

# **Fundamentals of Object Oriented Programming**

Dr. Ayesha Hakim

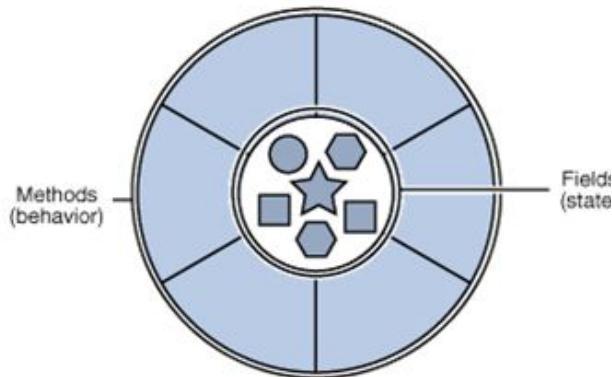
# MODULE 2

2	<b>Class, Object, Method and Constructor</b>			
	<b>2.1</b>	Class Object and Method: member, method, Modifier, Selector, iterator, State of an object. Memory allocation of object using new operator, Command line Arguments. instanceof operator in Java.		
	<b>2.2</b>	Method overloading & overriding, constructor, destructor in C++, Types of constructor (Default, Parameterized, copy constructor with object), Constructor overloading, this, final, super keyword, Garbage collection in Java.	08	CO 1, CO 2

# Objects

---

- **object:** An entity that encapsulates data and behavior.
  - *data:* variables inside the object
  - *behavior:* methods inside the object
    - You interact with the methods; the data is hidden in the object.



- Constructing (creating) an object:

```
Type objectName = new Type (parameters);
```

- Calling an object's method:

```
objectName . methodName ( parameters );
```

# Objects in Java

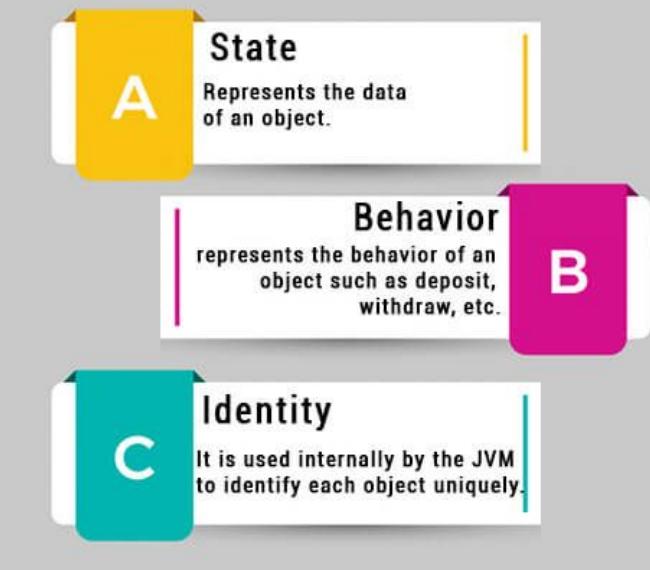
An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use **new()**.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

## Characteristics of Object



# Classes

---

- **class:** A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.
- **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.
  - **abstraction:** Separation between concepts and details.  
Objects and classes provide abstraction in programming.

# ACCESSING THE DATA MEMBERS

- The public data members of objects of a class can be accessed using the direct member access operator (.)

```
#include <iostream>
using namespace std;
class Box{
public:
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
};

int main(){
    Box Box1; // Declare Box1 of type Box
    Box Box2; // Declare Box2 of type Box
    double volume = 0.0; // Store the
    volume of a box here
    // box 1 specification Box1.height = 5.0;
    Box1.length = 6.0;
    Box1.breadth = 7.0;
    // box 2 specification
    Box2.height = 10.0;
    Box2.length = 12.0;
    Box2.breadth = 13.0; //
    volume of //box 1
    volume = Box1.height *
    Box1.length * Box1.breadth;
    cout << "Volume of Box1 : " <<
    volume << endl; // volume of
    box 2
    volume = Box2.height *
    Box2.length * Box2.breadth;
    cout << "Volume of Box2 : " <<
    volume << endl; return 0;}
```

## MEMBERS OF CLASS

- A member function of a class is a function that has its definition or its prototype within the class
- Members are accessed using dot operator(.)

```
class Box
{
public:
    double length;          // Length of a box
    double breadth;         // Breadth of a box
    double height;          // Height of a box
    double getVolume(void); // Returns box volume
    // member function
};
```

data member

## MEMBERS OF CLASS(CONTINUED...)

- Member functions can be defined within the class definition or separately using **scope resolution operator**, ::

```
class Box{  
public:  
    double length;    // Length of a box  
    double breadth;   // Breadth of a box  
    double height;    // Height of a box  
    double getVolume(void){  
        return length * breadth * height;  
    }  
};// class end
```

## MEMBERS OF CLASS(CONTINUED...)

- If you like you can define same function outside the class using **scope resolution operator**, :: as follows:

```
double Box::getVolume(void){  
    return length * breadth * height;  
}
```

# Classes ARE Object Definitions

- OOP - object oriented programming
- code built from objects
- In Java these are called **classes**
- Each class definition is coded in a separate .java file
- Name of the object must match the class/object name

# Classes and Objects

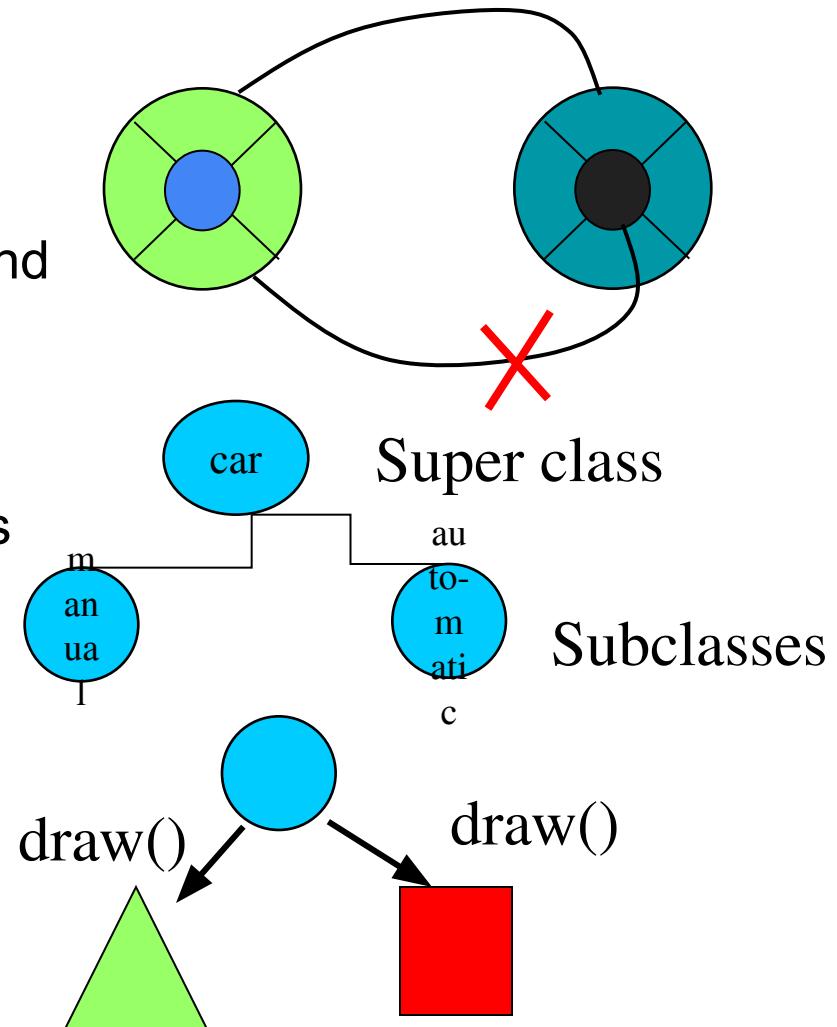
- The **class** is the unit of programming
- A Java program is a **collection of classes**
  - Each class definition (usually) in its own `.java` file
  - *The file name must match the class name*
- A class describes **objects (instances)**
  - Describes their common characteristics: is a *blueprint*
  - Thus all the instances have these same characteristics
- These characteristics are:
  - **Data fields** for each object
  - **Methods** (operations) that do work on the objects

# Grouping Classes: The Java API

- API = *Application Programming Interface*
- Java = small core + extensive collection of packages
- A **package** consists of some related Java classes:
  - Swing: a GUI (graphical user interface) package
  - AWT: Application Window Toolkit (more GUI)
  - util: utility data structures
- The **import** statement tells the compiler to make available classes and methods of another package
- A **main** method indicates where to begin executing a class (if it is designed to be run as a program)

# The three principles of OOP

- Encapsulation
  - Objects hide their functions (**methods**) and data (**instance variables**)
- Inheritance
  - Each **subclass** inherits all variables of its **superclass**
- Polymorphism
  - Interface same despite different data types



# Simple Class and Method

```
Class Fruit{  
    int grams;  
    int cals_per_gram;  
  
    int total_calories() {  
        return(grams*cals_per_gram);  
    }  
}
```

# Methods

- A method is a named sequence of code that can be invoked by other Java code.
- A method takes some parameters, performs some computations and then optionally returns a value (or object).
- Methods can be used as part of an expression statement.

```
public float convertCelsius(float tempC) {  
    return( ((tempC * 9.0f) / 5.0f) + 32.0 );  
}
```

# Method Signatures

- A method signature specifies:
    - The name of the method.
    - The type and name of each parameter.
    - The type of the value (or object) returned by the method.
    - The checked exceptions thrown by the method.
    - Various method modifiers.
    - *modifiers type name ( parameter list ) [throws exceptions ]*
- public float convertCelsius (float tCelsius) {}
- public boolean setUserInfo ( int i, int j, String name ) throws  
IndexOutOfBoundsException {}

# Public/private

- Methods/data may be declared ***public*** or ***private*** meaning they may or may not be accessed by code in other classes ...
- Good practice:
  - keep data private
  - keep most methods private
- well-defined interface between classes - helps to eliminate errors

# Using objects

- Here, code in one class creates an instance of another class and does something with it ...

```
Fruit plum=new Fruit();
int cals;
cals = plum.total_calories();
```

- ***Dot operator*** allows you to access (public) data/methods inside Fruit class

## instanceof operator

- **instanceof** is an operator that determines if an object is an instance of a specified class:

```
Book b1 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
System.out.println(b1 instanceof Book);
```

True

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

# Java Iterator

An `Iterator` is an object that can be used to loop through collections, like `ArrayList` and `HashSet`. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the `java.util` package.

The `iterator()` method can be used to get an `Iterator` for any collection.

## Syntax

```
Iterator itr = c.iterator();
```

*Note:* Here "c" is any Collection object. `itr` is of type `Iterator` interface and refers to "c".

## Methods of Iterator Interface in Java

The iterator interface defines **three** methods as listed below:

- 1. hasNext():** Returns true if the iteration has more elements.

```
public boolean hasNext();
```

- 2. next():** Returns the next element in the iteration. It throws **NoSuchElementException** if no more element is present.

```
public Object next();
```

- 3. remove():** Removes the next element in the iteration. This method can be called only once per call to next().

```
public void remove();
```

# Selector

*selector*

(or *method selector*) a word (e.g., `draw`) that performs an operation on a variety classes. A selector describes *what* operation to perform.

*selector invocation*

a call of a selector. One argument of the call is used for determining which method is used. i.e., a message (consisting of the selector and the other arguments) is sent to the object.

*receiving object*

the object used for determining the method executed by a selector invocation.

