default.

The default modifier is accessible only within package. It can not accessed from outside the package.

• Package? (C: #include <string.h>
 Java: import util)

package > class > methods
 + files + variables

It provides more accessibility than private.

But it's more restricted than protected and public.

in another folder

Package pack;

class A{

void msg(){}

System.out.println("Hello");}

}

could access it
declare

Package my.pack;

import pack.*;

all

class B{

public static void main(String args[])

{ A obj = new A(); }

obj.msg();

}

Output: Compile time error

because default access modifier
cannot access external package

#Protected:

The protected access modifier is accessible within package and outside the package but through inheritance only.

Package pack;

public class A {

protected void msg() {

System.out.println("Hello");

}

Package mypack;

import pack.*;

class B extends A {

public static void main(String args[]) {

`B obj = new B();`

`obj.msg();`

obj is an

`B`

method of class

whose definition is present in the object's class

Output: Hello

- Explore NetBeans project types.

slide chart for access modifiers.

"access modifiers" page 42

- Polymorphism → overloading concepts

Method overriding is one of the ways by which Java achieves Run Time Polymorphism.

32

Polymorphism:

There are two types of polymorphism

→ compile-time polymorphism

(when the decision to use a overloaded method is done while compiling)

→ runtime polymorphism

(Method overriding)

compile-time polymorphism

class Calculation {

 int add(int a, int b) {

 return a+b;

}

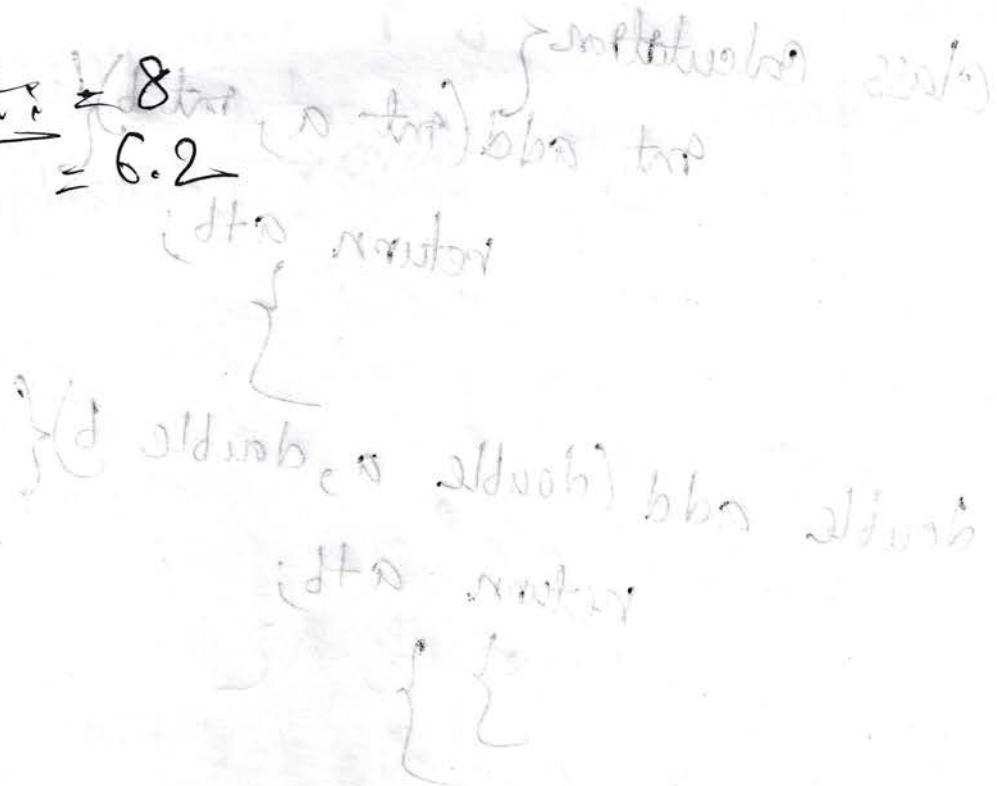
 double add(double a, double b) {

 return a+b;

}

```
public class CompleteTimePolymorphism {
    public static void main (String args[]){}
        Calculation calc = new Calculation();
        System.out.println ("Sum of integers:" + calc.add (5,3));
        System.out.println ("Sum of double:" + calc.add (2.5,3.7));
    }
}
```

Output:
= 8
= 6.2



Solve: 2(b) → (Solve with ~~no~~ method overloading)

```
{[{"Topic part 2 from blog site reading"}]
class Book{
```

```
    String title; wwww = Ed. No. 9
```

```
    String author; wwww = Ed. No. 9
```

```
    int year; "Author: 19.11"
```

```
void BookInfo(String t){
```

```
    title = t;
```

```
System.out.println("Title:" + title); }
```

```
void BookInfo(String t, String a){
```

```
    title = t; author = a;
```

```
System.out.println("Title:" + title + "Author:" + author); }
```

```
void BookInfo(String t, String a, int y){
```

```
    title = t; author = a; year = y;
```

```
System.out.println("Title:" + title + "Author:" + author + "Year:" + year); }
```

```
}
```

class main

public static void main(String args[]){

 Book b1 = new Book();

 Book b2 = new Book();

 Book b3 = new Book();

 b1.BookInfo("Discrete math");

 b2.BookInfo("Math math", "Teacher");

 b3.BookInfo("SPL", "Schildt", 1992);

can be
skipped

[10.11.24]

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 based on the object being referred to by the reference variable.

most recent method

child class's override
parent class's method

while method
overloading
inheritance

Up casting:

If the reference variable of parent class refers to the object of child class, it is known as upcasting.

Typecasting is one of the most important concepts which basically deals with the conversion of one data type to another datatype implicitly or explicitly. In this article, the concept of typecasting for objects is discussed.

Just like the data types, the objects can also be typecasted. However, in objects, there are only two types of objects, i.e. parent object and child object. Therefore, typecasting of objects basically means that one type of object (i.e.) child or parent to another. There are two types of typecasting. They are:

1. Upcasting: Upcasting is the typecasting of a child object to a parent object.

Upcasting can be done implicitly. Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can access the overridden methods.

2. Downcasting: Similarly, downcasting means the typecasting of a parent object to a child object. Downcasting cannot be implicit.

```
// Java program to demonstrate
// Upcasting Vs Downcasting

// Parent class
class Parent {
    String name;

    // A method which prints the
    // signature of the parent class
    void method()
    {
        System.out.println("Method from Parent");
    }
}

// Child class
class Child extends Parent {
    int id;

    // Overriding the parent method
    // to print the signature of the
    // child class
```

```
@Override void method()
{
    System.out.println("Method from Child");
}
}

// Demo class to see the difference
// between upcasting and downcasting
public class GFG {

    // Driver code
    public static void main(String[] args)
    {
        // Upcasting
        Parent p = new Child();
        p.name = "GeeksforGeeks";

        //Printing the parentclass name
        System.out.println(p.name);
        //parent class method is overridden method hence this will be executed
        p.method();

        // Trying to Downcasting Implicitly
        // Child c = new Parent(); - > compile time error

        // Downcasting Explicitly
        Child c = (Child)p;

        c.id = 1;
        System.out.println(c.name);
        System.out.println(c.id);
        c.method();
    }
}
```

Output

```
GeeksforGeeks
Method from Child
GeeksforGeeks
1
Method from Child
```

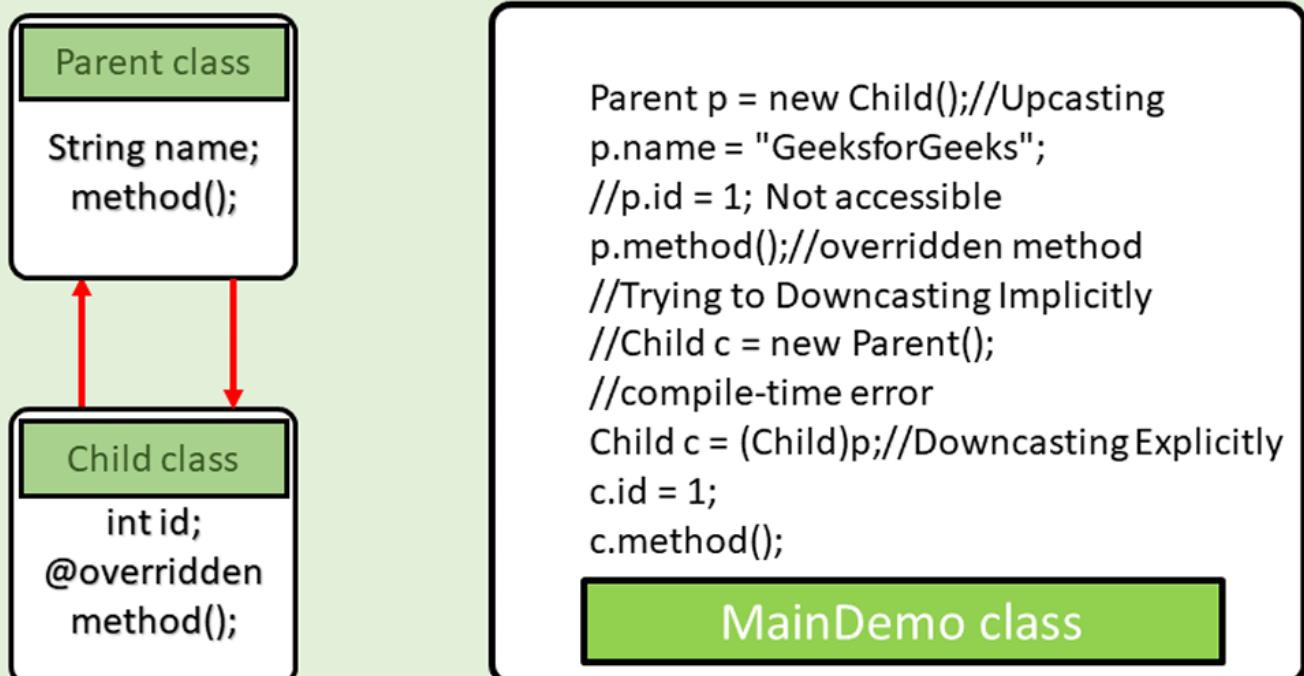


Figure to illustrate the Concept of Upcasting Vs Downcasting

1. Syntax of Upcasting:

```
Parent p = new Child();
```

- Upcasting will be done internally and due to upcasting the object is allowed to access only parent class members and child class specified members (overridden methods, etc.) but not all members.

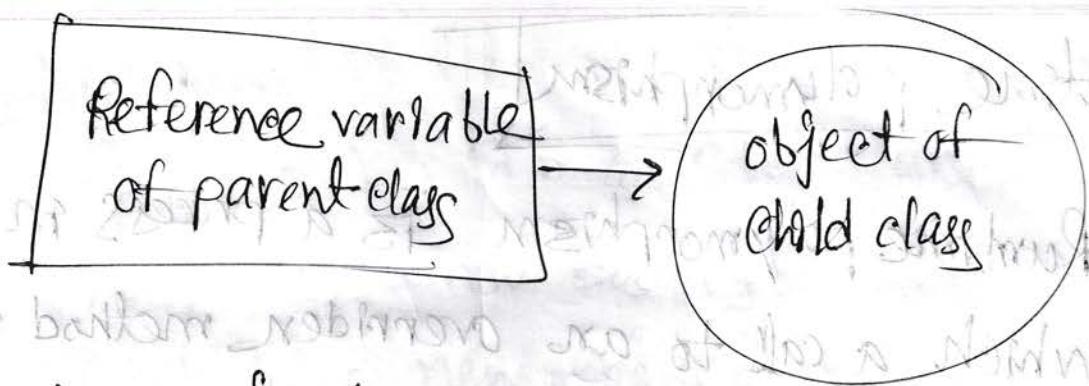
```
// This variable is not
// accessible
p.id = 1;
```

