OOPS

Tuesday, July 13, 2021 10:22 PM

MISCELLANEOUS

Differences between Procedural and Object Oriented Programming

Procedural Programming:

Procedural Programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself.

Languages used in Procedural Programming:

```
FORTRAN, ALGOL, COBOL,
BASIC, Pascal and C.
```

Object Oriented Programming:

Object oriented programming can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept of objects that interact with real world. Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Languages used in Object Oriented Programming:

```
Java, C++, C#, Python,
PHP, JavaScript, Ruby, Perl,
Objective-C, Dart, Swift, Scala.
```

Difference between Procedural Programming and Object Oriented Programming:

Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding	Object oriented programming provides data hiding so it is ***more***

data so it is ***less secure***.	***secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on ***unreal world***.	Object oriented programming is based on ***real world***.
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

From <https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/>

Why Java is not a purely Object-Oriented Language?

- Difficulty Level : Easy
- Last Updated : 30 May, 2017

Pure Object Oriented Language or Complete Object Oriented Language are Fully Object Oriented Language which supports or have features which treats everything inside program as objects. It doesn't support primitive datatype(like int, char, float, bool, etc.). There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.
   Example: Smalltalk

Why Java is not a Pure Object Oriented Language?

Java supports property 1, 2, 3, 4 and 6 but fails to support property 5 and 7 given above. Java language is not a Pure Object Oriented Language as it contain these properties:

- **Primitive Data Type ex. int, long, bool, float, char, etc as Objects:** Smalltalk is a "pure" object-oriented programming language unlike Java and C++ as there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects.
  In Java, we have predefined types as non-objects (primitive types).

```
int a = 5;
System.out.print(a);
```

- **The static keyword:** When we declares a class as static then it can be used without the use of an object in Java. If we are using static function or static variable then we can't call that function or variable by using dot(.) or class object defying object oriented feature.
- **Wrapper Class:** Wrapper class provides the mechanism to convert primitive into object and object into primitive. In Java, you can use Integer, Float etc. instead of int, float etc. We can communicate with objects without calling their methods. ex. using arithmetic operators.

```
String s1 = "ABC" + "A" ;
```

Even using Wrapper classes does not make Java a pure OOP language, as internally it will use the operations like Unboxing and Autoboxing. So if you create instead of int Integer and do any mathematical operation on it, under the hoods Java is going to use primitive type int only.

```
public class BoxingExample
{
```

```
    public static void main(String[] args)
    {
            Integer i = new Integer(10);
            Integer j = new Integer(20);
            Integer k = new Integer(i.intValue() +
j.intValue());
            System.out.println("Output: "+ k);
    }
}
```

In the above code, there are 2 problems where Java fails to work as pure OOP:

8. While creating Integer class you are using primitive type “int” i.e. numbers 10, 20.
9. While doing addition Java is using primitive type “int”.

From <https://www.geeksforgeeks.org/java-not-purely-object-oriented-language/>

Is an array a primitive type or an object in Java?

- Difficulty Level : Medium
- Last Updated : 11 Dec, 2018

An array in Java **is an object**. Now the question how is this possible? What is the reason behind that? In Java, we can create arrays by using new operator and we know that every object is created using new operator. Hence we can say that array is also an object. Now the question also arises, every time we create an object for a class then what is the class of array?

- In Java, there is a class for every array type, so there’s a class for int[] and similarly for float, double etc.
- The direct superclass of an array type is Object. Every array type implements the interfaces Cloneable and java.io.Serializable.
- In the Java programming language, arrays are objects (§4.3.1), are dynamically created, and may be assigned to variables of type Object (§4.3.2). All methods of class Object may be invoked on an array.

For every array type corresponding classes are available and these classes are the part of java language and not available to the programmer level. To know the class of any array, we can go with the following approach:

```
// Here x is the name of the array.
System.out.println(x.getClass().getName());
```

```
// Java program to display class of
// int array type
public class Test
{
    public static void main(String[] args)
    {
        int[] x = new int[3];
        System.out.println(x.getClass().getName());
    }
}
```

Output:

```
[I
```

NOTE:[I this is the class for this array, one [(square bracket) because it is one dimensional and I for integer data type.