# Access Modifiers in Java

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Access Modifier	within class	within package	outside package by subclass only	outside package
<b>Private</b>	Y	N	N	N
<b>Default</b>	Y	Y	N	N
<b>Protected</b>	Y	Y	Y	N
<b>Public</b>	Y	Y	Y	Y

The `public` keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- Access Modifiers - controls the access level
- Non-Access Modifiers - do not control access level, but provides other functionality

For **classes**, you can use either `public` or `default`:

Modifier	Description
<code>public</code>	The class is accessible by any other class
<code>default</code>	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the <a href="#">Packages chapter</a>

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
<code>public</code>	The code is accessible for all classes
<code>private</code>	The code is only accessible within the declared class

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
<code>public</code>	The code is accessible for all classes
<code>private</code>	The code is only accessible within the declared class
<code>default</code>	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the <a href="#">Packages chapter</a>
<code>protected</code>	The code is accessible in the same package and <b>subclasses</b> . You will learn more about subclasses and superclasses in the <a href="#">Inheritance chapter</a>

# Non-Access Modifiers

For **classes**, you can use either **final** or **abstract**:

Modifier	Description
<b>final</b>	The class cannot be inherited by other classes (You will learn more about inheritance in the <a href="#">Inheritance chapter</a> )
<b>abstract</b>	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the <a href="#">Inheritance</a> and <a href="#">Abstraction</a> chapters)

For **attributes and methods**, you can use the one of the following:

Modifier	Description
<code>final</code>	Attributes and methods cannot be overridden/modified
<code>static</code>	Attributes and methods belongs to the class, rather than an object
<code>abstract</code>	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example <b>abstract void run();</b> . The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the <u>Inheritance</u> and <u>Abstraction</u> chapters
<code>transient</code>	Attributes and methods are skipped when serializing the object containing them
<code>synchronized</code>	Methods can only be accessed by one thread at a time
<code>volatile</code>	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

# The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The **super** reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

# The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the **super** reference to call the parent's constructor
- The **super** reference can also be used to reference other variables and methods defined in the parent's class

## Using super

### Using super

**super** has two general forms.

- ✓ The first calls the superclass constructor.
- ✓ The second is used to access a member of the superclass that has been hidden by a member of a subclass

# Inheritance

## Using super

### A first Use for super

```
class a
{
    a(){System.out.println("A");}
}
class b extends a
{
    b()
    {
        super();
        System.out.println("B");
    }
}
class c extends b
{
    c(){System.out.println("c");}
}
class last
{
    public static void main(String args[])
    {
        c obj = new c();
    }
}
```

# Inheritance

## Using super

### A Second Use for super

The second form of **super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.** This usage has the following general form:

`super.member`

Here, *member* can be either a method or an instance variable.

**Most applicable to situations in which member names of a subclass hide members by the same name in the superclass.** Consider this simple class hierarchy:

# Inheritance

## Using super

### A Second Use for super

```
// Using super to overcome name hiding.  
class A {  
    int i;  
}  
// Create a subclass by extending class A.  
class B extends A {  
    int i; // this i hides the i in A  
    B(int a, int b) {  
        super.i = a; // i in A  
        i = b; // i in B  
    }  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}
```

```
class UseSuper {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2);  
        subOb.show();  
    }  
}
```

This program displays the following:

i in superclass: 1  
i in subclass: 2

# **The super keyword in Java**

- super keyword is similar to this keyword in Java.
- It is used to refer to the immediate parent class of the object.

## **Use of super keyword in Java**

- super () calls the parent class constructor with no argument.
- super.methodname calls method from parents class.
- It is used to call the parents class variable.

### Example: Sample program for invoking parents class constructor

```
class Animal
{
    public Animal(String str)
    {
        System.out.println("Constructor of Animal: " + str);
    }
}
class Lion extends Animal
{
    public Lion()
    {
        super("super call Lion constructor");
        System.out.println("Constructer of Lion.");
    }
}
public class Test
{
    public static void main(String[] a)
    {
        Lion lion = new Lion();
    }
}
```

#### Output:

Constructor of Animal: super call from Lion constructor  
Constructor of Lion.

### Example: Sample program for calling parents class variable

```
class Base
{
    int a = 50;
}

public class Derive extends Base
{
    int a = 100;
    void display()
    {
        System.out.println(super.a);
        System.out.println(a);
    }
    public static void main(String[] args)
    {
        Derive derive = new Derive();
        derive.display();
    }
}
```

#### Output:

50

100

# this keyword

- The keyword “this” is used in an instance method to refer to the object that contains the method, i.e. the it refers to the current object.
- Whenever and wherever, a reference to an object of the current class type is required, ‘this’ can be used.
- It has two other usages:
  - Differentiating between instance variables and local variables
  - Constructor chaining

Example codes: <https://www.javatpoint.com/this-keyword>

# this keyword (contd.)

- To differentiate between Instance variables and local variables

```
Room2( double l, double b, double h) {  
    this.length = l;  
    this.breadth = b;  
    this.height = h;  
}
```

# Constructor chaining

- It means a constructor can be called from another constructor.

```
/*First Const.*/ Room2()
{
    // constructor chained
    this(14,12,10);
}
/*Second Const.*/ Room2( double l, double b, double h) {
    length = l;
    breadth = b;
    height = h;
}
```

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using **free() function in C language** and **delete() in C++**. But, in java it is performed automatically. So, java provides better memory management.

# Cleaning up unused object

- Java allows a programmer to create as many objects as he/she wants but frees him/her from worrying about destroying them. The Java runtime environment deletes objects when it determines that they are no longer required.
- The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer needed.
- Before an object gets garbage collected, the garbage collector gives the object an opportunity to clean up itself through a call to the object's finalize() method. This process is known as finalization.

<https://www.javatpoint.com/Garbage-Collection>

# Garbage Collector

- Two basic approaches :
  - *Reference counting* and
  - *tracing*.
- Reference counting maintains a reference count for every object.
- It is incremented and decremented as and when the references on objects increase or decrease (leave an object)
- When reference count for a particular object is 0, the object can be garbage collected.

# Garbage Collector (contd.)

- Tracing technique traces the entire set of objects (starting from root) and all objects having reference on them are marked in some way.
- a.k.a *mark and sweep* garbage
- The objects that are not marked (not referenced) are assumed to be garbage and their memory is reclaimed.

# Garbage Collector (contd.)

- Mark and sweep collectors further use the techniques of Compaction and Copying for fragmentation problems that may arise once you sweep the unreferenced objects.
- Compaction moves all the live objects towards one end making the other end a large free space and copying techniques copies all live objects besides each other into a new space and the old space is considered free now.

# Advantages

- Free From worrying about deallocation of memory
- Helps in ensuring integrity of programs.
- No way by which Java programmers can knowingly or unknowingly free memory incorrectly.
- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

# Disadvantages

- overhead to keep track of which objects are being referenced by the executing program and which are not being referenced
- overhead is also incurred on finalization and freeing memory of the unreferenced objects.
- These activities will incur more CPU time than would have been incurred if the programmers would have explicitly deallocated memory.

# How can an object be unreferenced?

01

By nulling the reference

02

By assigning a reference to  
another

03

By anonymous object etc.

## 1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

## 2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

## 3) By anonymous object:

```
new Employee();
```

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize()
```



Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc()
```

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```



Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

## Simple Example of garbage collection in java

```
public class TestGarbage1{  
    public void finalize(){System.out.println("object is garbage collected");}  
    public static void main(String args[]){  
        TestGarbage1 s1=new TestGarbage1();  
        TestGarbage1 s2=new TestGarbage1();  
        s1=null;  
        s2=null;  
        System.gc();  
    }  
}
```

**Test it Now**

object is garbage collected  
object is garbage collected

# The final keyword in Java

The **final** keyword in Java indicates that no further modification is possible. Final can be Variable, Method or Class

## Final Variable

Final variable is a constant. We cannot change the value of final variable after initialization.

Example: Sample program for final keyword in Java

```
class FinalVarDemo
{
    final int a = 10;
    void show()
    {
        a = 20;
        System.out.println("a : "+a);
    }
    public static void main(String args[])
    {
        FinalVarDemo var = new FinalVarDemo();
        var.show();
    }
}
```

**Output:**  
Compile time error

# The Final Modifier (Inheritance)

- Methods preceded by the final modifier cannot be overridden
  - e.g.,    public *final* void displayTwo ()
- Classes preceded by the final modifier cannot be extended
  - e.g., *final* public class ParentFoo

# Final method

When we declare any method as final, it cannot be overridden.

**Example:** Sample program for final method in Java

```
class Animal
{
    final void eat()
    {
        System.out.println("Animals are eating");
    }
}
public class Dear extends Animal
{
    void eat()
    {
        System.out.println("Dear is eating");
    }
    public static void main(String args[])
    {
        Dear dear = new Dear();
        dear.eat();
    }
}
```

**Output:**  
Compile time error

## Final Class

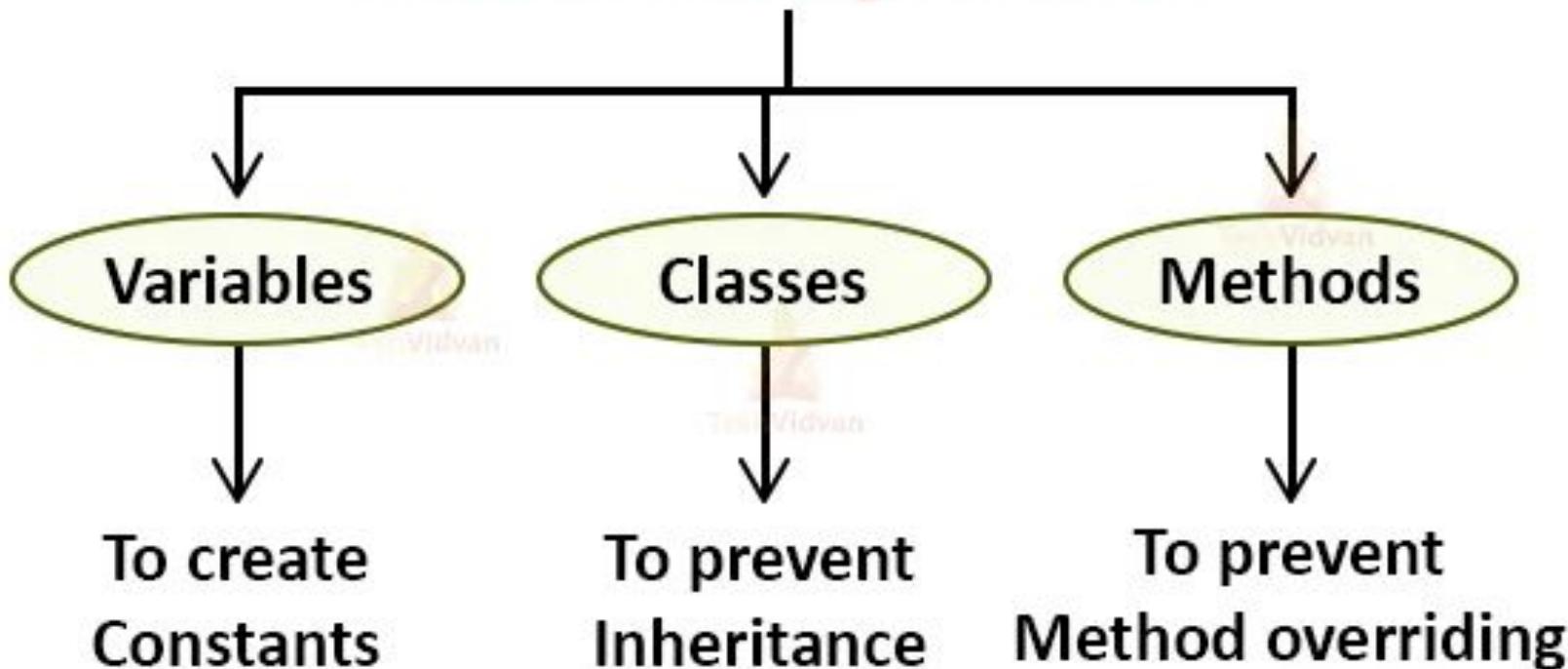
When a class is declared as a final, it cannot be extended.