1. Syntax of Downcasting:

```
Child c = (Child)p;
```

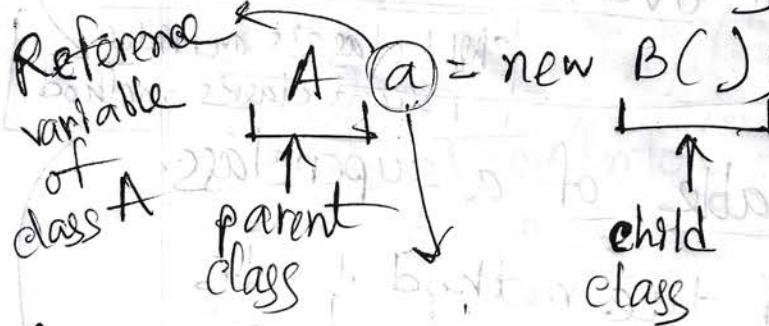
- Downcasting has to be done externally and due to downcasting a child object can acquire the properties of the parent object.

```
c.name = p.name;
i.e., c.name = "GeeksforGeeks"
```



class A { }

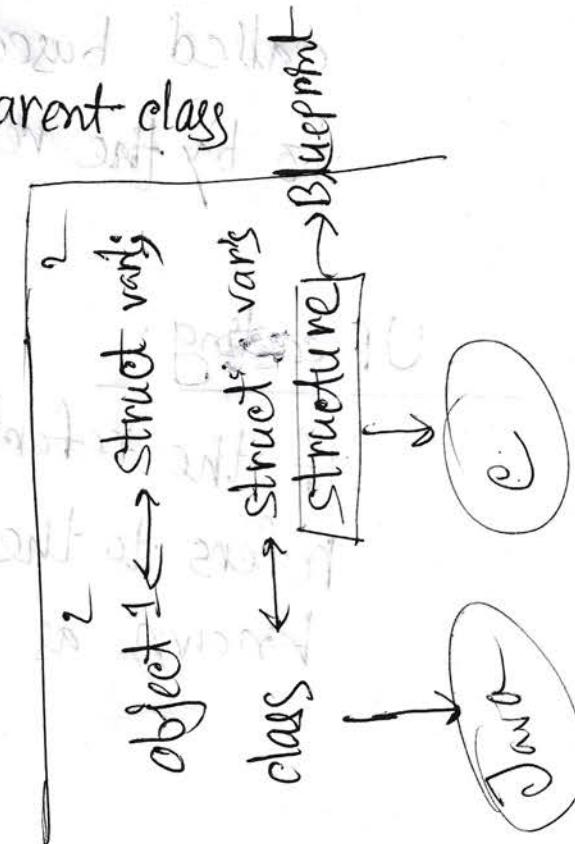
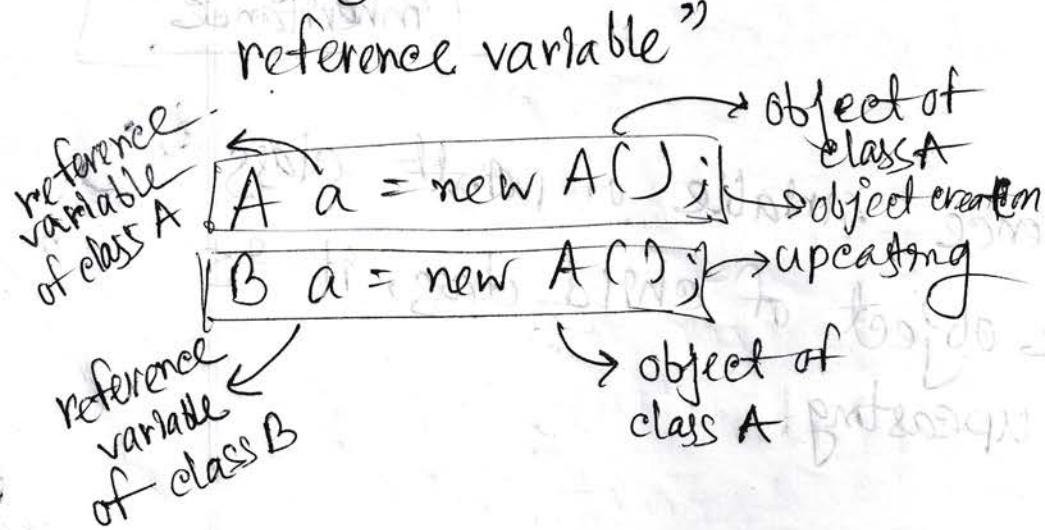
class B extends A { }



upcasting in inheritance

for a parent class and subclasses

"putting child class object in parent class reference variable"

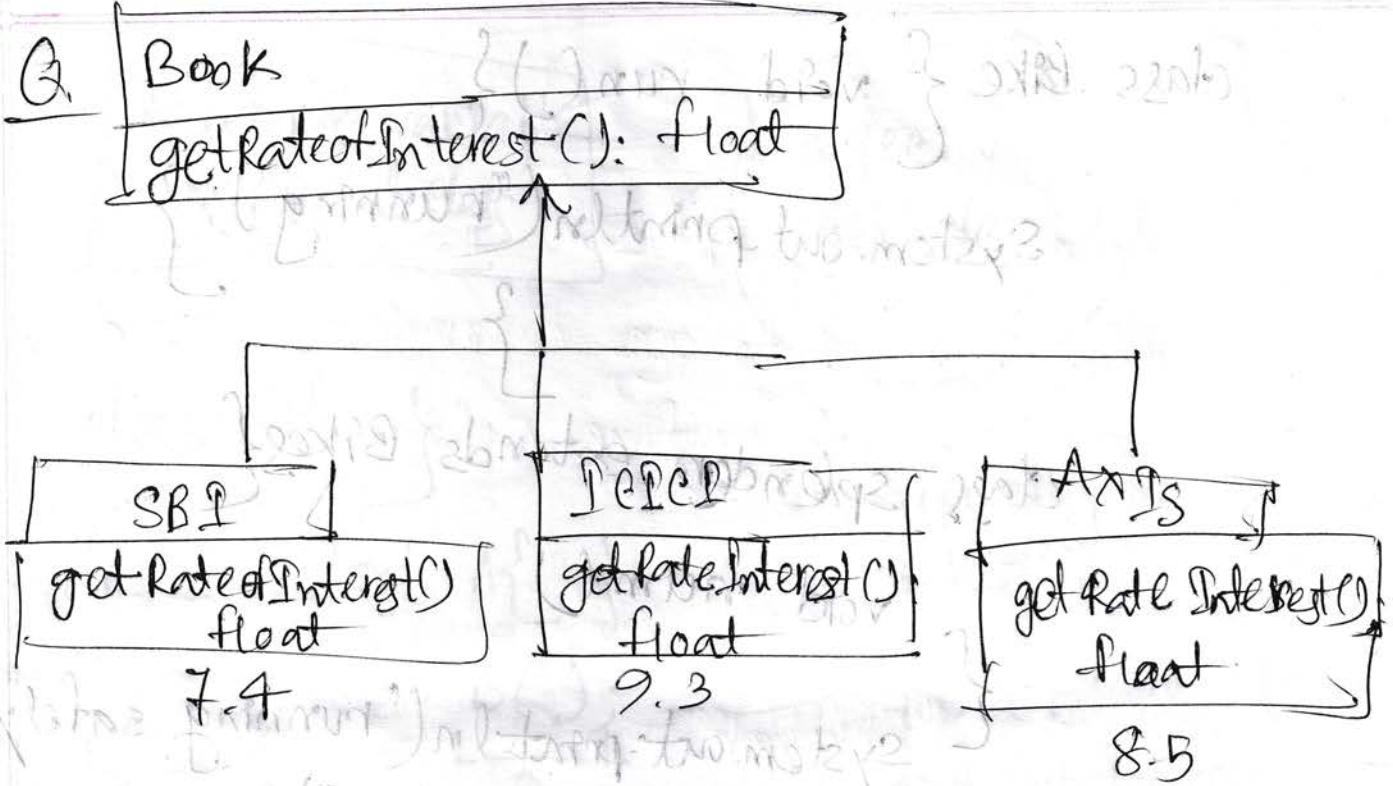


```
class Bike { void run() {
    System.out.println("running"); }}
```

```
class Splendor extends Bike {
    void run() {
        System.out.println("running safely
with 60km/h"); }}
```

```
public static void main(String args[]) {
    Bike b = new Splendor();
    b.run();
}}
```

- useful for hierarchical inheritance
(and multilevel inheritance) (3)



\Rightarrow

```

class Book {
    float a;
    getRateofInterest() {
        a = 5.0F;
        System.out.println("a");
    }
}
  
```

```

class SBP extends Book {
    a = 7.4F;
}

void getRateofInterest() {
    System.out.println("a");
}
  
```

class PCLC1 extends Book { $a = 9.3\text{f};$

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

class AXIS extends Book {

void getRateOfInterest() { $a = 8.5\text{f};$
 System.out.println("a"); } }

}

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

Bank b = new SBP(); → creates a new
 object under
 differ child
 classes but using
 the same parents
 class's variable
 ↓
 (name b)

b.getRateOfInterest();

Bank b = new PCLC1();

b.getRateOfInterest();

Bank b = new AXIS();

b.getRateOfInterest(); } }

Bank b = new

103.11.24

COPT

Overriding is a process
and not a key word

[@Override]

Just an Annotation used in IDEs
as convention

```
23
24     double area() {
25         Add @Override Annotation >   ⇢
26             return dim1 * dim2;
27         }
28     }
29
30 class Triangle extends Fig.
```

```
23
24     @Override
25     double area() {
26         System.out.println("Inside area of triangle");
27         return dim1 * dim2;
28     }
29
30 }
```

Yes, the `@Override` annotation in Java is optional; you can override a method without explicitly using it, but it's considered a good practice to use it because it helps catch potential errors by informing the compiler that you intend to override a method, leading to a compile-time error if the method signature doesn't match a superclass method properly.

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called through its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.  
class A {  
    int i, j;  
    A(int a, int b) {  
        i = a;  
        j = b;  
    }  
  
    // display i and j  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

(Metaphor!!)

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    // display k - this overrides show() in A  
    void show() {  
        System.out.println("k: " + k);  
    }  
  
    class Override {  
        public static void main(String[] args) {  
            B subOb = new B(1, 2, 3);  
  
            subOb.show(); // this calls show() in B  
        }  
    }  
}
```

It is optional to
add @Override
(for some reason)

Built-in methods can be thought of as
parent class for the user defined methods

1. class Child{
2. Constructor(){
3. //object properties
4. }
5. //when object is printed, using override, its properties are converted into String and printed instead
6. `@Override`
7. `public String toString(){`
8. `return "String";`
9. `}`
10. class Main{
11. public static void main(String args[]){
12. Child object = new Child();
13. System.out.println(object); //object printed
14. }

Output:

String //printed instead of object memory address

The output produced by this program is shown here:

k: 3

Java `toString()` Method

If you want to represent any object as a string, **`toString()` method** comes into existence.

The `toString()` method returns the String representation of the object.

If you print any object, Java compiler internally invokes the `toString()` method on the object. So overriding the `toString()` method, returns the desired output, it can be the state of an object etc. depending on your implementation.

Advantage of Java `toString()` method

By overriding the `toString()` method of the `Object` class, we can return values of the object, so we don't need to write much code.

Understanding problem without `toString()` method

Let's see the simple code that prints reference.

Student.java

```
class Student{
    int rollno;
    String name;
    String city;

    Student(int rollno, String name, String city){
        this.rollno=rollno;
        this.name=name;
        this.city=city;
    }

    public static void main(String args[]){
        Student s1=new Student(101,"Raj","lucknow");
        Student s2=new Student(102,"Vijay","ghaziabad");
    }
}
```

```
System.out.println(s1);//compiler writes here s1.toString()
System.out.println(s2);//compiler writes here s2.toString()
}
}
```

Output:

```
Student@1fee6fc
Student@1eed786
```

As you can see in the above example, printing s1 and s2 prints the **hashcode values of the objects** but I want to print the values of these objects. Since Java compiler internally calls `toString()` method, overriding this method will return the specified values. Let's understand it with the example given below:

Example of Java `toString()` method

Let's see an example of `toString()` method.

Student.java

```
class Student{
    int rollno;
    String name;
    String city;

    Student(int rollno, String name, String city){
        this.rollno=rollno;
        this.name=name;
        this.city=city;
    }

    public String toString(){//overriding the toString() method
        return rollno+" "+name+" "+city;
    }

    public static void main(String args[]){
        Student s1=new Student(101,"Raj","lucknow");
        Student s2=new Student(102,"Vijay","ghaziabad");
    }
}
```

```
System.out.println(s1);//compiler writes here s1.toString()
System.out.println(s2);//compiler writes here s2.toString()
}
}
```

Output:

```
101 Raj lucknow
102 Vijay ghaziabad
```

In the above program, Java compiler internally calls **toString()** method, overriding this method will return the specified values of **s1** and **s2** objects of Student class.

Java **toString()** method MCQ

1. What is the default behavior of the **toString()** method in the Object class?

1. Returns the object's state
2. Returns the object's class name and hashCode
3. Returns null
4. Returns the object's memory address

 **Show Answer**

Workspace

2. Which class should override the **toString()** method to provide a meaningful string representation?

1. Any class that requires a custom string representation
2. Only utility classes
3. Only data classes
4. Only service classes

 **Show Answer**

Workspace

3. What is the recommended way to generate a string representation of an object in a custom **toString()** method?

Method overriding is one of the ways by which Java achieves [Run Time Polymorphism](#). The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

Rules for Java Method Overriding

1. Overriding and Access Modifiers

The [access modifier](#) for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass. Doing so will generate a [compile-time error](#).

2. Final methods can not be overridden

If we don't want a method to be overridden, we declare it as [final](#). Please see [Using Final with Inheritance](#).

3. Static methods can not be overridden(Method Overriding vs Method Hiding):

When you define a static method with the same signature as a static method in the base class, it is known as [method hiding](#). The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

4. Private methods can not be overridden

[Private methods](#) cannot be overridden as they are [bonded during compile time](#). Therefore we can't even override private methods in a subclass.(See [this](#) for details).

5. The overriding method must have the same return type (or subtype)

From Java 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child's return type should be a sub-type of the parent's return type. This phenomenon is known as the [covariant return type](#).

6. Invoking overridden method from sub-class

We can call the parent class method in the overriding method using the [super keyword](#).

Overriding and Constructor

We [can not override the constructor](#) as the parent and child class can never have a constructor with the same name(The constructor name must always be the same as the Class name).

Overriding and Exception-Handling

Below are two rules to note when overriding methods related to exception handling.

Rule #1

If the super-class overridden method does not throw an exception, the subclass overriding method can only throw [the unchecked exception](#), throwing a checked exception will lead to a compile-time error.

Rule #2

If the superclass overridden method does throw an exception, the subclass overriding method can only throw the same, subclass exception. Throwing parent exceptions in [the Exception hierarchy](#) will lead to compile time error. Also, there is no issue if the subclass overridden method is not throwing any exception.

Overriding and Abstract Method

Abstract methods in an interface or abstract class [are meant to be overridden](#) in derived concrete classes otherwise a compile-time error will be thrown.

→ foolproofing

16.16.24

Abstraction in Java: Abstraction in Java is a process of hiding the implementation details and showing only the functionality to the user. It shows only the essential things to the user and hides the internal details.

Ways to achieve abstraction:

- using Abstract class (0 to 100%)
- using Interface (100%)

Blueprint of a class

- Abstract and non-abstract, conversion into abstract data
- postfix, memory management (?), conditions (?)

Process:

public Abstract class Shape {

abstract void draw(); }
 ↑
 abstract method/non-abstract
 (when no keyword used) }
 ↑
 used before return type
 key word used before class declaration
 } }
 ↑
 only exists inside abstract class
 } }
 ↑
 // only initialized
 // can be used only through inheritance

- * methods must be used by overriding
- * An abstract class must be declared with an abstract keyword.
- * It can have abstract method and non-abstract method
- * It can not be instantiated → can't create objects under this class, but can create reference variable. [Upcasting]
- * can use static, final and constructor.

Syntax: abstract class Bike{
 abstract void run(); }
class Honda4 extends Bike{
 void run(){
 System.out.println("running Safety"); }
 }
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run(); } }

Bike obj; → under Bike class

obj = new Honda4(); → under Honda4 class

Sout = System.out.println

NetBeans Shortcut

Q. Mobile User

abstract sendMessage().void

Rahim
Send Message()

Karim
Send Message()

Mam

Hi, this is Rahim

→ public abstract class Mobile_User
abstract void sendMessage();

class Mam [extends] Mobile_User {

public static void main (String args []) {

Mobile_User a = new Rahim();

a.sendMessage();

Mobile_User a = new Karim();

a.sendMessage();

}

class Rahim extends Mobile_User {

void sendMessage () { System.out.println ("Hi this is Rahim"); }

class Karim extends Mobile_User {

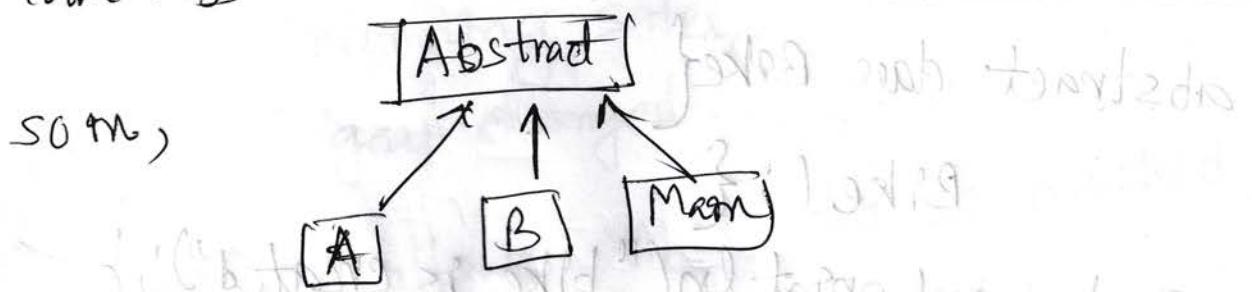
void sendMessage () { System.out.println ("Hi this is Karim"); }

28
Java

(PPT)
+

28

Main class can access all of code
Main class can control



Main class can control

#Abstract class have constructor and Methods

```
abstract class Bike{
```

```
    Bike(){}
```

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

```
    abstract void run();
```

```
    void changeGear(){
```

```
        System.out.println("gear change");}
```

```
}
```

```
class Honda extends Bike{
```

```
    void run(){
```

```
        System.out.println("running safely");}
```

```
class Test Abstract{
```

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

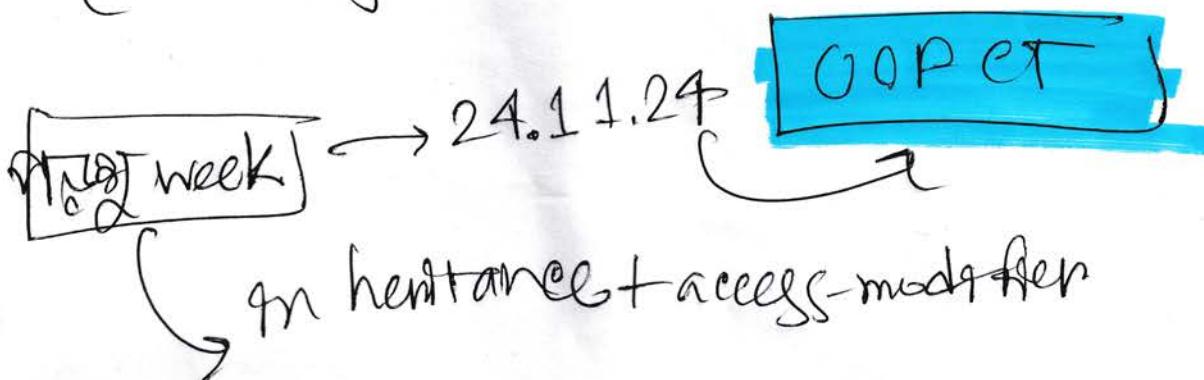
```
        Bike obj = new Honda();
```

```
        obj.run();
```

```
        obj.changeGear();}}
```

Output: bike is created
running safely...
gear changed.

Inheritance allows `Bike()` to be invoked
when object is created under `Honda()`;
(when object is created, all constructors are called
until parent class)



does invoking child class constructor invoke parent class constructors implicitly



Yes, in Java, invoking a child class constructor **implicitly invokes the parent class's constructor**. This happens because the constructor of a child class starts with an implicit call to the parent class's no-argument constructor (`super()`), even if you don't explicitly write it in your code.

How Parent Constructors Are Invoked

- When a child class constructor is called, the first thing it does is call the parent class's no-argument constructor.
- If the parent class does not have a no-argument constructor, and you don't explicitly call another constructor in the parent class using `super(arguments)`, a compilation error occurs.

3. Explicitly using `super(arguments)` allows you to call specific constructors in the parent class.

Example with Implicit Call to Parent Constructor

```
java

class Parent {
    Parent() {
        System.out.println("Parent constructor invoked");
    }
}

class Child extends Parent {
    Child() {
        System.out.println("Child constructor invoked");
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
    }
}
```

Output:

```
kotlin

Parent constructor invoked
Child constructor invoked
```

Example with Explicit Call to Parent Constructor

If the parent class has parameterized constructors, you need to explicitly use `super(arguments)` in the child class constructor.

```
java

class Parent {
    Parent(String message) {
        System.out.println("Parent constructor: " + message);
    }
}

class Child extends Parent {
    Child() {
        super("Hello from Parent!"); // Explicitly calling parent constructor
        System.out.println("Child constructor invoked");
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
    }
}
```

Output:

```
kotlin

Parent constructor: Hello from Parent!
Child constructor invoked
```

Key Points**1. Implicit Call:**

- o If you don't explicitly invoke a parent class constructor using `super()`, Java will insert a call to the no-argument constructor (`super()`) automatically.

2. Explicit Call:

- o Use `super(arguments)` to invoke a specific parent class constructor with parameters.

3. Compilation Errors:

- o If the parent class does not have a no-argument constructor and you don't explicitly call another parent constructor, a compilation error will occur.

Scenario Without Parent No-Arg Constructor

```
java

class Parent {
    Parent(String message) {
        System.out.println("Parent constructor: " + message);
    }
}

class Child extends Parent {
    Child() {
        // Implicit call to super() will cause a compilation error because
        // Parent(String) is the only constructor available.
        System.out.println("Child constructor invoked");
    }
}
```

Compilation Error:

```
vbnnet

Constructor Parent in class Parent cannot be applied to given types;
 required: String
 found: no arguments
```

Conclusion

Yes, the constructor of a child class implicitly invokes the parent class's no-argument constructor if no explicit call to `super(arguments)` is made. To avoid issues, ensure the appropriate constructor in the parent class is accessible either implicitly or explicitly.

Abstraction

Definition: Abstraction focuses on **hiding implementation details** while exposing only the essential features and functionalities.

Key Points:

- Deals with **what** a class or object does, not **how** it does it.
- Achieved using **abstract classes** and **interfaces** in Java.
- Promotes design flexibility and reduces complexity by focusing on high-level operations.
- Example:
 - A `List` in Java (like `ArrayList` or `LinkedList`) provides methods like `add()`, `remove()`, or `get()`, but hides how the underlying data structure manages those operations.

Implementation:

1. Abstract Class:

```
java

abstract class Animal {
    abstract void makeSound(); // Abstract method
}
class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark");
    }
}
```

2. Interface:

```
java

interface Shape {
    void draw();
}
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
```

Encapsulation

Definition: Encapsulation is about **hiding data** (information hiding) by restricting direct access to an object's internal state and only allowing controlled access through public methods.

Key Points:

- Focuses on **how** data is accessed or modified.
- Achieved using **private fields** and **public getter/setter methods** in Java.
- Protects the internal state of the object and ensures data integrity.
- Example:
 - A class `Account` with private balance data ensures users can access or update it only through predefined methods like `deposit()` or `withdraw()`.

Implementation:

```
java
```

```
class Account {  
    private double balance;  
  
    public Account(double initialBalance) {  
        this.balance = initialBalance;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void deposit(double amount) {  
        if (amount > 0) {  
            balance += amount;  
        }  
    }  
  
    public void withdraw(double amount) {  
        if (amount > 0 && balance >= amount) {  
            balance -= amount;  
        } else {  
            System.out.println("Insufficient funds");  
        }  
    }  
}
```

Comparison: Abstraction vs Encapsulation

| Aspect | Abstraction | Encapsulation |
|----------------|---|--|
| Focus | Hides implementation details (functionality). | Hides internal data (information). |
| Implementation | Achieved through abstract classes and interfaces. | Achieved through access modifiers (private, public). |
| Purpose | Reduce complexity and promote design flexibility. | Ensure data security and integrity. |
| Access | Focuses on showing the required functionality. | Focuses on restricting direct access to data. |
| Example | Abstract method or interface definition. | Use of private fields with public getters/setters. |

When to Use?

- Use **abstraction** to define the behavior or contract of an object without specifying implementation.
- Use **encapsulation** to enforce data security and manage access effectively.

Both principles together improve code readability, maintainability, and security.

similar to

Interface in Java:

- An interface in java is a blueprint of a class . It has static constants and abstract methods.
- The interface in java is a mechanism to achieve abstraction. There can be only abstract methods in java interface, not method body. It used to achieve abstraction and multiple inheritance in java.

key points:

- ① Java Interface also represents the IS-A Relationship.
- ② It cannot be instantiated just like the abstract class.
 - can't create objects
- ③ Since Java 8, we can have default and static methods in an interface.

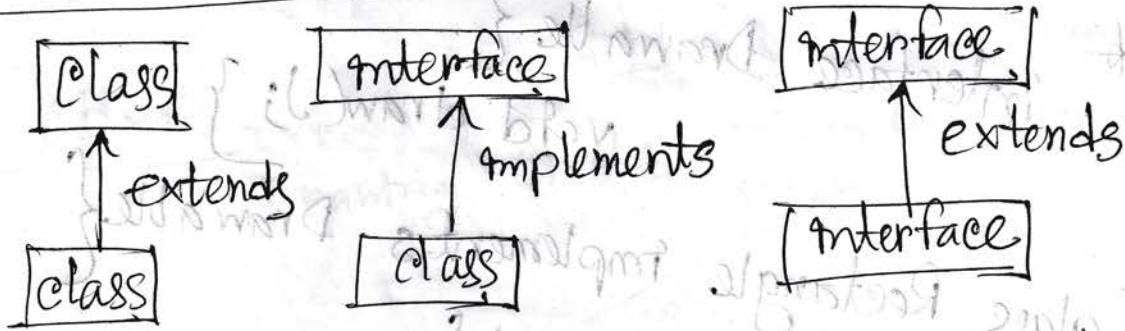
④ Since Java 9, we can have private methods in an interface.

Interface Syntax:

```
interface Interface_Name {  
    abstract method();  
    method(); // auto  
    abstract  
}  
  
void method1();  
abstract void method1();
```

→ Syntax.
for codes, return type
is needed

Relationship between Class and Interface:



class
interface
together

```
* interface printable {  
    void print(); }  
  
class InterfaceExample implements printable {  
    public void print() {  
        System.out.print("Hello");  
    }  
  
    public static void main(String args[]) {  
        InterfaceExample obj = new InterfaceExample();  
        obj.print(); } }
```

* interface Drawble {
 void draw(); }

```
class Rectangle implements Drawble {  
    public void draw() {  
        System.out.println("drawing Rectangle"); } }
```

7

```

class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class TestInterface {
    public static void main() {
        Drawable d = new Circle();
        d.draw();
    }
}

```

Output:

Drawing Circle.

holds
reference
variable for
class Drawable

*
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();
 }
}

POP

18-11-24

- private can be used from Java-9 and up
- "public" or "default" before "class"
- static , private method used in interface?

~~Q. static , private method used in interface?~~

check theory

Interface has [100% abstraction]

variable and method data declared inside are automatically abstracted

Abstract class has [0 - 100% abstraction]

the abstracted block can have non-abstract data as well

Interface Methods Usage Java



can I use interface methods without overriding in java



In Java, an interface is a contract that requires classes implementing it to override and provide concrete implementations for its methods (except for `default` methods, which already have a concrete implementation). If you attempt to use an interface method in a class without overriding it, it won't work unless one of the following applies:

1. Abstract Classes

If your class is declared as `abstract`, you can choose not to override the interface methods. This allows the subclass of your abstract class to override the methods instead.

```
java

interface MyInterface {
    void myMethod();
}

abstract class MyAbstractClass implements MyInterface {
    // Not overriding myMethod here is allowed because the class is abstract
}

class MyConcreteClass extends MyAbstractClass {
    @Override
    public void myMethod() {
        System.out.println("Implemented in concrete class.");
    }
}
```

2. `default` Methods in Interfaces

Starting with Java 8, interfaces can define `default` methods, which have an actual implementation. Classes implementing such an interface can use these methods without overriding them.