Here is the table specifying the corresponding class name for some array types:-

Array type	Corresponding class Name
int[]	[I
int[][]	[[I
double[]	[D
double[][]	[[D
short[]	[S
byte[]	[B
boolean[]	[Z

In Java programming language, arrays are objects which are dynamically created, and may be assigned to variables of type

Object. All methods of class Object may be invoked on an array.

```
// Java program to check the class of
// int array type
public class Test {
    public static void main(String[] args)
    {
        // Object is the parent class of all classes
        // of Java. Here args is the object of String
        // class.
        System.out.println(args instanceof Object);

        int[] arr = new int[2];

        // Here arr is also an object of int class.
        System.out.println(arr instanceof Object);
    }
}
```

Output:

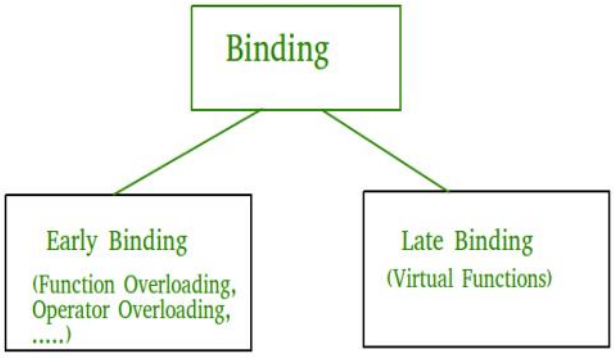
```
true
true
```

From <https://www.geeksforgeeks.org/array-primitive-type-object-java/>

Early binding and Late binding in C++

- Difficulty Level : Medium
- Last Updated : 05 Feb, 2018

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism) As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

By default early binding happens in C++. Late binding (discussed below) is achieved with the help of virtual keyword)

```
// CPP Program to illustrate early binding.
// Any normal function call (without virtual)
// is binded early. Here we have taken base
// and derived class example so that readers
// can easily compare and see difference in
// outputs.
#include<iostream>
using namespace std;

class Base
{
public:
```

```
    void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived;

    // The function call decided at
    // compile time (compiler sees type
    // of pointer and calls base class
    // function.
    bp->show();

    return 0;
}
```

Output:

```
In Base
```

Late Binding : (Run time polymorphism) In this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition (Refer this for details). This can be achieved by declaring a virtual function.

```
// CPP Program to illustrate late binding
#include<iostream>
using namespace std;

class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived;
    bp->show();  // RUN-TIME POLYMORPHISM
    return 0;
}
```

Output:

```
In Derived
```

From <https://www.geeksforgeeks.org/early-binding-late-binding-c/>

Access Modifiers in Java

- Difficulty Level : Easy
- Last Updated : 25 May, 2021

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:

10. Default – No keyword required
11. Private
12. Protected
13. Public

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- **Default**: When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default.
- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.
  In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.
- Java

```
// Java program to illustrate default modifier
package p1;

// Class Geeks is having Default access modifier
class Geek
{
    void display()
    {
        System.out.println("Hello World!");
    }
}
```

- Java

```
// Java program to illustrate error while
// using class from different package with
// default modifier
package p2;
import p1.*;

// This class is having default access modifier
class GeekNew
{
    public static void main(String args[])
    {
        // Accessing class Geek from package p1
        Geeks obj = new Geek();

        obj.display();
    }
}
```

Output:

```
Compile time error
```

- **Private**: The private access modifier is specified using the keyword **private**.
- The methods or data members declared as private are accessible only **within the class** in which they are declared.
- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because

14. private means “only visible within the enclosing class”.
15. protected means “only visible within the enclosing class and any subclasses”
    Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

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

- Java

```
// Java program to illustrate error while
// using class from different package with
// private modifier
package p1;

class A
{
private void display()
    {
        System.out.println("GeeksforGeeks");
    }
}

class B
{
public static void main(String args[])
    {
        A obj = new A();
        // Trying to access private method
        // of another class
        obj.display();
    }
}
```

Output:

```
error: display() has private access in A
        obj.display();
```

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

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

- Java

```
// Java program to illustrate
// protected modifier
package p1;

// Class A
public class A
{
protected void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
```

- Java

```
// Java program to illustrate
// protected modifier
package p2;
import p1.*; // importing all classes in package p1

// Class B is subclass of A
class B extends A
{
public static void main(String args[])
{
    B obj = new B();
    obj.display();
}

}
```