**Example:** Sample program for final class in Java

```
final class Animal
{
    void eat()
    {
        System.out.println("Animals are eating");
    }
}
class Dear extends Animal
{
    void eat()
    {
        System.out.println("Dear is eating");
    }
    public static void main(String args[])
    {
        Dear dear = new Dear();
        dear.eat();
    }
}
```

**Output:**  
Compile time error

# Final Keyword



<https://www.javatpoint.com/final-keyword>

## Characteristics of final keyword in java:

In Java, the final keyword is used to indicate that a variable, method, or class cannot be modified or extended. Here are some of its characteristics:

- **Final variables:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.
- **Final methods:** When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class's public API and should not be modified by subclasses.
- **Final classes:** When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.
- **Initialization:** Final variables must be initialized either at the time of declaration or in the constructor of the class. This ensures that the value of the variable is set and cannot be changed.
- **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- **Security:** final can help improve security by preventing malicious code from modifying sensitive data or behavior.

# Practise

- Java Program to Add Two Integers (entered by user)
- Java Program to Swap Two Numbers
- Java Program to Check Whether a Number is Even or Odd
- Java Program to Check Whether an Alphabet is Vowel or Consonant
- Java Program to Find Factorial of a Number
- Java Program to Check Whether a Number is Prime or Not
- Java Program to Find Factorial of a Number Using Recursion

\$ Java Cheat Sheet

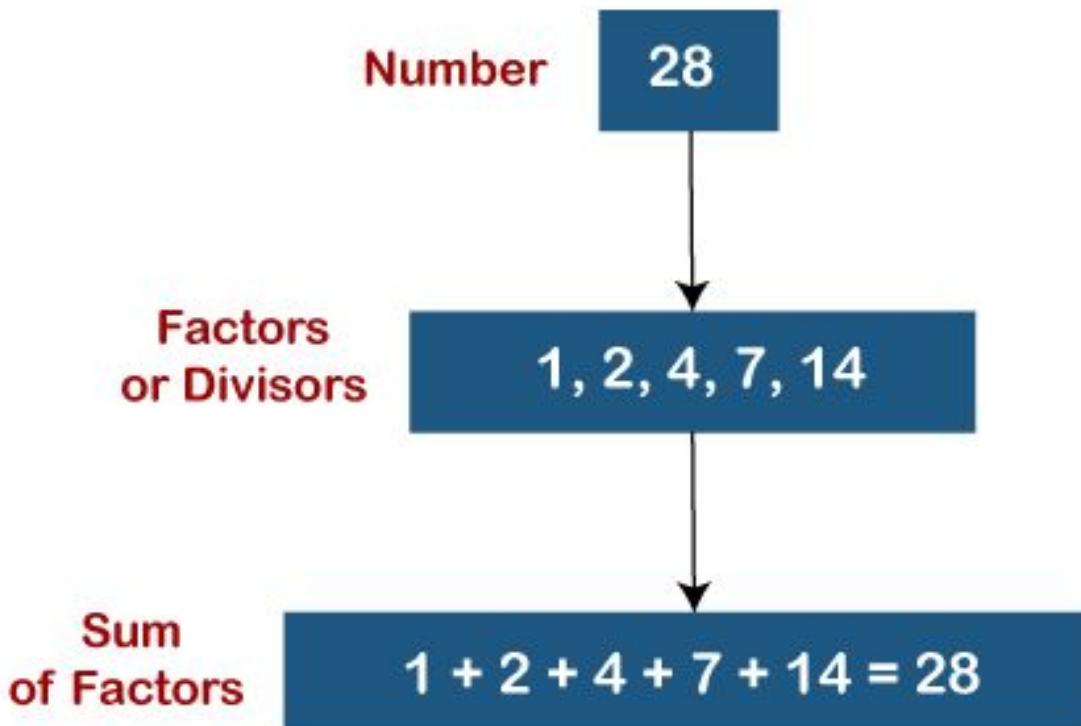
# Finding Perfect Number

- A number whose sum of factors (excluding the number itself) is equal to the number is called a perfect number. In other words, if the sum of positive divisors (excluding the number itself) of a number equals the number itself is called a perfect number.

## EXAMPLE:

- Let's take the number 496 and check it is a perfect number or not.
- First, we find the factors of 496 i.e. 1, 2, 4, 8, 16, 31, 62, 124, and 248. Let's find the sum of factors ( $1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248 = 496$ ).
- We observe that the sum of factors is equal to the number itself. Hence, the number 496 is a perfect number.

## Perfect Number in Java



Sum of Factors = Number

28 is a perfect Number

Perfect Number	Positive Factors	Sum of all factors excluding itself
6	1, 2, 3, 6	6
28	1, 2, 4, 7, 14, 28	28
496	1, 2, 4, 8, 16, 31, 62, 124, 248, 496	496
8,128	1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064, 8128	8,128

$$6 = 2^1(2^2 - 1) = 1 + 2 + 3,$$

$$28 = 2^2(2^3 - 1) = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 1^3 + 3^3,$$

$$\begin{aligned} 496 &= 2^4(2^5 - 1) = 1 + 2 + 3 + \cdots + 29 + 30 + 31 \\ &= 1^3 + 3^3 + 5^3 + 7^3, \end{aligned}$$

$$\begin{aligned} 8128 &= 2^6(2^7 - 1) = 1 + 2 + 3 + \cdots + 125 + 126 + 127 \\ &= 1^3 + 3^3 + 5^3 + 7^3 + 9^3 + 11^3 + 13^3 + 15^3, \end{aligned}$$

$$\begin{aligned} 33550336 &= 2^{12}(2^{13} - 1) = 1 + 2 + 3 + \cdots + 8189 + 8190 + 8191 \\ &= 1^3 + 3^3 + 5^3 + \cdots + 123^3 + 125^3 + 127^3. \end{aligned}$$

# Steps to Find Perfect Number

- Read or initialize a number (n).
- Declare a variable (s) for storing sum.
- Find the factors of the given number (n) by using a loop (for/ while).
- Calculate the sum of factors and store it in a variable s.
- Compare the sum (s) with the number (n):
- If both (s and n) are equal, then the given number is a perfect number.
- Else, the number is not a perfect number.
- **Implement the above steps in a Java program to find Perfect Numbers Between Given Range**

## **Algorithm to Find Perfect Number in Java**

- Insert an integer from the user.
- Set the initial value of a different variable to 0 to store the sum of the appropriate positive divisors.
- Check each number from 1 to  $n/2$  to see if it is a divisor. Count the sum of all the divisors. If the sum is exactly  $n$ , the number is a perfect number; otherwise, it is not.

## **Program to Find Perfect Number in Java**

There are three ways to find the perfect number in Java:

- Using for Loop
- Using while Loop
- Using recursion in Java

<https://www.javatpoint.com/perfect-number-program-in-java>

# Finding Armstrong Number

An Armstrong number, also known as a narcissistic number, is a number that is the sum of its own digits when each digit is raised to the power of the number of digits in the number.

Ex. **371** is an Armstrong number because  $3^3 + 7^3 + 1^3 = 371$

# Algorithm for Armstrong Number in Java

1. Define a function isArmstrongNumber() that takes an integer num as input and returns a boolean value.
2. Declare a variable originalNum and set it equal to num.
3. Convert num to a string and find the length of the string. This will give us the number of digits in num.
4. Declare a variable sum and set it to 0. This will be used to store the sum of the digits raised to the power of the number of digits.
5. Begin a loop that will iterate as long as num is greater than 0.
6. Inside the loop, find the last digit of num by using the modulus operator (%) to get the remainder when num is divided by 10.
7. Raise the digit to the power of the number of digits, and add the result to sum.
8. Divide num by 10 to remove the last digit.
9. End the loop.
10. After the loop, check if originalNum is equal to sum. If it is, return true. Otherwise, return false.

## Pseudo-code Algorithm

```
FUNCTION isArmstrongNumber(num: INTEGER) -> BOOLEAN:  
    originalNum = num  
    numOfDigits = LENGTH(num AS STRING)  
    sum = 0  
    WHILE num > 0:  
        digit = num MOD 10  
        sum = sum + digit^numOfDigits  
        num = num DIV 10  
    END WHILE  
    RETURN originalNum == sum  
END FUNCTION
```

Here are a few examples of Armstrong numbers:

- 153 ( $1^3 + 5^3 + 3^3 = 153$ )
- 371 ( $3^3 + 7^3 + 1^3 = 371$ )
- 9474 ( $9^4 + 4^4 + 7^4 + 4^4 = 9474$ )

## CODE:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Armstrong numbers between 1 and  
1000:");  
  
        for (int number = 1; number <= 1000; number++) {  
            if (isArmstrong(number)) {  
                System.out.println(number);  
            }  
        }  
    }  
}
```

```
public static boolean isArmstrong(int number) {  
    int originalNumber = number;  
    int sum = 0;  
    int digits = String.valueOf(number).length();  
  
    while (number != 0) {  
        int digit = number % 10;  
        sum += Math.pow(digit, digits);  
        number /= 10;  
    }  
  
    return originalNumber == sum;  
}
```

# OUTPUT

Output

Clear

```
java -cp /tmp/bH5v7Ear0S/Main
Armstrong numbers between 1 and 1000:
1
2
3
4
5
6
7
8
9
153
370
371
407
```

==== Code Execution Successful ===

# Static Method

```
import java.io.*;  
  
public class StaticExample {  
  
    static int num = 100;  
    static String str = "you can call a static method without  
creating an object of the class";  
  
    // This is Static method  
    static void display()  
    {  
        System.out.println("static number is " + num);  
        System.out.println("static string is " + str);  
    }  
}
```

## **// non-static method**

```
void nonstatic()
{
    // our static method can accessed in non static method
    display();
}
```

## **// main method**

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

    // This is object to call non static function
    obj.nonstatic();

    // static method can called directly without an object
    display();
}
```

<https://www.geeksforgeeks.org/difference-between-static-and-non-static-method-in-java/>

# Output

Output

Clear

```
▲ java -cp /tmp/qQIh0HaeFM StaticExample
      static number is 100
      static string is you can call a static method without creating an
          object of the class
      static number is 100static string is you can call a static method
          without creating an object of the class
```

# **WAP for displaying Armstrong numbers in a given range, use a non static method**

- 1. Create a class** with a non-static method to check if a number is an Armstrong number.
- 2. Use this method** to find and display Armstrong numbers within a given range.

```
public class ArmstrongNumberFinder {  
  
    // Non-static method to check if a number is an Armstrong  
    number  
    public boolean isArmstrongNumber(int number) {  
        int originalNumber = number;  
        int sum = 0;  
        int digits = String.valueOf(number).length();  
  
        while (number > 0) {  
            int digit = number % 10;  
            sum += Math.pow(digit, digits);  
            number /= 10;  
        }  
  
        return sum == originalNumber;  
    }  
}
```

```
// Method to display Armstrong numbers in a given range

    public void displayArmstrongNumbers(int startRange, int
endRange) {
        System.out.println("Armstrong numbers between " +
startRange + " and " + endRange + " are:");
        for (int num = startRange; num <= endRange; num++)
{
            if (isArmstrongNumber(num)) {
                System.out.println(num);
            }
        }
    }
}
```