```
java

interface MyInterface {
    default void myMethod() {
        System.out.println("Default implementation.");
    }
}

class MyClass implements MyInterface {
    // No need to override myMethod, as the interface provides a default implementation
}

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

```

    MyClass obj = new MyClass();
    obj.myMethod(); // Output: Default implementation.
}
}

```

3. Static Methods in Interfaces

Java interfaces can also contain `static` methods (since Java 8). These methods belong to the interface itself and are not meant to be overridden.

```

java

interface MyInterface {
    static void staticMethod() {
        System.out.println("This is a static method in the interface.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyInterface.staticMethod(); // Call the static method directly using the interface name
    }
}

```

Summary

- **Required to override:** Interface methods without `default` implementations.
- **Not required to override:**
 - Methods in `abstract` classes that implement interfaces.
 - `default` methods.
 - `static` methods in interfaces (called directly via the interface name).

If you don't provide an override in a concrete class for methods without `default` implementations, you will get a compilation error.

```

package p07.finalterm;

// CASE 1: interface with abstract methods (no body, only declared)

/*interface Parent1{
    void fun(){ //error when no default access modifier
        System.out.println("Printed Parent1\n");
    }
}

interface Parent2{
    void fun(); //error when no default access modifier
}
*/
}

// CASE 2: interface with default methods

interface Parent1{
    default void fun(){
        System.out.println("Printed Parent1\n");
    }
}

/*interface Parent2{
    default void fun(){
        System.out.println("Printed Parent2\n");
    }
}
*/

// CASE 3: multiple inheritance with interface and class

class Parent2{
    public void fun(){
        System.out.println("Printed Parent1\n");
    }
}

// Main class for CASE 1 and 2

```

```

/*public class Main implements Parent2, Parent1{
    /*@Override
    public void fun(){ //printsonly Parent1 method
        System.out.println("Printed Parent1\n");
    }*/
    /* @Override
    public void fun(){ //prints both interface methods
        Parent1.super.fun();
        Parent2.super.fun();
    }

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

// Main class for CASE 3

public class Main extends Parent2 implements Parent1{

    /*
        @Override
        public void fun(){
            System.out.println("Printed Parent1\n");
        }*/
    @Override
    public void fun(){ //prints both interface methods
        Parent1.super.fun();
        Parent2.super.fun();
    }

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

```

The code provided demonstrates concepts related to Java interfaces, default methods, method overriding, and the challenges with multiple inheritance. Below are clarifications and a breakdown of the different sections:

CASE 1: Interfaces with Abstract Methods

In Java, an interface method without a body is implicitly abstract, meaning you cannot provide implementation within the interface unless using `default` or `static`. The code here is commented out because:

1. `void fun()` is not allowed to have a body without a modifier (`default` or `static`):

```
java
```

```
void fun() {  
    System.out.println("Printed Parent1\\n");  
}
```

- Java interfaces require methods to either be `default` (with body) or have no body at all if abstract.

2. These declarations will not compile until corrected.

CASE 2: Interfaces with `default` Methods

In this part, interfaces declare methods using the `default` modifier:

1. `Parent1` has:

```
java
```

```
default void fun() {  
    System.out.println("Printed Parent1\\n");  
}
```

- This is legal and allows the interface to have implementation.
2. `Main` can extend multiple interfaces but must resolve name conflicts. In this case, `Parent2` is commented out:

```
java
```

```
interface Parent2 {  
    default void fun() {  
        System.out.println("Printed Parent2\\n");  
    }  
}
```

- When both `Parent1` and `Parent2` provide default `fun()`, the `Main` class must override `fun()` to resolve ambiguity.

```
java
@Override
public void fun() { // Call both interfaces' methods
    Parent1.super.fun();
    Parent2.super.fun();
}
```

6

CASE 3: Multiple Inheritance with Class and Interface

Java allows a class to extend another class and implement multiple interfaces. This is demonstrated with:

1. Class `Parent2`:

```
java
class Parent2 {
    public void fun() {
        System.out.println("Printed Parent1\n");
    }
}
```

2. Class `Main`:

- `Main` extends `Parent2` and implements `Parent1`. If `Parent2` and `Parent1` both provide `fun()`:
 - The class (`Parent2`) takes precedence.
 - If the method in `Parent2` is overridden:

```
java
@Override
public void fun() {
    Parent1.super.fun();
    Parent2.super.fun(); // Error: Cannot call class method this way
}
```

- Calling `Parent2.super.fun()` will not compile, as `super` in this context can only refer to an interface default method.

3. Correct implementation should call methods properly:

```
java
```

```
@Override  
public void fun() {  
    Parent1.super.fun(); // Call interface method  
    super.fun(); // Call class method  
}
```

Execution

The corrected code in CASE 3:

```
java
```

```
class Parent2 {  
    public void fun() {  
        System.out.println("Printed Parent2\n");  
    }  
}  
  
interface Parent1 {  
  
    default void fun() {  
        System.out.println("Printed Parent1\n");  
    }  
}  
  
public class Main extends Parent2 implements Parent1 {  
    @Override  
    public void fun() {  
        Parent1.super.fun(); // Call default method of Parent1  
        super.fun(); // Call fun() of Parent2  
    }  
  
    public static void main(String[] args) {  
        Main obj = new Main();  
        obj.fun();  
    }  
}
```

Output:

```
Printed Parent1
```

```
Printed Parent2
```

```

package p07.finalterm;

// CASE 1: interface with abstract methods (no body, only
declared)

/*interface Parent1{
    void fun(){ //error when no default access modifier
        System.out.println("Printed Parent1\n");
    }
}

interface Parent2{
    void fun(); //error when no default access modifier
}
*/
}

// CASE 2: interface with default methods

interface Parent1{
    default void fun(){
        System.out.println("Printed Parent1\n");
    }
}

interface Parent2{
    default void fun(){
        System.out.println("Printed Parent2\n");
    }
}

// CASE 3: multiple inheritance with interface and class

class Parent2{
    public void fun(){
        System.out.println("Printed Parent1\n");
    }
}

// Main class for CASE 1 and 2

```

```

public class Main implements Parent2, Parent1{
    /*@Override
    public void fun(){ //printsonly Parent1 method
        System.out.println("Printed Parent1\n");
    }*/
    @Override
    public void fun(){ //prints both interface methods
        Parent1.super.fun();
        Parent2.super.fun();
    }

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

// Main class for CASE 3

/*public class Main extends Parent2 implements Parent1{

*/
/*
@Override
public void fun(){
    System.out.println("Printed Parent1\n");
}/*
@Override
public void fun(){ //prints both interface methods
    Parent1.super.fun();
    Parent2.super.fun();
}

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

This code explores various cases of inheritance using Java interfaces and classes, specifically focusing on:

Key Observations:

1. CASE 1: Abstract Methods in Interfaces (commented out)

- Declaring a method with a body (e.g., `void fun() {}`) inside an interface without the `default` or `static` modifier is **not allowed** in Java.
- Methods inside an interface without these modifiers must be abstract by default, and thus should not have a method body.

Example that results in a compilation error:

```
java
```

```
interface Parent1 {  
    void fun() { // Error  
        System.out.println("Printed Parent1\n");  
    }  
}
```

2. CASE 2: Default Methods in Interfaces

- Default methods in interfaces are allowed in Java 8 and later. They can have method bodies and allow classes implementing the interface to either use or override them.
- In the `Main` class, `Parent1.super.fun()` and `Parent2.super.fun()` are used to explicitly call the `fun` methods from both interfaces.

Execution will print:

```
plaintext
```

```
Printed Parent1
```

```
Printed Parent2
```

3. CASE 3: Interface and Class Multiple Inheritance (commented out)

- Java allows a class to inherit from both a regular class and multiple interfaces. When both the parent class and interface provide implementations for the same method, **class methods take priority**. This is because the Java inheritance rule favors class-based inheritance over interfaces.

How It Works:

- Default Interface Methods (CASE 2):

```
java
```

```
public void fun() {  
    Parent1.super.fun(); // Explicitly calls Parent1's default implementation  
    Parent2.super.fun(); // Explicitly calls Parent2's default implementation  
}
```

- Interface vs. Class Methods (CASE 3):

If the `Parent2` class (in the commented-out section) defines a method with the same signature as `Parent1`'s default method:

```
java
```

```
class Parent2 {  
    public void fun() {  
        System.out.println("Printed Parent1\n");  
    }  
}
```

Then the method from `Parent2` (the class) will override any implementation from the interface.

Suggestions:

To make the code clearer:

- CASE 1: Remove or fix the commented-out interfaces with method bodies since they aren't valid Java code. Instead, define the methods as default or abstract.
- CASE 3: If uncommented, clearly describe the resolution of conflicts when both an interface and a class define methods with the same signature.

#Static keyword:

The static keyword in Java used for memory management mainly.

We can apply static keyword in variable and method.

Doesn't require object creation for usage/access

→ this keyword applications

Syntax:

```
class Student { int rollno;
                String name;
                String college = "UIITS";
}
Student(int r, String n) {
    rollno = r;
    name = n;
}
void display() {
    System.out.println(rollno + " " + Name
                      + " " + college);
}
```

without static more memory would be allocated for each object created

static → object created with the value assigned to its instance (static or not)

Q(2)

```

public class TestStatic {
    public static void main(String args[]) {
        Student s1 = new Student(111, "karom");
        Student s2 = new Student(112, "Aryom");
        s1.display();
        s2.display();
    }
}

```

output:

111 karom OPTS

112 Aryom OPTS

(1) (111, "karom") -> 111 karom
 (2) (112, "Aryom") -> 112 Aryom

(1) (111, "karom") -> 111 karom
 (2) (112, "Aryom") -> 112 Aryom

(1) (111, "karom") -> 111 karom
 (2) (112, "Aryom") -> 112 Aryom

(1) (111, "karom") -> 111 karom
 (2) (112, "Aryom") -> 112 Aryom

{ (1) (111, "karom") -> 111 karom
 { (2) (112, "Aryom") -> 112 Aryom }

System:

```
class Student { int rollno;  
String name;  
static String college = "UITS";  
static void change() {  
college = "BBDIT"; }  
Student (int r, String n) {  
roll no = r;  
name = n; }  
void display() {  
System.out.println(String args[1]); }
```

call by class reference
↓ class name

```
public class Test { static void main(String args[]) {  
Student.change();  
Student s1 = new Student(111, "Karan");  
Student s2 = new Student(112, "Aryam");  
s1.display();  
s2.display(); } }
```

instead of `student.chang()`;

we could

`student.college` = "BBDIT";



we would need
some reference
to the class

ODP

Topics:

- final keyword
- Exception handling

Final keyword

The final keyword in java is used to restrict the user. The java final keyword can be used in many contexts.

1. variable
2. method
3. class (can't be inherited)

Key points:

- Stop value change
- Stop method Overriding
- Stop inheritance

- used for restriction

24.11.24

* ~~(class Bike {~~

```

class Bike {
    final int speedLimit = 90;
    void run() {
        speedLimit = 400; } // can't assign
    public static void main(String args[]) {
        Bike obj = new Bike();
        obj.run(); } }
```

Output: compile ^{time} error

* Java final method

```

class Bike {
    final void run() {
        System.out.println("Running"); }
    class Honda extends Bike {
        void run() {
            System.out.println("running safely with
                100 km"); }}
```

PS. H. PS
public static void main(String args[]){

Honda honda = new Honda();

honda.run(); } }

* final class Bike{ }

class Honda1 extends Bike{
 void run(){

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

public static void main(String args[]){

Honda1 honda = new Honda1();

honda.run(); }

Output: compile time error

Exception Handling

Arithmatic exception → $\frac{\text{variable}}{0} = \infty$

Building class →
↑

try catch block

The exception handling in java is one of the powerful mechanisms to handle the run time errors so that the normal flow of the application can be maintained.

Runtime error such as -

IO Exception, SQL Exception

Exception

↓
IOException

SQL "

Runtime "

Arithmatic exception

↓ Nullpointer "

↓ Numberformat "

↓ Index of bound "

Error

↓
Stack overflow
Out of memory

(add after P12)

15.12.24

1. Input - output
2. Database
3. Runtime
 4. Arithmetic invalid operations
 5. ~~if~~ Pointer address null (no address)
 6. Typecasting/parsing error
 7. Index size error
8. Stack size exceed ↴ overflow
9. memory limit exceed / ~~buff~~ data can't be found
within malloc

Runtime ...
PO Exception, SQL Exception

- Exception
- 1 IOException
 - 2 SQL "
 - 3 Runtime "
 - 4 Arithmetic exception
 - 5 Nullpointer "
 - 6 Numberformatte "
 - 7 Index ^{out} of Bounds "

- Error
- 8 Stack overflow
 - 9 Out of memory

• Threading, multithreading

Keywords:

- ① try
 - ② catch
 - ③ finally
 - ④ throw
 - ⑤ throws
- keep the codes inside
block that might
show error
- will work either
way. "must be executed"

Syntax:

try {

}

catch (Exception e) {

}

finally {

}

Reference to Exception classes
(can specify certain classes
like "Runtime e"
or, "Arithmatic e")