Output:

GeeksforGeeks

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

```
// Java program to illustrate
// public modifier
package p1;
public class A
{
public void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
```

- Java

```
package p2;
import p1.*;
class B {
    public static void main(String args[])
    {
        A obj = new A;
        obj.display();
    }
}
```

Output:

GeeksforGeeks

Important Points:

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

From <https://www.geeksforgeeks.org/access-modifiers-java/>

Abstract Classes in Java

- Difficulty Level : Easy
- Last Updated : 19 Feb, 2021

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

- Java

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function
in
    // C++)
    abstract void draw();
}
```

Following are some important observations about abstract classes in Java.

1) Like C++, in Java, an instance of an abstract class cannot be created, we can have references to abstract class type though.

- Java

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun()
    {
        System.out.println("Derived fun() called");
```

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

        // Uncommenting the following line will cause
        // compiler error as the line tries to create an
        // instance of abstract class. Base b = new
Base();

        // We can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

Output

```
Derived fun() called
```

2) Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created. For example, the following is a valid Java program.

- Java

```
// An abstract class with constructor
abstract class Base {
    Base()
    {
        System.out.println("Base Constructor Called");
    }
    abstract void fun();
}
class Derived extends Base {
    Derived()
    {
        System.out.println("Derived Constructor Called");
    }
    void fun()
    {
        System.out.println("Derived fun() called");
    }
}
class Main {
    public static void main(String args[])
    {
        Derived d = new Derived();
    }
}
```

Output

```
Base Constructor Called
Derived Constructor Called
```

3) In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited.

- Java

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun()
called"); }
}

class Derived extends Base {
}

class Main {
    public static void main(String args[])
    {
        Derived d = new Derived();
        d.fun();
    }
}
```

Output

```
Base fun() called
```

4) Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program

compiles and runs fine.

- Java

```
// An abstract class with a final method
abstract class Base {
    final void fun()
    {
        System.out.println("Derived fun() called");
    }
}

class Derived extends Base {
}

class Main {
    public static void main(String args[])
    {
        Base b = new Derived();
        b.fun();
    }
}
```

Output

```
Derived fun() called
```

5) For any abstract java class we are not allowed to create an object i.e., for abstract class instantiation is not possible.

- Java

```
// An abstract class example
abstract class Test {
    public static void main(String args[])
    {
        // Try to create an object
        Test t = new Test();
    }
}
```

Output:

```
Compile time error. Test is abstract;
cannot be instantiated Test t=new Test();
```

6) Similar to the interface we can define static methods in an abstract class that can be called independently without an object.

- Java

```
abstract class Party {
    static void doParty()
    {
        System.out.println("Lets have some fun!!");
    }
}

public class Main extends Party {
    public static void main(String[] args)
    {
        Party.doParty();
    }
}
```

Output