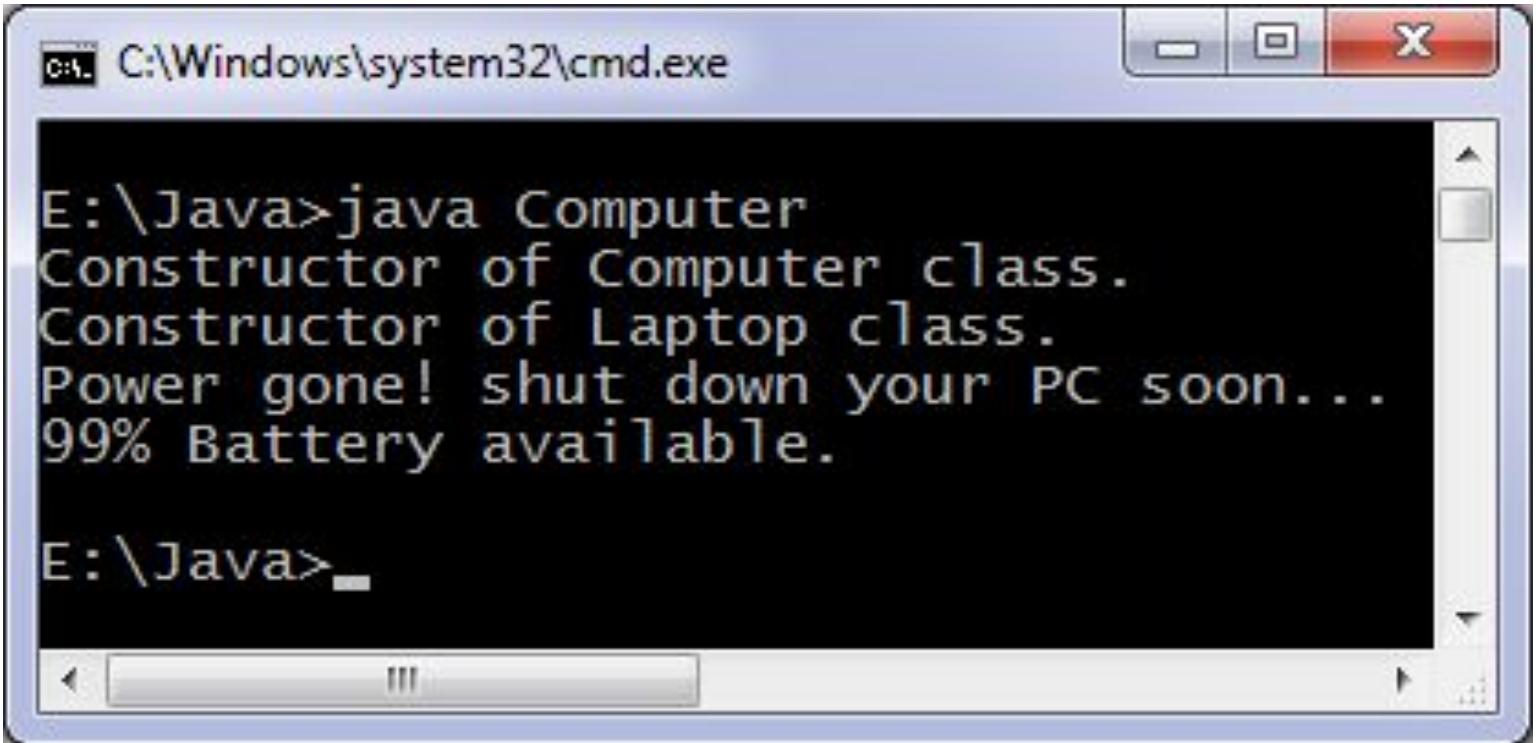
```
// Main method to run the program
public static void main(String[] args) {
    ArmstrongNumberFinder finder = new
ArmstrongNumberFinder();
    int startRange = 1;
    int endRange = 10000;

    // You can modify startRange and endRange as
needed
    finder.displayArmstrongNumbers(startRange,
endRange);
}
}
```

# Using two classes in Java program

```
class Computer {  
    Computer() {  
        System.out.println("Constructor of Computer class.");  
    }  
  
    void computer_method() {  
        System.out.println("Power gone! Shut down your PC soon...");  
    }  
  
    public static void main(String[] args) {  
        Computer my = new Computer();  
        Laptop your = new Laptop();  
  
        my.computer_method();  
        your.laptop_method();  
    }  
}  
  
class Laptop {  
    Laptop() {  
        System.out.println("Constructor of Laptop class.");  
    }  
  
    void laptop_method() {  
        System.out.println("99% Battery available.");  
    }  
}
```

# OUTPUT



A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window contains the following text output:

```
E:\Java>java Computer
Constructor of Computer class.
Constructor of Laptop class.
Power gone! shut down your PC soon...
99% Battery available.

E:\Java>
```

You can also create objects in a method of Laptop class. When you compile the above program, two .class files are created, which are Computer.class and Laptop.class. Its advantage is you can reuse your .class file somewhere in other projects without recompiling the code.

In a nutshell, **number of .class files created are equal to the number of classes in the program**. You can create as many classes as you want, but writing many classes in a single file isn't recommended, as it makes code difficult to read.

Instead, you can create a separate file for every class. You can also group classes in packages for efficiently managing the development of your application.

# Kaprekar Numbers

**Write a Java program to generate and show all Kaprekar numbers less than 1000.**

In number theory, a Kaprekar number for a given base is a non-negative integer, the representation of whose square in that base can be split into two parts that add up to the original number again.

Ex. 45 is a Kaprekar number, because  $45^2 = 2025$  and  $20 + 25 = 45$ .

```
1 public class KaprekarNumbers {  
2  
3     public static void main(String[] args) {  
4         System.out.println("Kaprekar numbers less than 1000 are:");  
5  
6         for (int i = 1; i < 1000; i++) {  
7             // Calculate the square of the number  
8             int square = i * i;  
9  
10            // Convert square to string for easier manipulation  
11            String squareStr = Integer.toString(square);  
12  
13            // Number of digits in the original number  
14            int numDigits = Integer.toString(i).length();  
15  
16            // Determine the length of the right part  
17            int rightLength = numDigits;  
18            int leftLength = squareStr.length() - rightLength;  
19  
20            // Extract left and right parts  
21            String rightPartStr = squareStr.substring(squareStr.length() - rightLength);  
22            String leftPartStr = leftLength > 0 ? squareStr.substring(0, leftLength) : "0";  
23  
24            // Convert parts back to integers  
25            int rightPart = Integer.parseInt(rightPartStr);  
26            int leftPart = Integer.parseInt(leftPartStr);  
27  
28            // Check if the sum of the two parts equals the original number  
29            if (leftPart + rightPart == i) {  
30                System.out.println(i);  
31            }  
32        }  
33    }  
34}  
35}
```

## Output

Clear

```
java -cp /tmp/Mo0iUQCtbF/KaprekarNumbers
```

```
Kaprekar numbers less than 1000 are:
```

```
1  
9  
45  
55  
99  
297  
703  
999
```

```
==== Code Execution Successful ===
```

1. **Loop through numbers:** Iterate from 1 to 999.
2. **Square the number:** Compute ``i * i`` for each number.
3. **Convert to string:** Convert the square of the number to a string to easily split it.
4. **Calculate lengths:** Determine the lengths of the right and left parts.
5. **Extract parts:** Use ``substring()`` to extract the right and left parts of the squared number as strings and then convert them back to integers.
6. **Check Kaprekar condition:** Compare the sum of the extracted parts to the original number.
7. **Print Kaprekar numbers:** If the condition is met, print the number.



# Constructors

- The line  
`plum = new Fruit();`
- invokes a constructor method with which you can set the initial data of an object
- You may choose different types of constructor with different argument lists  
eg `Fruit()`, `Fruit(a)` ...

Run example programs from: <https://www.javatpoint.com/java-constructor>

# Difference between constructor and method in Java

A constructor is used to Initialize the state of an object.

A constructor must not have a return type.

The constructor Is Invoked Implicitly.

The Java compiler provides a default constructor If you don't have any constructor in a class.

The constructor name must be same as the class name.

A method Is used to expose the behavior of an object.

A method must have a return type.

The method Is Invoked explicitly.

The method Is not provided by the compiler in any case.

The method name may or may not be same as class name.

# THE CLASS CONSTRUCTOR

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.
- **A constructor will have exact same name as the class** and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.
- **0-argument constructor**
- **Parameterized constructor**
- **Copy constructor**

# Copy Constructor with Object - Algorithm

- **Define a class:** Create a class that represents the object you want to manage.
- **Define instance variables:** Within the class, define instance variables that represent the data you want to manage.
- Define a constructor: Define a constructor for the class that takes an instance of the same class as its argument. This constructor will be used to create a copy of the object.
- **Initialize the instance variables:** Within the constructor, initialize the instance variables with the values from the argument object.
- Use the `this` keyword to refer to the instance variables: To refer to the instance variables of the class within the constructor, use the `this` keyword.
- **Check for null values:** If the argument object is null, return a new instance of the class with default values for the instance variables.
- **Implement deep copying:** If the instance variables are objects, create new instances of those objects within the constructor and initialize them with the values from the argument object. This is called deep copying and ensures that changes to the copied object do not affect the original object.

**Code:** <https://www.geeksforgeeks.org/copy-constructor-in-java/>

<https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaoqQTkrpxjKrWp1O1EU/edit?usp=sharing>

# THE CLASS DESTRUCTOR (IN C++)

- A **destructor** is a special member function of a class that is **executed whenever an object of its class goes out of scope or whenever the delete expression is applied** to a pointer to the object of that class.
- A **destructor will have exact same name as the class prefixed with a tilde (~)** and it can neither return a value nor can it take any parameters.
- Destructor can be very **useful for releasing resources** before coming out of the program like closing files, releasing memories etc.
- <https://www.geeksforgeeks.org/destructors-c/>  
(example codes to see usage of destructor)

# Overloading

- Can have several versions of a method in class with different types/numbers of arguments

```
Fruit() {grams=50;}
```

```
Fruit(a,b) { grams=a; cals_per_gram=b; }
```

- By looking at arguments Java decides which version to use

# overloading methods and its purpose

- ✓ If a class have multiple methods by same name but different parameters it is known as method overloading.
- ✓ If we have to perform only one operation , having same name of the methods increases the readability of the program.
- ✓ Different ways to overload the method:
  1. By changing number of arguments
  2. By changing data type

# Method overloading

## Method overloading with an example:

```
// Demonstrate method overloading.  
class OverloadDemo  
{  
    void test()  
    {  
        System.out.println("No parameters"); }  
        // Overload test for one integer parameter.  
    void test(int a) {  
        System.out.println("a: " + a); }  
        // Overload test for two integer parameters.  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b); }  
        // overload test for a double parameter  
    double test(double a) {  
        System.out.println("double a: " + a); return a*a; }  
    }  
  
    class Overload {  
        public static void main(String args[]) {  
            OverloadDemo ob = new OverloadDemo();  
            double result;  
            // call all versions of test()  
            ob.test();  
            ob.test(10);  
            ob.test(10, 20);  
            result = ob.test(123.25);  
            System.out.println("Result of ob.test(123.25): " +  
                result); } }
```

This program generates the following output:

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

## Method overloading

Why method overloading is not possible by changing the return type?

Can we overload main() method?

✓ Method overloading is not possible by changing the return type of method because there may occur ambiguity.