```

* class javaExceptionExample {
    public static void main(String args[]){
        try {
            int data = 100 / 0;
        } catch (ArithmaticException e) {
            System.out.println("EXCEPTION");
            System.out.println("Rest of the code");
        }
    }
}

alt: S.O.P(e);

```

outputs

alt: Exception in thread main java.lang.ArithmaticException
on by zero

EXCEPTION

rest of the code

printed when e is printed

printed when "EXCEPTION" is printed

Next → exception handling

→ bring both Khaka

(+ OOP Lab 11)

OOP

works like
if(f) if(f)
checks all
catch block
or, switch-case
without
breaks

#Java Multiple - ~~try~~ catch block:

public class MultipleCatch block

k1{

public static void main (String args []){

try {

int a[[↑]5] = new int [5];
no size given

a[5] = 30 / 0; } } } }

catch (ArithmaticException e)

{ System.out.println ("Arithmatic Exception occurs"); } }

out

catch (ArrayIndexOutOfBoundsException e)

{ System.out.println ("Array Index Out of Bounds Exception"); } }

catch (ArrayIndexOutOfBoundsException e)

{ System.out.println ("Array Index Out of Bounds Exception"); } }

{ System.out.println ("Array Index Out of Bounds Exception"); } }

catch (Exception e) {

System.out.println ("Parent exception occurs"); } }

System.out.println ("Rest of the code"); } }

default
case
(for all other
exceptions)

(+ OOP
Lab 11)

08-12-24

#Throw and Throws:

Throw: The throw keyword in Java is

used to explicitly throw an exception from a method or any block of code. The throw keyword is mainly used to throw custom exception.

Throws: Throws is a keyword in Java that

is used in the signature of a method to

indicate that this method might throw one of the listed type exceptions.

for
throws,
replace
this line
with

```
public class A1 {  
    public static void main(String args[ ]) {  
        try { double x = 3/0; }  
        throw new ArithmeticException();  
    }  
    catch(ArithmeticException e) {}  
}
```

(Throws)

```
// public static void main(String args[ ])  
throws IOException  
/* rest of the code */
```

makes try-catch block unnecessary as throws will output default exception messages ~~not~~ regardless of try-catch blocks written.

(in a single line)

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowbleInstance;
```

Here, *ThrowbleInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String[] args) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

This program gets two chances to deal with the same error. First, **main()** sets up an exception context and then calls **demoproc()**. The **demoproc()** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args) {
        throwOne();
    }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne()** throws **IllegalAccessException**. Second, **main()** must define a **try / catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

When exceptions are thrown, execution in a method takes a rather abrupt, **nonlinear path** that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to **return prematurely**. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will **not want** the code that closes the file to be bypassed by the **exception-handling mechanism**. The **finally** keyword is designed to address this **contingency**.

finally creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block. **The finally block will execute whether or not an exception is thrown.** If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an **uncaught exception** or an **explicit return** statement, the **finally** clause is also executed just before the method returns. This can be **useful for closing file handles** and **freeing up any other resources** that might have been allocated at the beginning of a method with the intent of disposing of them before returning. **The finally clause is optional.** However, each **try** statement requires at least **one catch or a finally clause.**

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

In this example, **procA()** **prematurely breaks out of the try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB()** returns. In **procC()**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

REMEMBER If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }

    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }

    // Execute a try block normally.
    static void procC() {
        try {
            System.out.println("inside procC");
        } finally {
            System.out.println("procC's finally");
        }
    }

    public static void main(String[] args) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        procB();
        procC();
    }
}
```

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

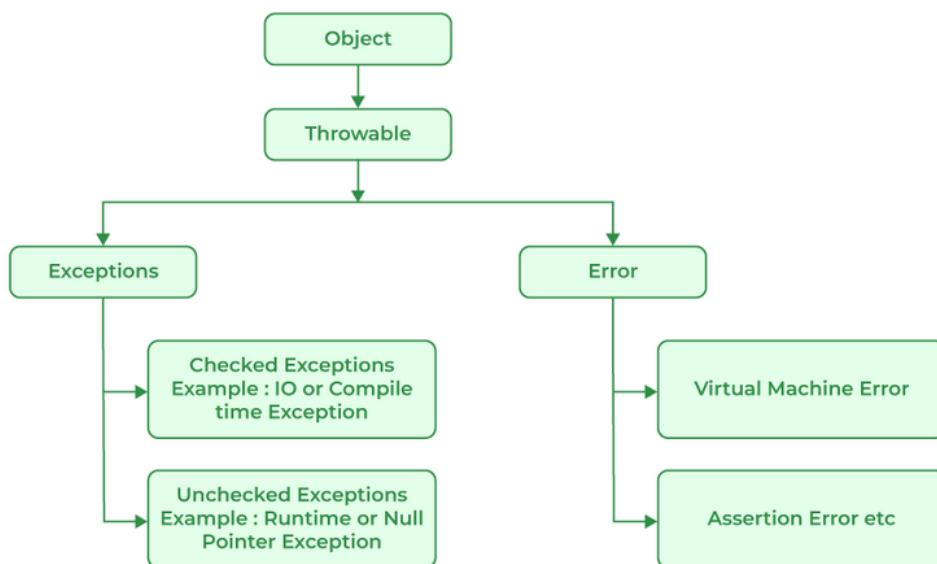
Difference between Error and Exception

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

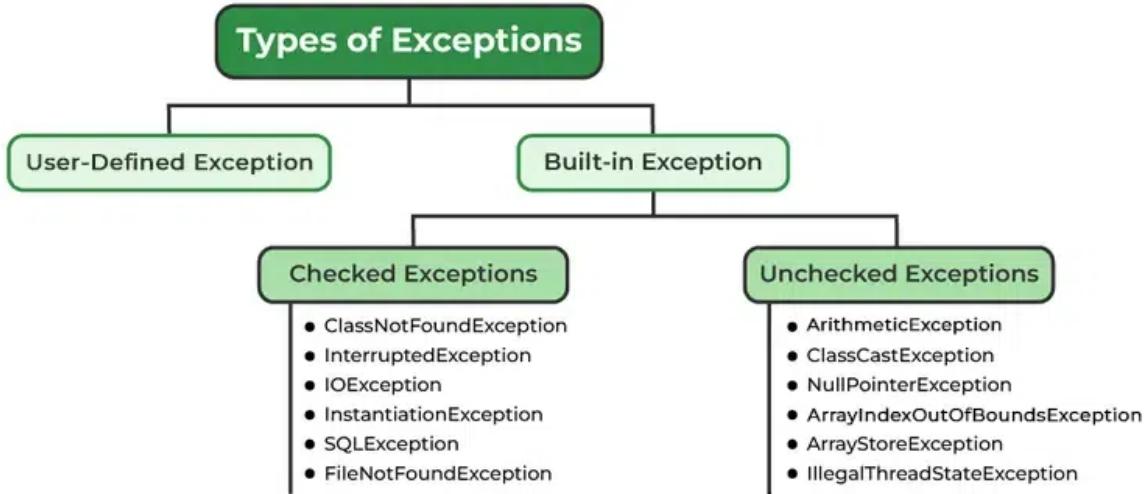
Exception Hierarchy

All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. `NullPointerException` is an example of such an exception. Another branch, **Error** is used by the Java run-time system([JVM](#)) to indicate errors having to do with the run-time environment itself(JRE). `StackOverflowError` is an example of such an error.



Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. Built-in Exceptions

- Checked Exception
- Unchecked Exception

2. User-Defined Exceptions

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Note: For checked vs unchecked exception, see [Checked vs Unchecked Exceptions](#)

Output

```
java.lang.ArithmetricException: / by zero
at GFG.main(File.java:10)
```

2. `toString()`

The `toString()` method prints exception information **in the format** of the Name of the exception: description of the exception.

Example:

Java

```
1 //program to print the exception information using
2 //toString() method
3
4 import java.io.*;
5
6 class GFG1 {
7     public static void main (String[] args) {
8         int a=5;
9         int b=0;
10        try{
11            System.out.println(a/b);
12        }
13        catch(ArithmetricException e){
14            System.out.println(e.toString());
15        }
16    }
17 }
```

Output

```
java.lang.ArithmetricException: / by zero
```

How Does JVM Handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an **Exception Object** and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
- The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.
- If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.
- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

```
Exception in thread "xxx" Name of Exception : Description  
... ..... . // Call Stack
```

Look at the below diagram to understand the flow of the call stack.

Lab Test

09.12.24

complete Java book → Herbert Schildt

Java, C++, Python → learn with depth

OOP

109.12.24

Theory

Threading: Multitasking in Java

Multiple programs run simultaneously

CPU task scheduling

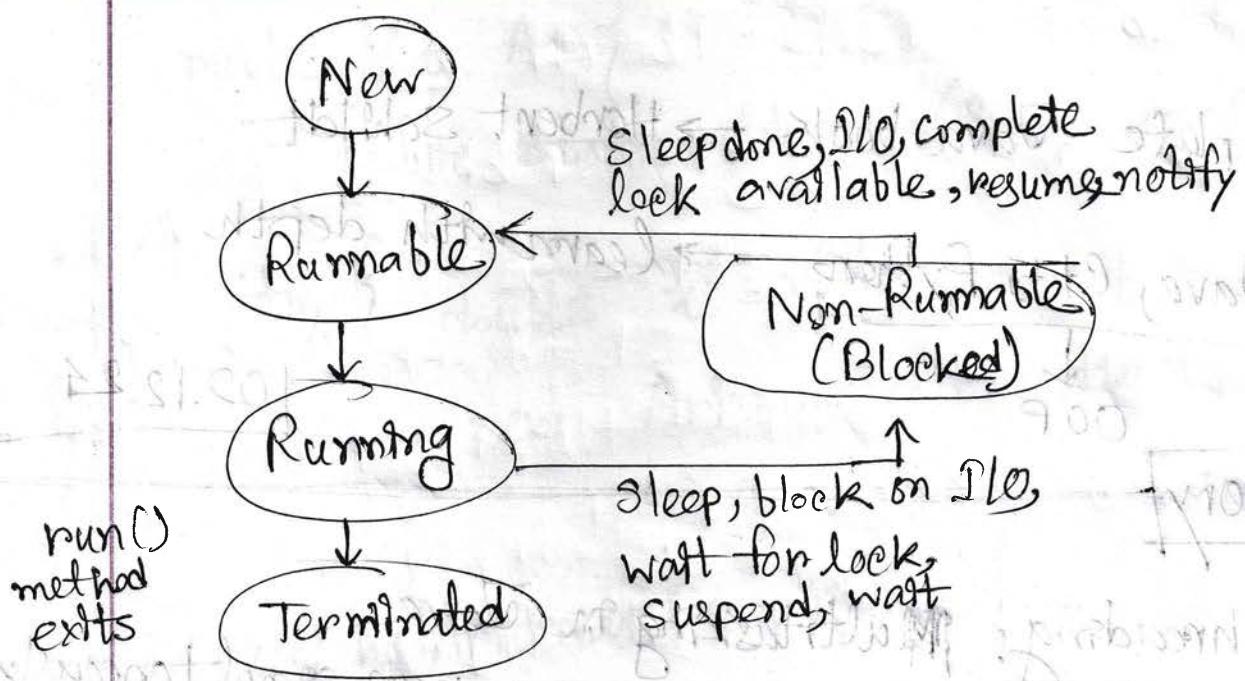
in OS

Run, Runnable, Terminated, . . .

Thread Concept in Java

A Thread is a very light-weighted process

or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.



- 1) New: Ready to run
→ A Thread is in New when it gets CPU time.
- 2) Running: A Thread is in Running state when it is Under execution.
- 3) Suspended: A thread is in the suspended state when it is temporarily inactive or under execution.
- 4) Blocked: A thread is in the Blocked state when it is waiting for resources.

Terminate:

- 5) A Thread come in terminate state when at any given time, it halts its execution immediately.

Q. Thread definition?

Q. Thread model?

Q. How to create thread? Example?

Q creating threads: A thread is created either

by "creating" or "implementing" the runnable interface or by extending the thread class.

single thread
 (Create method) → there's also implement method

```

public class ThreadExample1 extends Thread {
  public void run() {
    int a = 10;
    int b = 12;
    int result = a + b;
    System.out.println("Thread started running");
    System.out.println("Sum of the Numbers" + result);
  }
}
  
```

overriding Thread class method
 Thread started

```

public static void main(String args[1]) {
  ThreadExample1 t1 = new ThreadExample1();
  t1.start();
}
  
```

- Other thread methods → (33)

~~class creation using array index~~

• See thread examples

• For multithread we create multiple classes and

finally a main class which calls run() each

Java Threads

[« Previous](#)[Next »](#)

Java Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Creating a Thread

There are two ways to create a thread.

It can be created by extending the `Thread` class and overriding its `run()` method:

Extend Syntax

[Get your own Java Server](#)

```
public class Main extends Thread {  
    public void run() {  
        System.out.println("This code is running in a thread");  
    }  
}
```

Another way to create a thread is to implement the `Runnable` interface:

Implement Syntax

```
public class Main implements Runnable {  
    public void run() {
```

```
        System.out.println("This code is running in a thread");
    }
}
```

Running Threads

If the class extends the `Thread` class, the thread can be run by creating an instance of the class and call its `start()` method:

Extend Example

```
public class Main extends Thread {
    public static void main(String[] args) {
        Main thread = new Main();
        thread.start();
        System.out.println("This code is outside of the thread");
    }
    public void run() {
        System.out.println("This code is running in a thread");
    }
}
```

[Try it Yourself »](#)

If the class implements the `Runnable` interface, the thread can be run by passing an instance of the class to a `Thread` object's constructor and then calling the thread's `start()` method:

Implement Example

```
public class Main implements Runnable {
    public static void main(String[] args) {
        Main obj = new Main();
        Thread thread = new Thread(obj);
        thread.start();
        System.out.println("This code is outside of the thread");
    }
}
```

```
public void run() {  
    System.out.println("This code is running in a thread");  
}  
}
```

Try it Yourself »

Differences between "extending" and "implementing" Threads

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class `MyClass extends OtherClass implements Runnable`.

Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

Example

A code example where the value of the variable **amount** is unpredictable:

```
public class Main extends Thread {  
    public static int amount = 0;  
  
    public static void main(String[] args) {  
        Main thread = new Main();  
        thread.start();  
        System.out.println(amount);  
        amount++;  
        System.out.println(amount);  
    }  
  
    public void run() {  
        amount++;  
    }  
}
```

```
    }  
}
```

[Try it Yourself »](#)

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the `isAlive()` method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

Example

Use `isAlive()` to prevent concurrency problems:

```
public class Main extends Thread {  
    public static int amount = 0;  
  
    public static void main(String[] args) {  
        Main thread = new Main();  
        thread.start();  
        // Wait for the thread to finish  
        while(thread.isAlive()) {  
            System.out.println("Waiting...");  
        }  
        // Update amount and print its value  
        System.out.println("Main: " + amount);  
        amount++;  
        System.out.println("Main: " + amount);  
    }  
    public void run() {  
        amount++;  
    }  
}
```

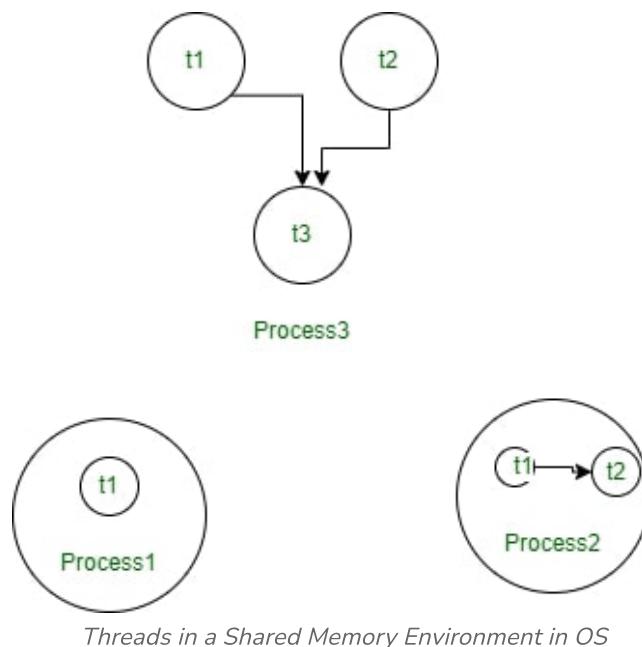
[Try it Yourself »](#)

Java Threads

Last Updated : 13 Aug, 2024



Typically, we can define threads as a subprocess with lightweight with the smallest unit of processes and also has separate paths of execution. The main advantage of multiple threads is efficiency (allowing multiple things at the same time). For example, in MS Word. one thread automatically formats the document while another thread is taking user input. Another advantage is quick response, if we use multiple threads in a process and if a thread gets stuck due to lack of resources or an exception, the other threads can continue to execution, allowing the process (which represents an application) to continue to be responsive.



As we can observe in, the above diagram a thread runs inside the process and there will be context-based switching between threads there can be multiple processes running in OS, and each process again can have multiple threads running simultaneously. The Multithreading concept is popularly applied in games, animation...etc.

The Concept Of Multitasking

To help users Operating System accommodates users the privilege of multitasking, where users can perform multiple actions simultaneously on the machine. This

Multitasking can be enabled in two ways:

- 1. Process-Based Multitasking**
- 2. Thread-Based Multitasking**

1. Process-Based Multitasking (Multiprocessing)

In this type of Multitasking, processes are heavyweight and each process was allocated by a separate memory area. And as the process is heavyweight the cost of communication between processes is high and it takes a long time for switching between processes as it involves actions such as loading, saving in registers, updating maps, lists, etc.

2. Thread-Based Multitasking

As we discussed above Threads are provided with lightweight nature and share the same address space, and the cost of communication between threads is also low.

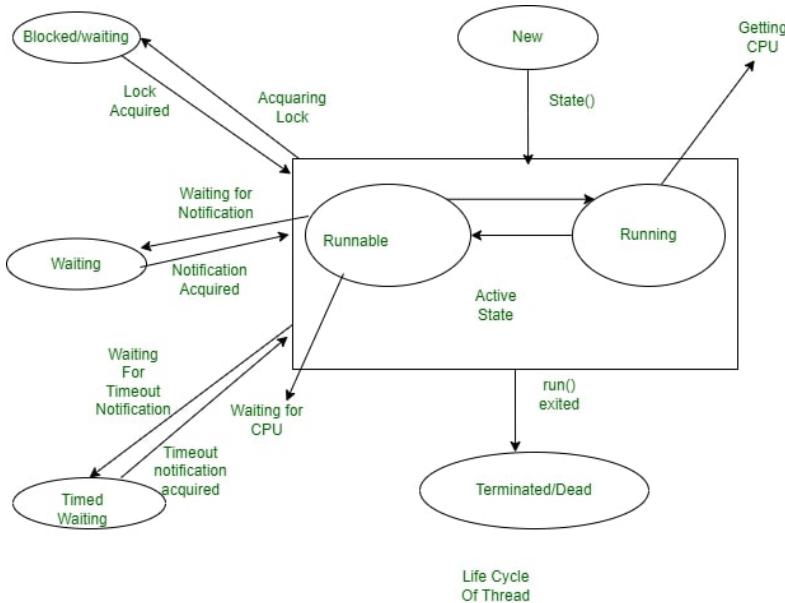
Why Threads are used?

Now, we can understand why threads are being used as they had the advantage of being lightweight and can provide communication between multiple threads at a Low Cost contributing to effective multi-tasking within a shared memory environment.

Life Cycle Of Thread

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State



We can see the working of different states in a Thread in the above Diagram, let us know in detail each and every state:

1. New State

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is not yet initiated.

2. Active State

A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states namely:

- **Runnable State:** In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.
- **Running State:** When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the “Runnable” state to the “Running” state. and after the expiry of its given time slice session, it again moves back to the “Runnable” state and waits for its next time slice.

3. Waiting/Blocked State

If a Thread is inactive but on a temporary time, then either it is a waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 needs to communicate to the camera and the other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

4. Timed Waiting State

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing a Critical Coding operation and if it does not exist the CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

5. Terminated State

A thread will be in Terminated State, due to the below reasons:

- Termination is achieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions...etc. and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longer available.

What is Main Thread?

As we are familiar, we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM, Similarly in this Multithreading Concept, Each Program has one Main Thread which was provided

by default by JVM, hence whenever a program is being created in java, JVM provides the Main Thread for its Execution.

How to Create Threads using Java Programming Language?

We can create Threads in java using two ways, namely :

1. Extending Thread Class
2. Implementing a Runnable interface

1. By Extending Thread Class

We can run Threads in Java by using Thread Class, which provides constructors and methods for creating and performing operations on a Thread, which extends a Thread class that can implement Runnable Interface. We use the following constructors for creating the Thread:

- Thread
- Thread(Runnable r)
- Thread(String name)
- Thread(Runnable r, String name)

Sample code to create Threads by Extending Thread Class:

The screenshot shows a Java code editor with the following features:

- A red "Java" tab at the top left.
- Two circular icons on the left: a copy icon and a run icon.
- Line numbers on the left side of the code area.
- Code syntax highlighting in purple, orange, green, and red.
- The code itself is as follows:

```
1 import java.io.*;
2 import java.util.*;
3
4 public class GFG extends Thread {
5     // initiated run method for Thread
6     public void run()
7     {
8         System.out.println("Thread Started
9             Running...");10    }
11    public static void main(String[] args)
12    {
13        GFG g1 = new GFG();
```

```
13  
14         // Invoking Thread using start() method  
15         g1.start();  
16     }  
17 }
```

Output

Thread Started Running...

Sample code to create Thread by using Runnable Interface:

Java



```
1 import java.io.*;  
2 import java.util.*;  
3  
4 public class GFG implements Runnable {  
5     // method to start Thread  
6     public void run()  
7     {  
8         System.out.println(  
9             "Thread is Running Successfully");  
10    }  
11  
12    public static void main(String[] args)  
13    {  
14        GFG g1 = new GFG();  
15        // initializing Thread Object  
16        Thread t1 = new Thread(g1);  
17        t1.start();  
18    }  
19 }
```

Output

Thread is Running Successfully

Sample Code to create Thread in Java using Thread(String name):

Java

```
1 import java.io.*;
2 import java.util.*;
3
4 public class GFG {
5     public static void main(String args[])
6     {
7         // Thread object created
8         // and initiated with data
9         Thread t = new Thread("Hello Geeks!");
10
11        // Thread gets started
12        t.start();
13
14        // getting data of
15        // Thread through String
16        String s = t.getName();
17        System.out.println(s);
18    }
19 }
```

Output

Hello Geeks!

Sample Java Code which creates Thread Object by using Thread(Runnable r, String name):

Java

```
1 import java.io.*;
2 import java.util.*;
3
4 public class GFG implements Runnable {
5     public void run()
```

```
6      {
7          System.out.println(
8              "Thread is created and running
9              successfully...");}
10     }
11     public static void main(String[] args)
12     {
13         // aligning GFG Class with
14         // Runnable interface
15         Runnable r1 = new GFG();
16         Thread t1 = new Thread(r1, "My Thread");
17         // Thread object started
18         t1.start();
19         // getting the Thread
20         // with String Method
21         String str = t1.getName();
22         System.out.println(str);
23     }
}
```

Output

```
My Thread
Thread is created and running successfully...
```

Java Program to explore different Thread States:

Let us see the working of thread states by implementing them on Threads t1 and t2.

Output:

Java

```
1 import java.io.*;
2 import java.util.*;
3
4 class GFG implements Runnable {
5     public void run()
6     {
```

```

7          // implementing try-catch Block to set sleep
state
8          // for inactive thread
9          try {
10             Thread.sleep(102);
11         }
12         catch (InterruptedException i1) {
13             i1.printStackTrace();
14         }
15         System.out.println(
16             "The state for t1 after it invoked join
method() on thread t2"
17             + " " + ThreadState.t1.getState());
18
19         // implementing try-catch block
20         try {
21             Thread.sleep(202);
22         }
23         catch (InterruptedException i2) {
24             i2.printStackTrace();
25         }
26     }
27 }
28
29 // creation of ThreadState class
30 // to implement Runnable interface
31 public class ThreadState implements Runnable {
32     public static Thread t1;
33     public static ThreadState o1;
34     public static void main(String args[])
35     {
36         o1 = new ThreadState();
37         t1 = new Thread(o1);
38         System.out.println("post-spanning, state of t1
is"
39                         + " " + t1.getState());
40         // lets invoke start() method on t1
41         t1.start();
42         // Now, Thread t1 is moved to runnable state
43         System.out.println(
44             "post invoking of start() method, state of
t1 is"

```

```

45             + " " + t1.getState());
46     }
47     public void run()
48     {
49         GFG g1 = new GFG();
50         Thread t2 = new Thread(g1);
51         // Thread is created and its in new state.
52         t2.start();
53         // Now t2 is moved to runnable state
54         System.out.println(
55             "state of t2 Thread, post-calling of
56 start() method is"
57             + " " + t2.getState());
58         // create a try-catch block to set t1 in
59         // waiting
60         // state
61         try {
62             Thread.sleep(202);
63         }
64         catch (InterruptedException i2) {
65             i2.printStackTrace();
66         }
67         System.out.println(
68             "State of Thread t2 after invoking to
69 method sleep() is"
70             + " " + t2.getState());
71         try {
72             t2.join();
73             System.out.println(
74                 "State of Thread t2 after join() is"
75                 + " " + t2.getState());
76         }
77         catch (InterruptedException i3) {
78             i3.printStackTrace();
79         }
80         System.out.println(
81             "state of Thread t1 after completing the
82 execution is"
83             + " " + t1.getState());
84     }
85 }
```

Output