✓ Yes, we can have any number of main methods in a class by method overloading

# Constructor overloading

## What is constructor overloading ?

- ✓ Just like in case of method overloading you have multiple methods with same name but different signature , in Constructor overloading you have multiple constructor with different signature with only difference that Constructor doesn't have return type in Java.
- ✓ Those constructor will be called as **overloaded constructor**. Overloading is also another form of polymorphism in Java which allows to have multiple constructor with different name in one Class in java.

## Constructor overloading

Why do you overload Constructor?

- ❑ Allows flexibility while create array list object.
- ❑ It may be possible that you don't know size of array list during creation than you can simply use default no argument constructor but if you know size then its best to use overloaded
- ❑ Constructor which takes capacity. Since ArrayList can also be created from another Collection, may be from another List than having another overloaded constructor makes lot of sense.
- ❑ By using overloaded constructor you can convert your ArrayList into Set or any other collection.

# Constructor overloading

## Constructor overload with an example

```
/* Here, Box defines three constructors to initialize  
the dimensions of a box various ways. */  
  
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // constructor used when all dimensions specified  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d; }  
  
    // constructor used when no dimensions specified  
    Box() {  
        width = -1;      // use -1 to indicate  
        height = -1;    // an uninitialized  
        depth = -1;     // box }  
  
    // constructor used when cube is created  
    Box(double len) {  
        width = height = depth = len; }  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth; } }
```

```
class OverloadCons {  
    public static void main(String args[]) {  
        // create boxes using the various constructors  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box();  
        Box mycube = new Box(7);  
        double vol;  
  
        // get volume of first  
        box vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
        // get volume of second  
        box vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
        // get volume of  
        cube vol = mycube.volume();  
        System.out.println("Volume of mycube is " + vol);  
    } }
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0  
Volume of mybox2 is -1.0  
Volume of mycube is 343.0
```

# Recursion

- ✓ Recursion is the process of defining something in terms of itself.
- ✓ As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- ✓ A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

For example, 3 factorial is  $1 \times 2 \times 3$ , or 6. Here is how a factorial can be computed by use of a recursive method:

# Recursion with example

```
// A simple example of recursion.  
class Factorial {  
    // this is a recursive method  
    int fact(int n) {  
        int result;  
        if(n==1)  
            return 1;  
        result = fact(n-1) * n; return result; } }  
class Recursion { public static void main(String args[ ]) {  
    Factorial f = new Factorial(); System.out.println("Factorial of 3 is " +  
    f.fact(3)); System.out.println("Factorial of 4 is " + f.fact(4));  
    System.out.println("Factorial of 5 is " + f.fact(5)); } }
```

The output from this program is shown here: Factorial of 3 is 6 Factorial of 4 is 24 Factorial of 5 is 120

# Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

# Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

# Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

# Method Overriding

- Different versions of a method can be implemented in different ways by the parent and child class in an inheritance hierarchy.
- Methods have the same name and parameter list (identical signature) but different bodies

```
public class Parent
{
    :
    public void method ()
    {
        System.out.println("m1");
    }
}
```

```
public class Child extends Parent
{
    :
    public void method ()
    {
        num = 1;
    }
}
```

# Method Overloading Vs. Method Overriding

- Method Overloading

- Multiple method implementations for the same class
- Each method has the same name but the type, number or order of the parameters is different (signatures are not the same)
- The method that is actually called is determined at program *compile time* (early binding).
- i.e., <reference name>.<method name> (parameter list);



Distinguishes  
overloaded methods

## Method Overloading Vs. Method Overriding (2)

- Example of method overloading:

```
public class Foo
{
    public void display () { }
    public void display (int i) { }
    public void display (char ch) { }
}
```

```
Foo f = new Foo ();
f.display();
f.display(10);
f.display('c');
```

# Method Overloading Vs. Method Overriding (3)

- Method Overriding

- The method is implemented differently between the parent and child classes.
- Each method has the same return value, name and parameter list (identical signatures).
- The method that is actually called is determined at program ***run time (late binding)***.
- i.e., <reference name>.<method name> (parameter list);



The type of the reference  
(implicit parameter “this”)  
distinguishes overridden  
methods

# Method Overloading Vs. Method Overriding (4)

- Example of method overriding:

```
public class Foo
{
    public void display () { ... }
    :
}
public class FooChild extends Foo
{
    public void display () { ... }
}
```

```
Foo f = new Foo ();
f.display();
```

```
FooChild fc = new FooChild ();
fc.display ();
```

**No**

## Method Overloading

## Method Overriding

1)	Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

## Java Method Overloading example

```
class OverloadingExample{  
  
    static int add(int a,int b){return a+b;}  
  
    static int add(int a,int b,int c){return a+b+c;}  
  
}
```

## Java Method Overriding example

```
class Animal{  
  
    void eat(){System.out.println("eating...");}  
  
}  
  
class Dog extends Animal{  
  
    void eat(){System.out.println("eating bread...");}  
  
}
```

Q1.

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

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

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

OUTPUT ???

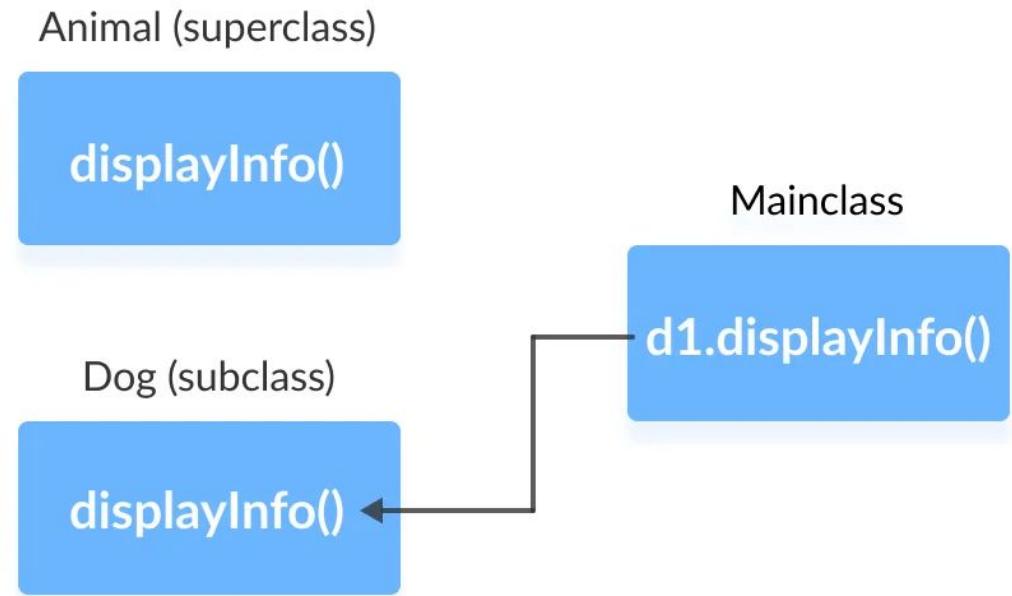
## Output:

I am a dog.

## WHY?

In the above program, the `displayInfo()` method is present in both the `Animal` superclass and the `Dog` subclass.

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



Here, the `@Override` annotation specifies the compiler that the method after this annotation overrides the method of the superclass. It is not mandatory to use `@Override`.

## **Java Overriding Rules**

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as final and static.

# Constructor Overloading

Defining two or more constructors with the **same name in a class** but with **different signatures** is called constructor overloading. We arrange them in a such a way that each constructor performs a different task.

*Java compiler* differentiates these constructors based on the number of the parameter lists and their types. Therefore, the **signature** of each constructor must be different.

The signature of a constructor consists of its name and sequence of parameter types.

If two constructors of a class have the same signature, it represents **ambiguity**. In this case, Java compiler will generate an error message because Java compiler will be unable to differentiate which form to execute.

```
public class Person
{
    Person() ←
    {
        // constructor body
    }
    Person(String name) ←
    {
        // constructor body
    }
    Person(String scName, int rollNo) ←
    {
        // constructor body
    }
    .....
    .....
}
```

Three  
overloaded  
constructors  
having a  
different  
parameter  
list

Fig: Java overloaded constructors based on parameter list

## FLOW OF EXECUTION OF PROGRAM

```
public class Employee {  
    Employee(){ ←  
        System.out.println("Employee Details:");  
    }  
    Employee(String name){ ←  
        System.out.println("Employee name: " +name);  
    }  
    Employee(String nCompany, int id){ ←  
        System.out.println("Company name: " +nCompany);  
        System.out.println("Employee id: " +id);  
    }  
}  
  
public class Myclass {  
    public static void main(String[] args) {  
        Employee emp=new Employee(); _____  
        Employee emp2=new Employee("Deep"); _____  
        Employee emp3=new Employee("HCL", 12234); _____  
    }  
}
```

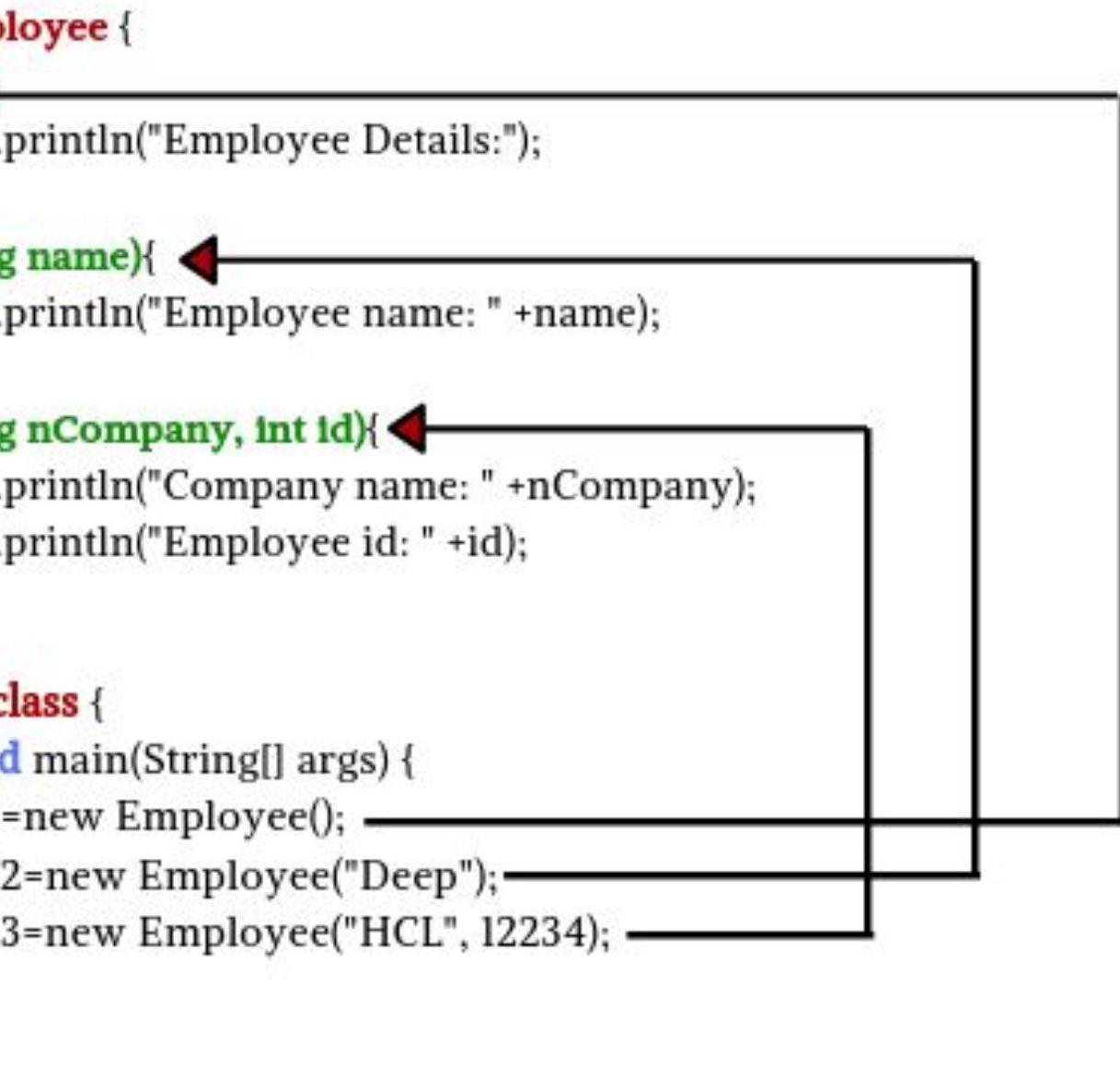


Fig: Overloaded constructors based on parameter list.

```

// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;

class construct
{
public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};

```

```

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}

```

Output:

```

0
200

```

# using PI in Java

```
import java.lang.Math.*;  
public class Pie {  
    public static void main(String[] args) {  
        //radius and length  
        double radius = 5;  
        double len = 15;  
        // calculate the area using PI  
        double area = radius * radius * Math.PI;  
        //and now volume  
        double volume = area * len;  
        System.out.println("Volume of cylinder is: " + volume);  
    }  
}
```

# CODING ACTIVITY

#1. WAJP to find area of circle and rectangle using a) method overloading, b) method overriding

#2. WAJP to demonstrate Simple Parameterized Constructor For i) Finding Prime Number or ii) finding area of rectangle

#3. WAJP to illustrate constructor overloading to print the names of students by creating a Student class. If no name is passed while creating an object of Student class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating object of Student class.

#1

```
class OverridingDemo
{
    public int length=10;
}
class Rectangle extends OverridingDemo
{
    void area()
    {
        int breadth=7;
        int area=length*breadth;
        System.out.println("The area of the rectangle is"+area+"sq
        units");
    }
}
class Circle extends OverridingDemo
{
    void area()
    {
        double area =3.14*length*length;
        System.out.println("The area of the circle is"+area+"sq units");
    }
}
class Inheritence
{
    public static void main(String args[])
    {
        Rectangle rec= new Rectangle();
        Circle cir= new Circle();
        rec.area();
        cir.area();
    }
}
```

OUTPUT  
java Inheritence  
The area of the rectangle is 70 sq units  
The area of the circle is 314.0 sq units

```
class OverloadDemo
{
    void area(float x,float y)
    {
        System.out.println("The area of the rectangle is "+x*y+"sq
        units");
    }
    void area(double x)
    {
        double z= 3.14*x*x;
        System.out.println("The area of the circle is "+z+"sq units");
    }
}
class Overload
{
    public static void main(String args[])
    {
        OverloadDemo ob= new OverloadDemo();
        ob.area(10,20);
        ob.area(2.5);
    }
}
```

OUTPUT:  
java Overload  
The area of the rectangle is 200.0sq units  
The area of the circle is 19.625sq units

## Example: How to check number is prime or not using constructor in java.

```
import java.util.Scanner;

class Test {
    int n, i, p = 1;

    Scanner sc = new Scanner(System.in);

    Test() {
        System.out.print("Enter a number:");
        n = sc.nextInt();
    }

    void checkPrime() {
        for (i = 2; i < n; i++) {
            if (n % i == 0) {
                p = 0;
                break;
            }
        }

        if (p == 1) {
            System.out.println("Number is prime:" + n);
        } else {
            System.out.println("Number is not prime:" + n);
        }
    }
}
```

```
class Main {

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

### Output:

```
Enter a number:13
Number is prime:13
```

## EXTRA EXAMPLE

## #2

```
1 class Prime{  
2     Prime(int n){  
3         int count=0;  
4         for(int i=2;i<=n;i++){  
5             if(n%i==0){  
6                 count=count+1;  
7             }  
8             if(count==0){  
9                 System.out.println("its a prime");  
10                break;  
11            }  
12            else{  
13                System.out.println("its not a prime");  
14                break;  
15            }  
16        }  
17    }  
18}  
19  
20 class main {  
21     public static void main(String[] args) {  
22         Prime p =new Prime(7);  
23     }  
24 }  
25 }  
26 }  
27 }
```

Output

```
java -cp /tmp/FUwutTEtp6 main  
its a prime
```

Clear

## Example: How to find the area of a rectangle using parameterized constructor in Java

#2

```
import java.util.Scanner;  
  
class Test {  
  
    int area;  
  
    Test(int l, int w) {  
        area = l * w;  
    }  
  
    void display() {  
        System.out.println("Area of rectangle is: " + area);  
    }  
}  
  
class Main {  
  
    public static void main(String args[]) {  
        int length, width;  
        Scanner sc = new Scanner(System.in);  
  
        System.out.print("Enter length of rectangle:");  
        length = sc.nextInt();  
        System.out.print("Enter width of rectangle:");  
        width = sc.nextInt();  
  
        Test t = new Test(length, width);  
        t.display();  
    }  
}
```

Java

### Output:

```
Enter length of rectangle:12  
Enter width of rectangle:3  
Area of rectangle is: 36
```

#3

```
class Student{
    String name;
    public Student(String s){
        name = s;
    }
    public Student(){
        name = "Unknown";
    }
}

class Ans{
    public static void main(String[] args){
        Student s = new Student("xyz");
        Student a = new Student();

        System.out.println(s.name);
        System.out.println(a.name);
    }
}
```

## **Problem: Student Grade Analyzer** (To illustrate the use of loops and decision-making statements in Java)

You are tasked with developing a simple program to analyze and report student grades. The program should:

1. **Prompt the user to enter the number of students.**
2. **For each student, prompt the user to enter their grade.**
3. **Determine if each grade falls into the following categories:**
  - o "Excellent" (grade  $\geq 90$ )
  - o "Good" ( $70 \leq \text{grade} < 90$ )
  - o "Average" ( $50 \leq \text{grade} < 70$ )
  - o "Below Average" ( $\text{grade} < 50$ )
4. **Calculate and display the following statistics:**
  - o Total number of students.
  - o Number of students in each grade category.
  - o The highest grade entered.
  - o The lowest grade entered.

## **Requirements**

1. **Use loops to iterate through the number of students and grades.**
2. **Use decision-making statements (e.g., `if-else`) to categorize grades and calculate statistics.**

## Main.java



Run

```
1 import java.util.Scanner;  
2  
3 public class GradeAnalyzer {  
4  
5     public static void main(String[] args) {  
6         Scanner scanner = new Scanner(System.in);  
7  
8         // Step 1: Prompt the user to enter the number of students  
9         System.out.print("Enter the number of students: ");  
10        int numStudents = scanner.nextInt();  
11  
12        // Initialize variables for statistics  
13        int countExcellent = 0;  
14        int countGood = 0;  
15        int countAverage = 0;  
16        int countBelowAverage = 0;  
17        int highestGrade = Integer.MIN_VALUE;  
18        int lowestGrade = Integer.MAX_VALUE;  
19
```

```
20     // Step 2: Loop through each student and get their grade
21     for (int i = 0; i < numStudents; i++) {
22         System.out.print("Enter grade for student " + (i + 1) + ": ");
23         int grade = scanner.nextInt();
24
25         // Update highest and lowest grade
26         if (grade > highestGrade) {
27             highestGrade = grade;
28         }
29         if (grade < lowestGrade) {
30             lowestGrade = grade;
31         }
32
33         // Step 3: Categorize the grade
34         if (grade >= 90) {
35             countExcellent++;
36         } else if (grade >= 70) {
37             countGood++;
38         } else if (grade >= 50) {
39             countAverage++;
40         } else {
41             countBelowAverage++;
42         }
43     }
```

```
45     // Close the scanner           → The Scanner.close() method is essential for releasing  
46     scanner.close();  
47  
48     // Step 4: Display the statistics  
49     System.out.println("Total number of students: " + numStudents);  
50     System.out.println("Number of students with Excellent grades: " +  
51         countExcellent);  
52     System.out.println("Number of students with Good grades: " + countGood  
53         );  
54     System.out.println("Number of students with Average grades: " +  
55         countAverage);  
56     System.out.println("Number of students with Below Average grades: " +  
57         countBelowAverage);  
58     System.out.println("Highest grade entered: " + highestGrade);  
59     System.out.println("Lowest grade entered: " + lowestGrade);  
60 }
```

## Output

Clear

```
java -cp /tmp/e0Z434Ro8B/GradeAnalyzer
Enter the number of students: 8
Enter grade for student 1: 78
Enter grade for student 2: 90
Enter grade for student 3: 87
Enter grade for student 4: 23
Enter grade for student 5: 45
Enter grade for student 6: 56
Enter grade for student 7: 78
Enter grade for student 8: 56
Total number of students: 8
Number of students with Excellent grades: 1
Number of students with Good grades: 3
Number of students with Average grades: 2
Number of students with Below Average grades: 2
Highest grade entered: 90
Lowest grade entered: 23

==== Code Execution Successful ====
```

# Explanation

1. **Loops:**
  - A **for** loop is used to iterate through the number of students and collect their grades.
2. **Decision-Making Statements:**
  - **if-else** statements are used to categorize each grade into "Excellent", "Good", "Average", or "Below Average" based on the grade value.
3. **Statistics Calculation:**
  - During the loop, the program keeps track of the highest and lowest grades and counts the number of students in each grade category.

**Q. You are working on a Java application that involves defining and using constants and immutable objects. You encounter the following code snippet and need to debug and understand how the `final` keyword affects the behavior of the code.**