```
post-spanning, state of t1 is NEW
post invoking of start() method, state of t1 is RUNNABLE
state of t2 Thread, post-calling of start() method is RUNNABLE
The state for t1 after it invoked join method() on thread t2
TIMED_WAITING
State of Thread t2 after invoking to method sleep() is TIMED_WAITING
State of Thread t2 after join() is TERMINATED
state of Thread t1 after completing the execution is RUNNABLE
```

OOP

Lab

Lab

02.09.24

class name = file name
 (String[] args) or (String args[1])
 ↓
 array

doesn't matter where
 it's written, it needs to be
 inside the ()

Output

Print syntax:
 Text → System.out.println("Hello World");
 Number → System.out.println(500);

Input syntax:

① import java.util.Scanner;
 public class Main {
 public static void main (String[] args) {

importing Scanner
class

② Scanner myObj = new Scanner (System.in);
 string username;
 username = myObj.nextLine();

creating new
object

③ String username = myObj.nextLine();

declaring variable

reading user input

and putting

④ System.out.println(username);

output the
read variable

Page No.

Variable Typecasting

- ① Widening Casting: small \rightarrow large type
 $\text{byte} < \text{short} < \text{char} < \text{int} < \text{long} < \text{float} < \text{double}$
- ② Narrowing Casting: large \rightarrow small type
 $\text{double} > \dots > \text{byte}$

Syntax:

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

Automatic
casting

① int myInt = 9;

double myDouble = myInt;

System.out.println(myInt);

System.out.println(myDouble);

System.out.println((int)myDouble);

Manual
casting

② double myDouble = 9.78;

int myInt = (int)myDouble;

System.out.println(myDouble);

System.out.println((int)myDouble);

System.out.println((int)myDouble);

Temporary variable

{ }
Setting break

Additional break

Left to right

Additional break

Lab

09.09.24

Primitive datatypes

Mid term

→ 14 October

1:30h

1. Data type

2. Operators

3. Array Statement: Switch, If else

4. Contract Statement

System, why SIS in capital is related to inheritance

float a = 20.5f;

without f, error shown
incompatible types: possible lossy conversion
from double to float.

Method Overloading same name, different parameters

int myMethod (int x)

float myMethod (float x)

double myMethod (double x, double y)

Method scope block of ~~code~~ refers to all of the code between curly braces {}.

variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in

which the variable was declared.

overloading scopes:

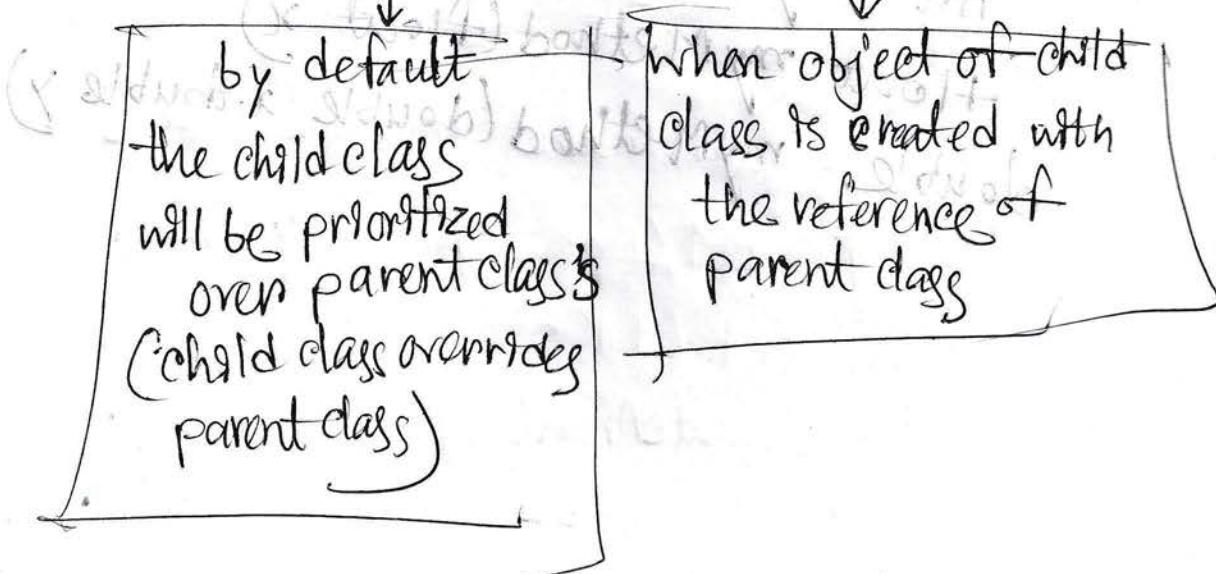
- Inside same class block:

objects from the main class determine which method to call [based on signature].

- Inside same package block:

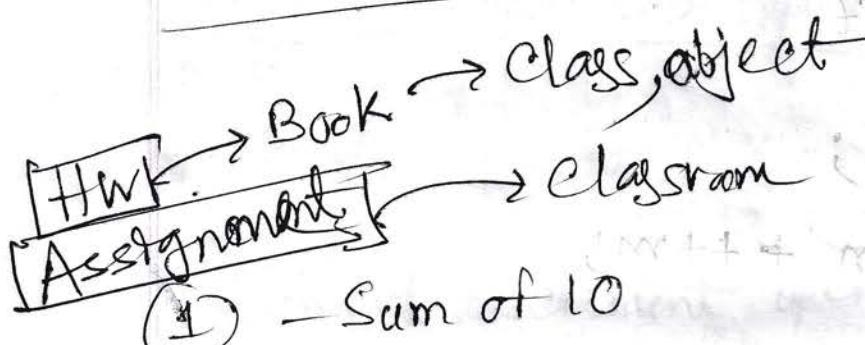
objects decide the method to invoke

[based on inheritance and upcasting]



1. Conditional Statements

Iteration: For Loop

2. Classes and objects

Java Stream → Largest

→ Java utility package (using Java stream)

import java.util.Arrays;