```
1  public class FinalKeywordDemo {  
2      // Final class: cannot be subclassed  
3      public final class ImmutableClass {  
4          private final int value; // Final instance variable  
5  
6          public ImmutableClass(int value) {  
7              this.value = value;  
8          }  
9  
10         public int getValue() {  
11             return value;  
12         }  
13  
14         // Final method: cannot be overridden in subclasses  
15         public final void displayValue() {  
16             System.out.println("Value: " + value);  
17         }  
18     }  
19  
20     public static void main(String[] args) {  
21         FinalKeywordDemo demo = new FinalKeywordDemo();  
22         ImmutableClass obj = demo.new ImmutableClass(10);  
23  
24         // Attempt to modify the value (commented out as it's not allowed)  
25         // obj.value = 20; // Compilation error: cannot assign a value to final variable value  
26  
27         // Attempt to call a final method  
28         obj.displayValue();  
29  
30         // Uncommenting the following lines would cause errors:  
31         // subclass that tries to extend ImmutableClass  
32         // ImmutableClass subclassObj = demo.new SubClass(30);  
33         // subclassObj.displayValue();  
34     }  
35  
36     // Uncommenting the following class will cause a compilation error:  
37     /*  
38         public class SubClass extends ImmutableClass {  
39             public SubClass(int value) {  
40                 super(value);  
41             }  
42  
43             @Override  
44             public void displayValue() {  
45                 System.out.println("Overridden value display");  
46             }  
47         }  
48     */  
49 }
```

## Tasks:

1. Explain the following aspects of the `final` keyword as used in this code:
  - The impact of `final` on the `ImmutableClass`.
  - The impact of `final` on the `value` instance variable.
  - The impact of `final` on the `displayValue` method.
2. What errors would occur if you uncomment the sections of code related to extending `ImmutableClass` and overriding `displayValue`?
3. Why is the `value` instance variable marked as `final`, and what does this mean for the `ImmutableClass` object after it's created?
4. Explain why you cannot modify the `displayValue` method or extend `ImmutableClass` with a subclass.

## Answers:

### 1. Explanation of the `final` keyword:

- **final class (`ImmutableClass`):** A `final` class cannot be subclassed. This means you cannot create a class that extends `ImmutableClass`, and thus no class can inherit from it.
- **final instance variable (`value`):** A `final` instance variable can only be assigned once. Once initialized in the constructor, it cannot be changed. This ensures that the state of the `ImmutableClass` object remains constant.
- **final method (`displayValue`):** A `final` method cannot be overridden by any subclass. If `ImmutableClass` were not `final`, a subclass could override `displayValue`, but this is prevented here.

### 2. Errors from uncommenting:

- **Extending `ImmutableClass` (`SubClass`):** This will cause a compilation error because `ImmutableClass` is marked as `final` and cannot be subclassed.
- **Overriding `displayValue` method:** This will also cause a compilation error because `displayValue` is marked as `final` and cannot be overridden.

### 3. `final` instance variable (`value`):

Marking `value` as `final` means that once it is set in the constructor, it cannot be changed. This ensures that the value assigned to `value` remains constant for the lifetime of the `ImmutableClass` object, making it immutable.

### 4. Reason for restrictions:

- **Cannot modify `displayValue` method:** This is to ensure that the behavior of the `displayValue` method remains consistent and cannot be altered by subclasses.
- **Cannot extend `ImmutableClass`:** This enforces the immutability and integrity of the class by preventing inheritance, which could potentially modify or extend the behavior in unintended ways.

## Q. Problem Statement:

You are working on a Java class that represents a simple bank account. The **Account** class has the following attributes and methods:

- **Attributes:**
  - **accountNumber** (String): The account number of the bank account.
  - **accountHolderName** (String): The name of the account holder.
  - **balance** (double): The current balance of the account.
- **Methods:**
  - **Constructor:** `Account(String accountNumber, String accountHolderName, double balance)`
    - Initializes the `accountNumber`, `accountHolderName`, and `balance` attributes.
  - **Method:** `void updateAccount(String accountNumber, String accountHolderName, double balance)`
    - Updates the `accountNumber`, `accountHolderName`, and `balance` of the account. It should use the `this` keyword to distinguish between instance variables and parameters with the same names.
  - **Method:** `void displayAccountInfo()`
    - Displays the account information in the format: `Account Number : [accountNumber], Holder Name : [accountHolderName], Balance : [balance]`.

### Task:

1. Implement the **Account** class in Java with the specified attributes and methods.
2. Ensure that the `updateAccount` method uses the `this` keyword to correctly refer to the instance variables.
3. Write a main method to create an **Account** object, update its details, and display its information.

```
1 public class Account {  
2     // Attributes  
3     private String accountNumber;  
4     private String accountHolderName;  
5     private double balance;  
6  
7     // Constructor  
8     public Account(String accountNumber, String accountHolderName, double balance) {  
9         this.accountNumber = accountNumber;  
10        this.accountHolderName = accountHolderName;  
11        this.balance = balance;  
12    }  
13  
14    // Update method using 'this' keyword  
15    public void updateAccount(String accountNumber, String accountHolderName, double balance) {  
16        this.accountNumber = accountNumber;  
17        this.accountHolderName = accountHolderName;  
18        this.balance = balance;  
19    }  
20  
21    // Display method  
22    public void displayAccountInfo() {  
23        System.out.println("Account Number: " + accountNumber);  
24        System.out.println("Holder Name: " + accountHolderName);  
25        System.out.println("Balance: " + balance);  
26    }  
27  
28    public static void main(String[] args) {  
29        // Create an Account object  
30        Account myAccount = new Account("123456", "John Doe", 1000.0);  
31  
32        // Display account information  
33        myAccount.displayAccountInfo();  
34  
35        // Update account information  
36        myAccount.updateAccount("654321", "Jane Doe", 1500.0);  
37  
38        // Display updated account information  
39        myAccount.displayAccountInfo();  
40    }  
41 }
```

**Q. Write a Java program to create a class Vehicle with a method called speedUp(). Create two subclasses Car and Bicycle. Override the speedUp() method in each subclass to increase the vehicle's speed differently.**

```
1 // Base class
2 class Vehicle {
3     // Default speed of the vehicle
4     protected int speed = 0;
5
6     // Method to increase the speed of the vehicle
7     public void speedUp() {
8         System.out.println("The vehicle is speeding up.");
9     }
10
11    // Method to get the current speed of the vehicle
12    public int getSpeed() {
13        return speed;
14    }
15}
16
17 // Subclass Car
18 class Car extends Vehicle {
19     @Override
20     public void speedUp() {
21         speed += 20; // Increase speed by 20 for Car
22         System.out.println("The car speeds up. Current speed: " + speed + " km/h");
23     }
24}
25
26 // Subclass Bicycle
27 class Bicycle extends Vehicle {
28     @Override
29     public void speedUp() {
30         speed += 5; // Increase speed by 5 for Bicycle
31         System.out.println("The bicycle speeds up. Current speed: " + speed + " km/h");
32     }
33}
34
```

```
35 // Main class to test the functionality
36 public class Main {
37     public static void main(String[] args) {
38         // Create instances of Car and Bicycle
39         Vehicle myCar = new Car();
40         Vehicle myBicycle = new Bicycle();
41
42         // Speed up the car and bicycle
43         myCar.speedUp();           // Output: The car speeds up. Current speed: 20 km/h
44         myBicycle.speedUp();      // Output: The bicycle speeds up. Current speed: 5 km/h
45
46         // Further speed up the car and bicycle
47         myCar.speedUp();           // Output: The car speeds up. Current speed: 40 km/h
48         myBicycle.speedUp();      // Output: The bicycle speeds up. Current speed: 10 km/h
49     }
50 }
51 }
```

## Output    Generated Files

```
The car speeds up. Current speed: 20 km/h
The bicycle speeds up. Current speed: 5 km/h
The car speeds up. Current speed: 40 km/h
The bicycle speeds up. Current speed: 10 km/h
```

ⓘ CPU Time: 0.03 sec(s) | Memory: 38564 kilobyte(s) | Compiled and executed in 1.388 sec(s)

## Explanation:

### 1. Base Class Vehicle:

- It has a `protected` field `speed` to store the vehicle's speed.
- The `speedUp()` method prints a default message and increases the speed (default implementation can be overridden by subclasses).
- The `getSpeed()` method returns the current speed of the vehicle.

### 2. Subclass Car:

- Overrides `speedUp()` to increase the speed by 20 km/h.
- Prints the updated speed to the console.

### 3. Subclass Bicycle:

- Overrides `speedUp()` to increase the speed by 5 km/h.
- Prints the updated speed to the console.

### 4. Main Class (Main):

- Creates instances of `Car` and `Bicycle`.
- Calls `speedUp()` on each instance to demonstrate polymorphism and different speed increments.

# Java Command Line Arguments

- The java command-line argument is an argument i.e. **passed at the time of running** the java program.
- The arguments **passed from the console** can be received in the java program and it can be used as an input.
- We need to pass the arguments as space-separated values.
- We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments. These arguments **convert into a string array** and are provided to the main() function as a string array argument.
- When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[]. It can be confirmed that they are wrapped up in an args array by checking the length of args using args.length.
- Internally, JVM wraps up these command-line arguments into the args[] array that we pass into the main() function.
- We can check these arguments using args.length method. JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.

## Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
    public static void main(String args[]){
        System.out.println("Your first argument is: "+args[0]);
    }
}
```

compile by > javac CommandLineExample.java  
run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo

## Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A{  
    public static void main(String args[]){  
  
        for(int i=0;i<args.length;i++){  
            System.out.println(args[i]);  
  
        }  
    }  
}
```

compile by > javac A.java  
run by > java A sonoo jaiswal 1 3 abc

Output: sonoo  
jaiswal  
1  
3  
abc

Q. Write a Java program that:

- 1. Accepts a list of integers as command-line arguments.**
- 2. Performs the following operations:**
  - Calculates the sum of the integers.
  - Calculates the average of the integers.
  - Finds the maximum and minimum values from the list.
  - Displays the results.
- 3. Handles the case where no integers are provided.**

```
1 public class CommandLineProcessor {  
2  
3     public static void main(String[] args) {  
4         // Check if any command-line arguments are provided  
5         if (args.length == 0) {  
6             System.out.println("No integers provided. Please provide integers as command-line arguments.");  
7             return;  
8         }  
9  
10        // Initialize variables for calculations  
11        int sum = 0;  
12        int count = args.length;  
13        int max = Integer.MIN_VALUE;  
14        int min = Integer.MAX_VALUE;  
15  
16        // Process each command-line argument  
17        for (String arg : args) {  
18            try {  
19                // Convert argument to integer  
20                int number = Integer.parseInt(arg);  
21  
22                // Update sum  
23                sum += number;  
24  
25                // Update max and min  
26                if (number > max) {  
27                    max = number;  
28                }  
29                if (number < min) {  
30                    min = number;  
31                }  
32            } catch (NumberFormatException e) {  
33                System.out.println("Invalid input: " + arg + " is not an integer.");  
34                return;  
35            }  
36        }  
37    }
```

```
39     // Calculate average
40     double average = (double) sum / count;
41
42     // Display the results
43     System.out.println("Total number of integers: " + count);
44     System.out.println("Sum of integers: " + sum);
45     System.out.println("Average of integers: " + average);
46     System.out.println("Maximum value: " + max);
47     System.out.println("Minimum value: " + min);
48 }
49 }
```

## Running the Program

To run the program, compile it and provide command-line arguments like so:

```
sh
javac CommandLineProcessor.java
java CommandLineProcessor 10 20 30 40 50
```

### Output:

```
yaml
Total number of integers: 5
Sum of integers: 150
Average of integers: 30.0
Maximum value: 50
Minimum value: 10
```

# Explanation

## 1. Command-Line Arguments:

- The `args` array contains the command-line arguments. Each argument is processed as a `String` which is then converted to an `int` using `Integer.parseInt()`.

## 2. Processing and Calculations:

- **Sum**: Added up during the loop.
- **Maximum and Minimum Values**: Updated based on comparisons with the current `max` and `min`.
- **Average**: Calculated after the loop by dividing the sum by the number of integers.

## 3. Error Handling:

- If a non-integer argument is encountered, a `NumberFormatException` is caught, and an error message is printed.

## 4. Output:

- The results (total number of integers, sum, average, maximum, and minimum values) are printed to the console.

**Q. You are given the following Java code involving inheritance, and you need to debug and correct the usage of the `super` keyword. The code is designed to simulate a basic banking system with two classes: `Account` and `SavingsAccount`.**

```
1  class Account {
2      private double balance;
3
4      public Account(double balance) {
5          this.balance = balance;
6      }
7
8      public void deposit(double amount) {
9          this.balance += amount;
10     }
11
12     public void withdraw(double amount) {
13         if (amount <= this.balance) {
14             this.balance -= amount;
15         } else {
16             System.out.println("Insufficient funds");
17         }
18     }
19
20     public double getBalance() {
21         return this.balance;
22     }
23 }
24
25 class SavingsAccount extends Account {
26     private double interestRate;
27
28     public SavingsAccount(double balance, double interestRate) {
29         super(balance);
30         this.interestRate = interestRate;
31     }
32
33     public void addInterest() {
34         double interest = this.getBalance() * this.interestRate;
35         this.deposit(interest); // Issue here
36     }
37
38     public double getInterestRate() {
39         return this.interestRate;
40     }
41 }
42 }
```

```
42
43 public class Main {
44     public static void main(String[] args) {
45         SavingsAccount myAccount = new SavingsAccount(1000.0, 0.05);
46         myAccount.addInterest();
47         System.out.println("Balance after interest: " + myAccount.getBalance());
48     }
49 }
50
```

## Task:

- 1. Identify the Problem:** Analyze the code and identify any issues related to the usage of the `super` keyword or related inheritance problems.
- 2. Debug the Code:** Correct the identified issues to ensure that the `SavingsAccount` class properly adds interest to the balance.

## Hints:

- Pay attention to how the `deposit` method is called in the `addInterest` method.
- Ensure that methods from the `Account` class are properly accessed and used in the `SavingsAccount` class.

**Expected Output:** After fixing the code, running the `Main` class should output the balance of `1050.0`, showing that the interest has been correctly added.

## Solution

The issue in the code is that the `deposit` method is called in the `addInterest` method, but it should be accessing the `deposit` method from the superclass `Account` correctly. However, the call to `deposit` is valid as it is correctly inherited from the superclass.

To ensure the `addInterest` method correctly adds interest to the balance, you can review and confirm the correctness of the method calls and logic:

```
java                                     ⌂ Copy code

public void addInterest() {
    double interest = this.getBalance() * this.interestRate;
    super.deposit(interest); // Use super to explicitly call the deposit method from the :
}
```

Using `super.deposit(interest)` clarifies that the `deposit` method from the `Account` class is being called, though in this case, it's not strictly necessary unless you are overriding methods or dealing with ambiguity.

After applying the fix, the `Main` class should produce the expected output:

```
java                                     ⌂ Copy code

Balance after interest: 1050.0
```

## Question:

Write a C++ class named `ResourceManager` that simulates managing a resource. **This class should allocate memory for a dynamic array of integers when an object is created and deallocate the memory when the object is destroyed.** Implement the following functionalities:

1. The class should have a constructor that takes an integer argument `size` which specifies the number of elements in the array. The constructor should allocate memory for the array and initialize it with the values from 1 to `size`.
2. The class should have a **destructor** that frees the allocated memory to prevent memory leaks.
3. Implement a member function `display()` that prints the elements of the array to the console.
4. Write a `main()` function that creates an object of `ResourceManager`, uses it to display the array elements, and then lets the object go out of scope to trigger the destructor.

## Main.cpp

```
1 #include <iostream>
2
3 class ResourceManager {
4 private:
5     int* array; // Pointer to hold the dynamic array
6     int size;   // Size of the array
7
8 public:
9     // Constructor that allocates memory and initializes the array
10    ResourceManager(int size) : size(size) {
11        // Allocate memory for the array
12        array = new int[size];
13        // Initialize the array with values from 1 to size
14        for (int i = 0; i < size; ++i) {
15            array[i] = i + 1;
16        }
17    }
18
19    // Destructor that deallocates the memory
20    ~ResourceManager() {
21        delete[] array;
22    }
23
24    // Member function to display the elements of the array
25    void display() const {
26        for (int i = 0; i < size; ++i) {
27            std::cout << array[i] << " ";
28        }
29        std::cout << std::endl;
30    }
31 };
32
33 int main() {
34     // Create an object of ResourceManager with a specified size
35     ResourceManager rm(5);
36
37     // Use the object to display the array elements
38     rm.display();
39
40     // The object `rm` will go out of scope here, and the destructor will be called
41
42     return 0;
43 }
```

Output:

1 2 3 4 5

## Explanation:

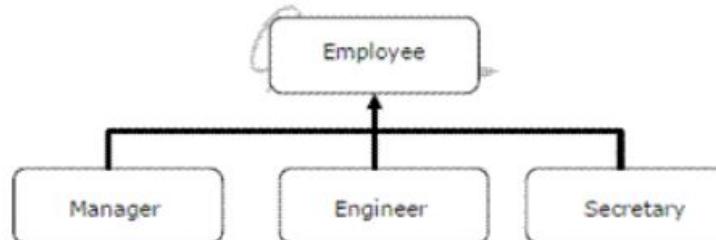
1. **Constructor (`ResourceManager(int size)`):**
  - Allocates memory for the dynamic array of integers.
  - Initializes the array elements with values from 1 up to `size`.
2. **Destructor (`~ResourceManager()`):**
  - Frees the memory allocated for the dynamic array to prevent memory leaks.
3. **Member Function (`display() const`):**
  - Prints the elements of the array to the console.
4. **`main()` Function:**
  - Creates an instance of `ResourceManager` with 5 elements.
  - Displays the contents of the array.
  - When the `ResourceManager` object goes out of scope at the end of `main()`, the destructor is automatically invoked to clean up the allocated memory.

This implementation ensures that **memory management is handled correctly** and that the resource is properly cleaned up when no longer needed.

# QUIZ TIME !!!

1. It is desired to design an object-oriented employee record system for a company. Each employee has a name, unique id and salary. Employees belong to different categories and their salary is determined by their category. The functions to get Name, getId and compute salary are required. Given the class hierarchy below, possible locations for these functions are: (GATE CS 2004)

- i. getId is implemented in the superclass
- ii. getId is implemented in the subclass
- iii. getName is an abstract function in the superclass
- iv. getName is implemented in the superclass
- v. getName is implemented in the subclass
- vi. getSalary is an abstract function in the superclass
- vii. getSalary is implemented in the superclass
- viii. getSalary is implemented in the subclass



Choose the best design

- (a) (i), (iv), (vi), (viii)
- (b) (i), (iv), (vii)
- (c) (i), (iii), (v), (vi), (viii)
- (d) (ii), (v), (viii)

### Answer (a)

Getid() and GetName() can be there in the base class as these functions have the same implementation for all subclasses. As the question says that every employee must have salary and salary is determined by their category, getSalary() must be there as an abstract function in base class. And all subclasses should implement salary according to their category.

Which one of the following are essential features of an object-oriented programming language?

- i) Abstraction and encapsulation
- ii) Strictly-typedness
- iii) Type-safe property coupled with sub-type rule
- iv) Polymorphism in the presence of inheritance

A (i) and (ii) only

B (i) and (iv) only

C (i), (ii) and (iv) only

D (i), (ii) and (iii) only

A (i) and (ii) only

Correct Answer

B (i) and (iv) only

C (i), (ii) and (iv) only

D (i), (ii) and (iii) only

**Which of these features of OOP would indicate code reusability?**

- a. Polymorphism
- b. Abstraction
- c. Inheritance
- d. Encapsulation

Answer: (c) Inheritance

**A constructor** \_\_\_\_\_.

- a. May consist of a return type
- b. Doesn't consist of any return type
- c. Has some return type
- d. Of a derived class has a return type

Answer: (b) Doesn't consist of any return type

**If in case, in multiple inheritances, a class R would inherit the Class Q, while Class Q would inherit the class P, then in which sequence would their destructors be called in case we declare an object of Class R?**

- a. ~R() then ~P() then ~Q()
- b. ~P() then ~Q() then ~R()
- c. ~Q() then ~R() then ~P()
- d. ~R() then ~Q() then ~P()

**Answer:** (d) ~R() then ~Q() then ~P()

**Which header file is required in C++ to use OOP?**

- a) OOP can be used without using any header file
- b) stdlib.h
- c) iostream.h
- d) stdio.h

**Answer:** (a)

**How many types of access specifiers are provided in OOP (C++)?**

- a) 4
- b) 3
- c) 2
- d) 1

Answer: (b)

**What is encapsulation in OOP?**

- a) It is a way of combining various data members and member functions that operate on those data members into a single unit
- b) It is a way of combining various data members and member functions into a single unit which can operate on any data
- c) It is a way of combining various data members into a single unit
- d) It is a way of combining various member functions into a single unit

Answer: (a)

17. Which constructor will be called from the object created in the below C++ code?

```
class A
{
    int i;
    A()
    {
        i=0; cout<i;
    }
    A(int x=0)
    {
        i=x; cout<i;
    }
};

A obj1;
```

- a) Parameterized constructor
- b) Default constructor
- c) Run time error
- d) Compile time error

## Answer: d

Explanation: When a default constructor is defined and another constructor with 1 default value argument is defined, creating object without parameter will create **ambiguity for the compiler**. The compiler won't be able to decide which constructor should be called, hence compile time error.

In which access should a constructor be defined, so that object of the class can be created in any function?

- a) Any access specifier will work
- b) Private
- c) Public
- d) Protected

Answer (c)

21. Which among the following is correct for the class defined below?

```
class student
{
    int marks;
    public: student(){}
    student(int x)
    {
        marks=x;
    }
};

main()
{
    student s1(100);
    student s2();
    student s3=100;
    return 0;
}
```

- a) Program will give compile time error
- b) Object s3, syntax error
- c) Only object s1 and s2 will be created
- d) Program runs and all objects are created

## Answer: d

Explanation: It is a special case of constructor with only 1 argument. While calling a constructor with one argument, you are actually implicitly **creating a conversion from the argument type to the type of class**. Hence you can directly specify the value of that one argument with assignment operator.

13. What are the total catch blocks that can be used using a single Try block?

- 1
- 2
- According to needs
- Maximum 256

Answer: (c)

14. Identify the scope resolution operator.

- :
- ::
- ?:
- None

Answer: (b)

31. Choose the option below for which instance of the class cannot be created.

- Anonymous class
- Parent class
- Nested class
- Abstract class

Answer: (d)

36. Choose the option below which is used to free the memory allocated for an object in C++.

- Free()
- unallocate()
- delete
- collect

Answer: (c)

**Which of the following differentiates between overloaded functions and overridden functions?**

- A. Overloading is a dynamic or runtime binding and overridden is a static or compile time binding.
- B. Overloading is a static or compile time binding and overriding is dynamic or runtime binding.
- C. Redefining a function in a friend class is called overloading, while redefining a function in a derived class is called as overridden function.
- D. Redefining a function in a derived class is called function overloading, while redefining a function in a friend class is called function overriding.

Answer: (B)

**The copy constructors can be used to \_\_\_\_\_**

- a) Copy an object so that it can be passed to another primitive type variable
- b) Copy an object for type casting
- c) Copy an object so that it can be passed to a function
- d) Copy an object so that it can be passed to a class

Answer: (c)

## **Which among the following represents correct constructor?**

- a) -classname()
- b) classname()
- c) ()classname
- d) ~classname()

**Answer:** (b)

Which of the statements are CORRECT about destructors?

- (A) Destructors are invoked automatically when objects go out of scope or are explicitly deleted.
- (B) Destructors can have parameters in some programming languages.
- (C) Destructors do not have return types and therefore cannot return values.
- (D) A class can have multiple destructors.

Choose the correct answer from the options given below:

1. (A), (C), and (B) Only

2. (A), (B), and (D) Only

Answer: (4)

3. (A), (C), and (D) Only

4. (A), and (C) Only

In object-oriented programming (OOP), which of the following concepts allows different classes to be treated as instances of the same superclass?

1. Encapsulation
2. Abstraction
3. Polymorphism
4. Inheritance

Answer: (3)

In which case it is mandatory to provide a destructor in a class?

1. Almost in every class
2. Class for which two or more than two objects will be created
3. Class for which copy constructor
4. Class whose objects will be created dynamically

Answer: (4)

\_\_\_\_\_ is used to declare and initialize an object from another object of same class in C++.

1. Copy constructor
2. Default constructor
3. Overloaded constructor
4. Non-parametrized constructor

Answer: (1)

**Assertion (A):** C++ is an object-oriented programming language.

**Reason (R):** C++ supports class, inheritance, templates, and exception handling.

1. Both (A) and (R) are true and (R) is correct explanation of (A).
2. Both (A) and (R) are true, but (R) is not the correct explanation of (A).
3. (A) is true, but (R) is false.
4. (A) is false, but (R) is true.

Answer: (1)

**ANY  
QUESTIONS?**