```
int arr[] = { ... };
```

int max = Arrays.stream(arr).max().
getAsInt();

Output(max)

class

P130 (Lab code)

class Box { double width; double height;
double depth; }

Box() { System.out.println("constructing
Box");
width = 10; height = 10; depth = 10; }

double volume() { return width * height * depth; }

}

class BoxDemo6 {

public static void main(String[] args) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

vol = mybox1.volume();

System.out.println("Volume is " + vol);

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

Lab 04 -

C++

Class and Object (Similar concept as Java)

Encapsulation

Syntax: Class Employee {

 private:

 int salary;

 we can access
 this
 using this

 public:

 void setSalary (int s) {
 salary = s; }

 int getSalary () {

 return salary; }

 };

 int main () { Employee myObj;

 myObj.setSalary (50000);

 cout << myObj.getSalary();

 return 0;

}

Lab - 05

25

Chapter 7 - methods and classes

- Assignment must be in A4 format with cover page
- 21 October Lab mid after OOP theory exam,
Inheritance 1:00 - 2:00 pm

- ① This keyword
- ② Overloading method
- ③ " constructor
- ④ Parameter passing

This

specifying the current class when same others
exist. It is a reference ~~variable~~ ^{variable} ~~keyword~~.

overloading

multiple methods with same name.

with different parameters (format
or number of variables)

Lab 06 - Scanner / User Input

outside classes:

import java.util.Scanner;

or
import java.util.*;

(Package)

(Class)

Scanners
method list.

inside main method:

Scanner s1 = new Scanner(System.in);

inputs as
[String] and
type casts
as integer
variable
declared

int x = s1.nextInt(); → type casting
int y = s1.nextInt(); → predefined
in util package

String s = s1.nextLine();
for string

concatenates
∴ using Scanner.nextInt() method

- stores
- typecasts
- adds newline

1. Constructor Overloading

2. User Input

3. User Input + Constructor Overload

4. this keyword

Practice Book problems 8.16, 8.17

Lab 07: Inheritance

Super and Sub object

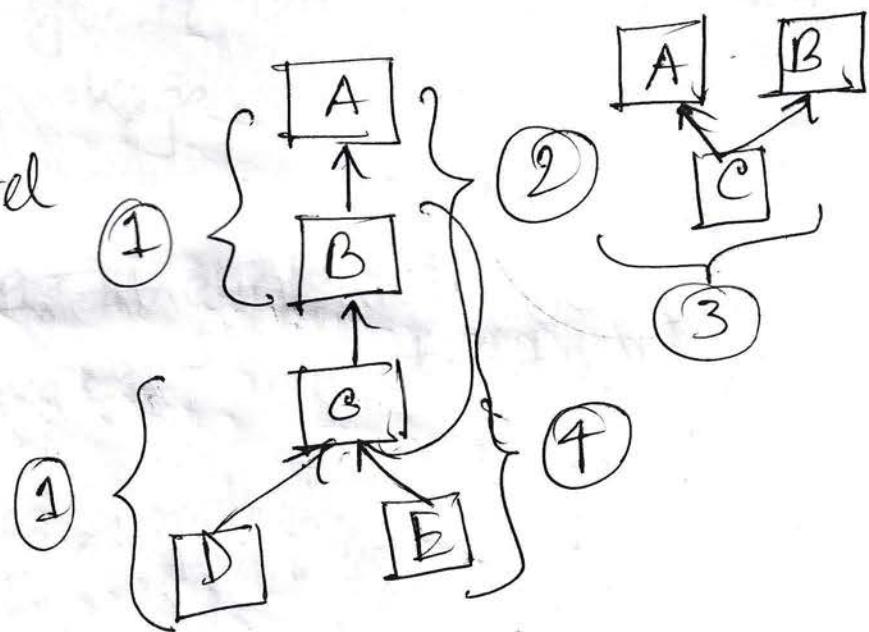
child class can't access private properties of parent class.

types ① single level

② multi level

③ multiple

④ hybrid



Class Diagram: (geeksforgeeks) → UML

Practice Book problems 8.16, 8.17

Lab 07: Inheritance

Super and Sub object

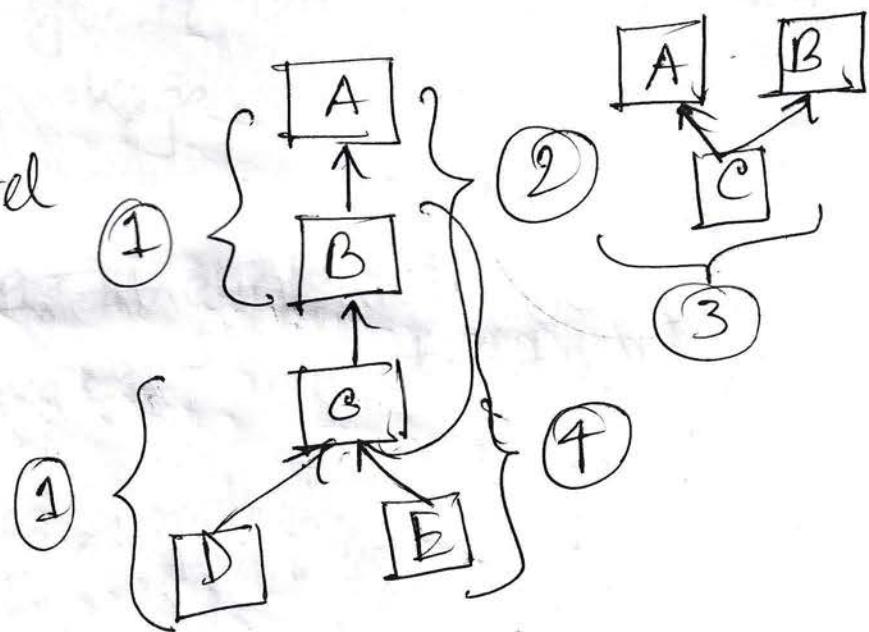
child class can't access private properties of parent class.

types ① single level

② multi level

③ multiple

④ hybrid



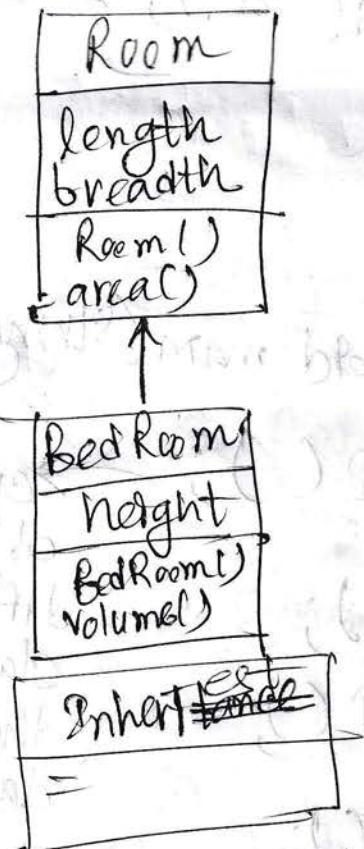
Class Diagram: (geeksforgeeks) → UML

[Lab]-08:

11.16.24

~~super(x,y); accesses the parent class variables x and y. Since x,y is not class variable of the child class but the parent class. Therefore without using super keyword we can't use them(2).~~

Q.



P-142

Book : Balaguru Samy
P-139
Program - 8.5
Herbert Schmidt
① P-159
② P-(160-161)
③ P-(167-168)
multilevel
Hierarchical

(2)

- ② → constructors under parent class not used
- ③ → using super keyword we are calling Superclass constructors. Super (w,h,d,m);

cc "constructing clone of an object"

CT ~~as
next class~~

on inheritance

parent class constructors

- through inheritance parent class constructors are invoked when objects are created under child class.

(?)

Project on inheritance

Parent {

Parent()

extends Parent

child {

child()

main {

child c = new child();

Parent class constructors becomes the ↑ through inheritance same child class constructor

public class A1 {

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

try { double x = 3/0; } // error

throw new ArithmeticException();

catch(ArithmeticException e){

//public static void main(String args[])

// throws D0Exception{}

(Throws)

makes try-catch block unnecessary as throws will output default exception messages regardless of try-catch blocks written.

Lab - 09: Interface, multiple inheritance [18.11.24]

Q. Java doesn't allow Multiple Inheritance

normally?

→ Needs concepts like Interface to work.

- Abstract class and Interface ~~doesn't have~~ objects. They have variables and methods.

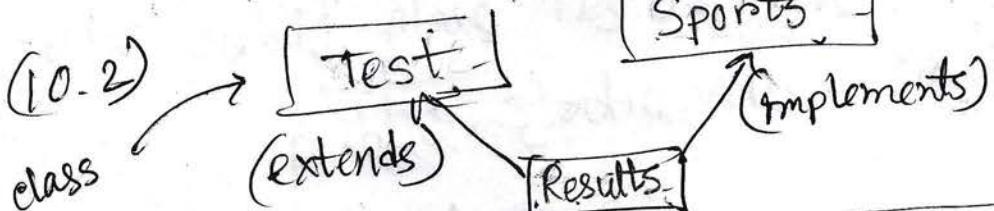
Program 10.1 (P184) } (3E primers)

Program 10.2

- final keyword: voids overriding

// final static float pi = 3.14F;

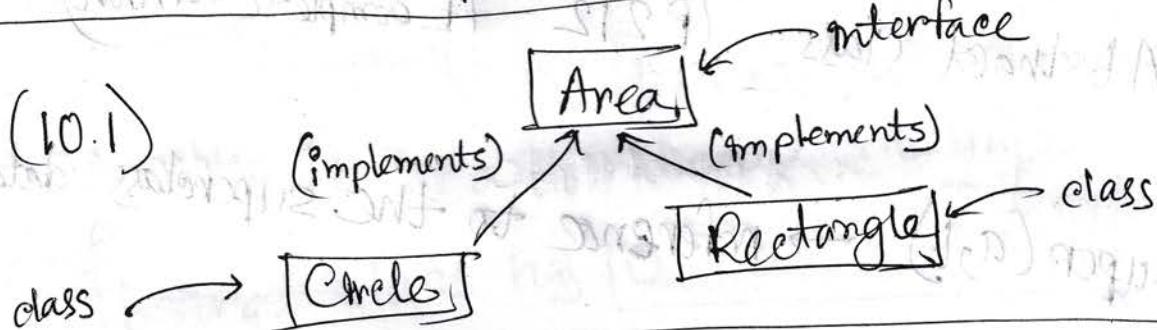
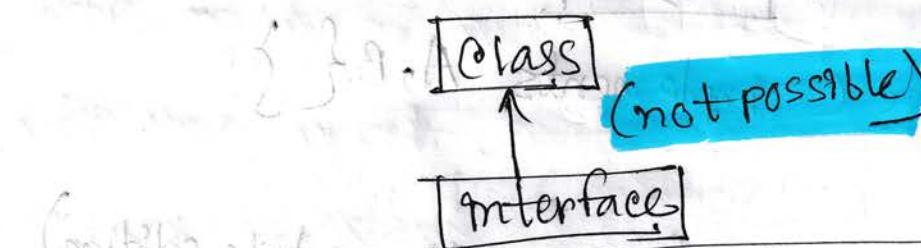
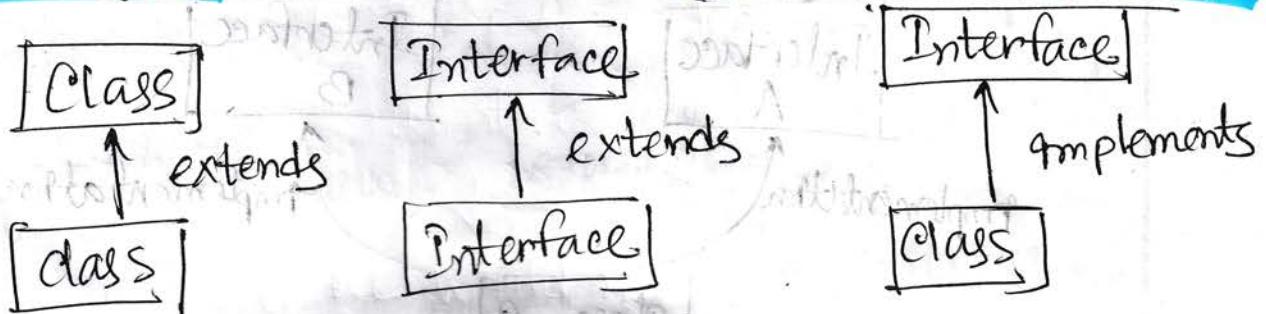
can use together?



for, class → extends same usage:
for, interface → implements inherits parent
extension and implementation class/interface

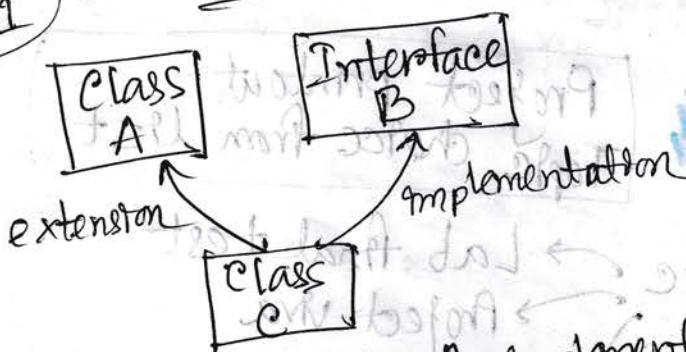
various forms of Interface implementation:

Interface is incomplete, can't inherit classes



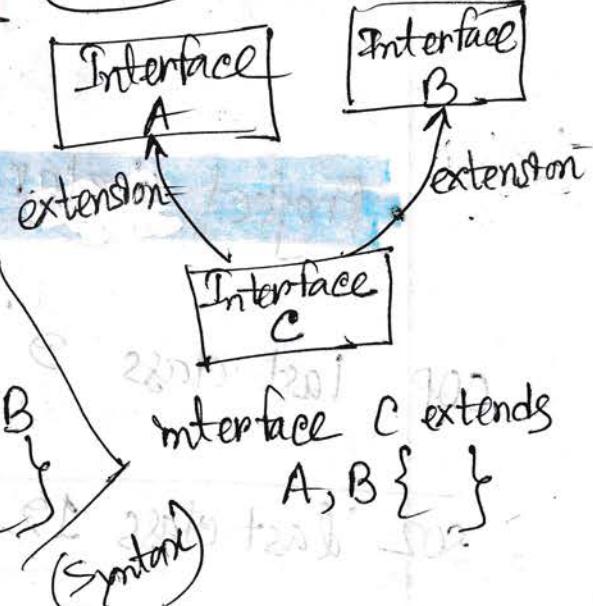
Multiple Inheritance:

Case-1



class C extends A implements B

Case-2

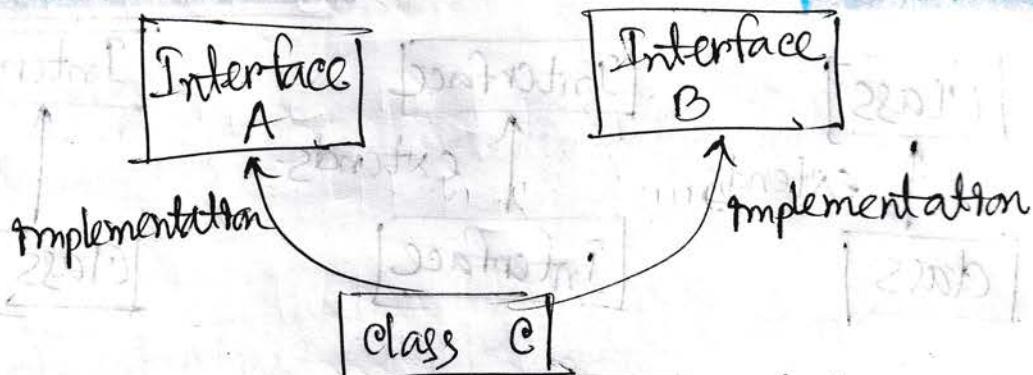


interface C extends A, B { }

(Syntax)

(Syntax)

Case-3



class C implements {A, B}

(Syntax)

#Abstract class:

(P212 - 7E complete edition)

(2.)

super(a,b) → reference to the superclass data

Project declare:

Project printout
Topic choice from list

OOP

Last class 9 dec

→ Lab final test

→ Project viva

→ Project report

→ Source code
Project team detail
Input/Output

S02 Last class 19 dec

Lab - 10

25.11.24

Project?

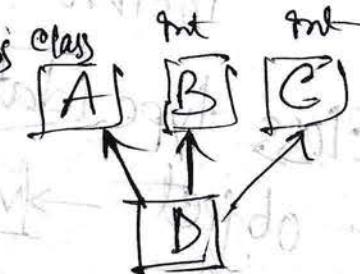
- 9 Dec, Lab Test + Project
- Lab Report on Project
 - ↓
Cover Page, Project
- comments beside code
- Must run on Lab computers

Java 18, NetBeans 8.2

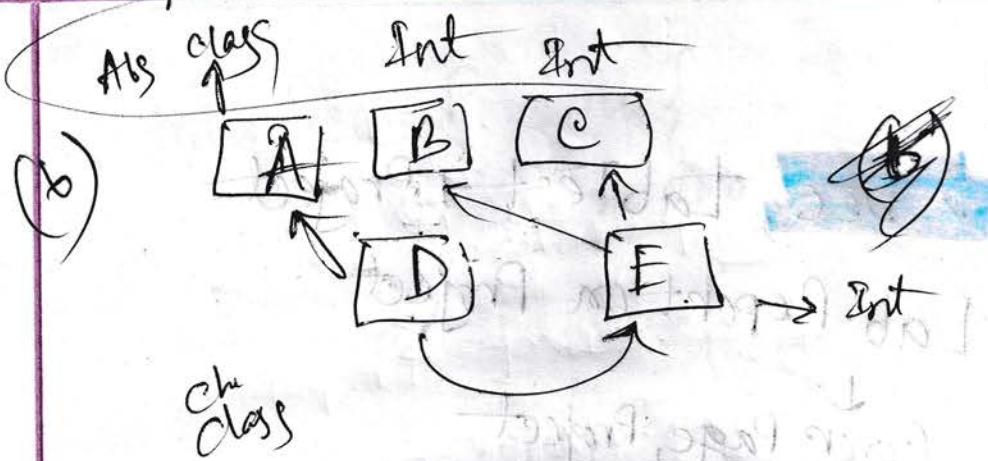
- No Presentation via GUI
- GUI not necessary
- Code with knowledge so far

Lab Test)

- (a) Abstract Class - 1 } Inherited by
 Interface ————— 2 } Class - 1
 Multiple Inheritance

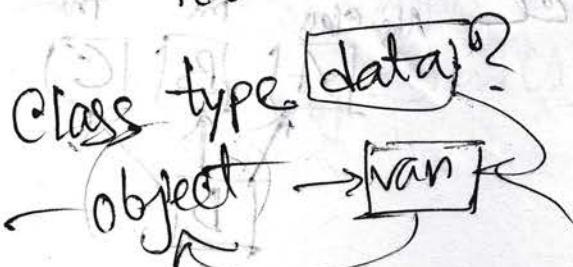


hybrid inheritance:



Java Swing:

- # Topics:
- Polymorphism
 - Overriding
 - Access modifier
 - AWT / Swing
 - ↓ Abstract window Toolkit
- overloading
final (when not overriding)
linked with Inheritance
private, protected, public, default



Data and Methods in OOP - [GPT]

- **fields (Attributes)**: Variables that hold data for an object. int, float, string
- **Methods**: Functions defined inside a class that operate on the objects of the class.
- **Constructors**: Special methods used to initialize objects.
- **Access modifiers**: Keywords that determine the visibility of fields and methods.
[private, protected, public]

Table:

| | private | protected | public | default |
|--------------|--------------|-----------|--------|---------|
| (2) location | class | ✓ | ✓ | ✓ |
| | static class | ✗ | ✓ | ✓ |
| | package | ✗ | ✓ | ✓ |
| | outside | ✗ | ✗ | ✓ |
| (2) type | constructor | ✓ | ✓ | ✓ |
| | variable | ✓ | ✓ | ✓ |
| | method | ✓ | ✓ | ✓ |
| | class | ✗ | ✗ | ✓ |

~~QUESTION~~ #Java Swing:

- Must use Library files

import java.awt.*;

import java.awt.event.*;

import java.awt.Container;

- Event?

- min, max, close built in created

- dob (date of birth) array

Next

→ Exception handling

Try and catch?

Q10

Project Briefs

- Projects, can overlap topics
- Must use inheritance concepts
- GUI (optional) → J Swing
- store data using array.
- One Report for each group.

Lab 11

02.12.24

* Next class → final Labtest + Project Submission

#Exception Handling:

compile time error → Syntax error

runtime error → logical error

Book: Chapter 13 (Primer), Chapter 10 (Herbert)

• Project must include Exception handling

~~Book~~

throw

Notice: 9.12.24

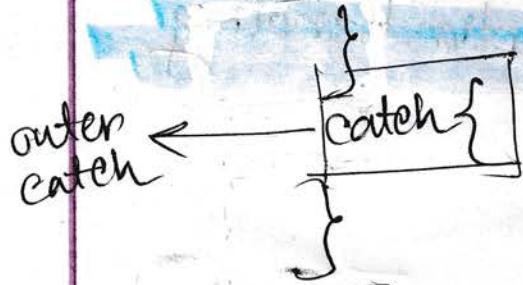
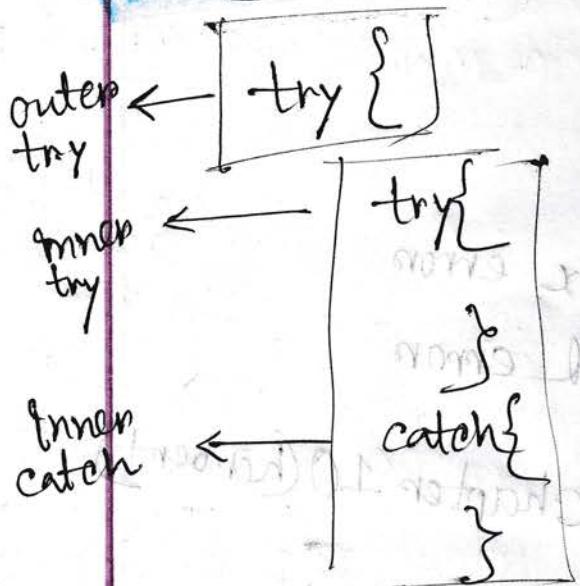
1. final Lab Tgt (8.00 - 9.00)

2. Project submission, VIVA (9.00 - 10.00)

Project Report submission

Painted

Nested try-catch:



• `throw[new Runtime Exception ("demo")];`

Intentionally/manually triggers/
creates exception in catch

creates new
Runtime Exception object

throws (literally)
object. Causing
the corresponding

the new created exception
the program to jump to
catch block

misc.

C

#define is a preprocessor directive that is used to define macros. The macros are identifiers defined by #define which are replaced by their value before compilation. We can define constants and functions like macros using #define. The generic in C are also implemented using the #define preprocessor directive.

along with Generic (?)