```
Lets have some fun!!
```

From <https://www.geeksforgeeks.org/abstract-classes-in-java/>

A Definition of Java Garbage Collection

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

How Java Garbage Collection Works

Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted. The garbage collection implementation lives in the JVM. Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification. Although there are many JVMs, Oracle's HotSpot is by far the most common. It offers a robust and mature set of garbage

collection options.
While HotSpot has multiple garbage collectors that are optimized for various use cases, all its garbage collectors follow the same basic process. In the first step, unreferenced objects are identified and marked as ready for garbage collection. In the second step, marked objects are deleted. Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap. The compaction process makes it easier to allocate memory to new objects sequentially after the block of memory allocated to existing objects.
All of HotSpot's garbage collectors implement a generational garbage collection strategy that categorizes objects by age. The rationale behind generational garbage collection is that most objects are short-lived and will be ready for garbage collection soon after creation.

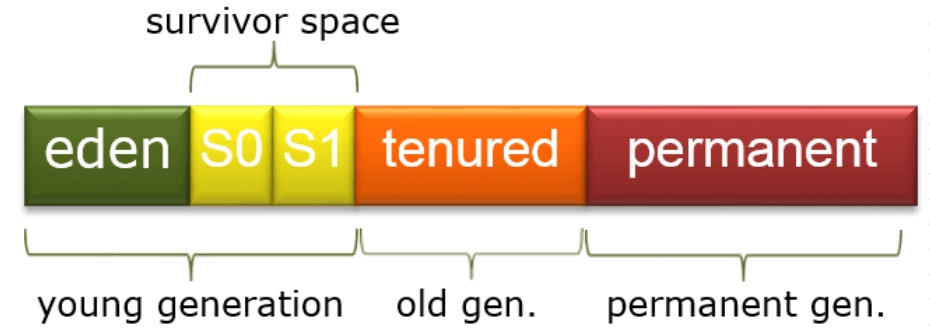


Image via Wikipedia
The heap is divided into three sections:

- **Young Generation**: Newly created objects start in the Young Generation. The Young Generation is further subdivided into an Eden space, where all new objects start, and two Survivor spaces, where objects are moved from Eden after surviving one garbage collection cycle. When objects are garbage collected from the Young Generation, it is a minor garbage collection event.
- **Old Generation:** Objects that are long-lived are eventually moved from the Young Generation to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.
- **Permanent Generation:** Metadata such as classes and methods are stored in the Permanent Generation. Classes that are no longer in use may be garbage collected from the Permanent Generation.

During a full garbage collection event, unused objects in all generations are garbage collected.
HotSpot has four garbage collectors:

- **Serial:** All garbage collection events are conducted serially in one thread. Compaction is executed after each garbage collection.
- **Parallel:** Multiple threads are used for minor garbage collection. A single thread is used for major garbage collection and Old Generation compaction. Alternatively, the Parallel Old variant uses multiple threads for major garbage collection and Old Generation compaction.
- **CMS (Concurrent Mark Sweep):** Multiple threads are used for minor garbage collection using the same algorithm as Parallel. Major garbage collection is multi-threaded, like Parallel Old, but CMS runs concurrently alongside application processes to minimize "stop the

world” events (i.e. when the garbage collector running stops the application). No compaction is performed.

- **G1 (Garbage First):** The newest garbage collector is intended as a replacement for CMS. It is parallel and concurrent like CMS, but it works quite differently under the hood compared to the older garbage collectors.

Benefits of Java Garbage Collection

The biggest benefit of Java garbage collection is that it automatically handles the deletion of unused objects or objects that are out of reach to free up vital memory resources. Programmers working in languages without garbage collection (like C and C++) must implement manual memory management in their code.

Despite the extra work required, some programmers argue in favor of manual memory management over garbage collection, primarily for reasons of control and performance. While the debate over memory management approaches continues to rage on, garbage collection is now a standard component of many popular programming languages. For scenarios in which the garbage collector is negatively impacting performance, Java offers many options for tuning the garbage collector to improve its efficiency.

Java Garbage Collection Best Practices

For many simple applications, Java garbage collection is not something that a programmer needs to consciously consider. However, for programmers who want to advance their Java skills, it is important to understand how Java garbage collection works and the ways in which it can be tuned.

Besides the basic mechanisms of garbage collection, one of the most important points to understand about garbage collection in Java is that it is non-deterministic, and there is no way to predict when garbage collection will occur at run time. It is possible to include a hint in the code to run the garbage collector with the System.gc() or Runtime.gc() methods, but they provide no guarantee that the garbage collector will actually run.

The best approach to tuning Java garbage collection is setting flags on the JVM. Flags can adjust the garbage collector to be used (e.g. Serial, G1, etc.), the initial and maximum size of the heap, the size of the heap sections (e.g. Young Generation, Old Generation), and more. The nature of the application being tuned is a good initial guide to settings. For example, the Parallel garbage collector is efficient but will frequently cause “stop the world” events, making it better suited for backend processing where long pauses for garbage collection are acceptable. On the other hand, the CMS garbage collector is designed to minimize pauses, making it ideal for GUI applications where responsiveness is important. Additional fine-tuning can be accomplished by changing the size of the heap or its sections and measuring garbage collection efficiency using a tool like jstat.

From <https://stackify.com/what-is-java-garbage-collection/>

Manipulators in C++ with Examples

- Difficulty Level : Medium
- Last Updated : 09 Jun, 2020

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

For example, if we want to print the hexadecimal value of 100 then we can print it as:

```
cout<<setbase(16)<<100
```

Types of Manipulators

There are various types of manipulators:

16. **Manipulators without arguments**: The most important manipulators defined by the **IOStream library** are provided below.

- **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
- **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
- **ends**: It is also defined in ostream and it inserts a null

character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.

- **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

Examples:

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
    istringstream str("           Programmer");
    string line;
    // Ignore all the whitespace in string
    // str before the first word.
    getline(str >> std::ws, line);

    // you can also write str>>ws
    // After printing the output it will automatically
    // write a new line in the output stream.
    cout << line << endl;

    // without flush, the output will be the same.
    cout << "only a test" << flush;

    // Use of ends Manipulator
    cout << "\na";

    // NULL character will be added in the Output
    cout << "b" << ends;
    cout << "c" << endl;

    return 0;
}
```

17. **Manipulators with Arguments:** Some of the manipulators are used with the argument like setw (20), setfill ('*'), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program.
    For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill ('*');
    - **Some important manipulators in *<iomanip>* are:**
18. **setw (val):** It is used to set the field width in output operations.
19. **setfill (c):** It is used to fill the character 'c' on output stream.
20. **setprecision (val):** It sets val as the new value for the precision of floating-point values.
21. **setbase(val):** It is used to set the numeric base value for numeric values.
22. **setiosflags(flag):** It is used to set the format flags specified by parameter mask.
23. **resetiosflags(m):** It is used to reset the format flags specified by parameter mask.
    - **Some important manipulators in <ios> are:**
24. **showpos:** It forces to show a positive sign on positive numbers.
25. **noshowpos:** It forces not to write a positive sign on positive numbers.
26. **showbase:** It indicates the numeric base of numeric values.
27. **uppercase:** It forces uppercase letters for numeric values.
28. **nouppercase:** It forces lowercase letters for numeric values.
29. **fixed:** It uses decimal notation for floating-point values.
30. **scientific:** It uses scientific floating-point notation.
31. **hex:** Read and write hexadecimal values for integers and it works same as the setbase(16).
32. **dec:** Read and write decimal values for integers i.e.

setbase(10).

33. **oct:** Read and write octal values for integers i.e. setbase(10).
34. **left:** It adjusts output to the left.
35. **right:** It adjusts output to the right.

Example:

```
#include <iomanip>
#include <iostream>
using namespace std;

int main()
{
    double A = 100;
    double B = 2001.5251;
    double C = 201455.2646;

    // We can use setbase(16) here instead of hex

    // formatting
    cout << hex << left << showbase << nouppercase;

    // actual printed part
    cout << (long long)A << endl;

    // We can use dec here instead of setbase(10)

    // formatting
    cout << setbase(10) << right << setw(15)
         << setfill('_') << showpos
         << fixed << setprecision(2);

    // actual printed part
    cout << B << endl;

    // formatting
    cout << scientific << uppercase
         << noshowpos << setprecision(9);

    // actual printed part
    cout << C << endl;
}
```

Output:

```
0x64
_______+2001.53
2.014552646E+05
```

From <https://www.geeksforgeeks.org/manipulators-in-c-with-examples/>

Java Program to Use finally block for Catching Exceptions

- Difficulty Level : Hard
- Last Updated : 27 Nov, 2020

The **finally block** in java is used to put important codes such as clean up code e.g. closing the file or closing the connection. The finally block executes whether exception rise or not and whether exception handled or not. A finally contains all the crucial statements regardless of the exception occurs or not.

There are 3 possible cases where finally block can be used:

Case 1: When an exception does not rise

In this case, the program runs fine without throwing any exception and finally block execute after the try block.

- Java

```
// Java program to demonstrate
// finally block in java When
// exception does not rise

import java.io.*;

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

```
        try {
            System.out.println("inside try block");

            // Not throw any exception
            System.out.println(34 / 2);
        }

        // Not execute in this case
        catch (ArithmeticException e) {

            System.out.println("Arithmetic Exception");

        }
        // Always execute
        finally {

            System.out.println(
                "finally : i execute always.");

        }
    }
}
```

Output

```
inside try block
17
finally : i execute always.
```

Case 2: When the exception rises and handled by the catch block

In this case, the program throws an exception but handled by the catch block, and finally block executes after the catch block.

- Java

```
// Java program to demonstrate finally block in java
// When exception rise and handled by catch

import java.io.*;

class GFG {

    public static void main(String[] args)
    {
        try {
            System.out.println("inside try block");

            // Throw an Arithmetic exception
            System.out.println(34 / 0);
        }

        // catch an Arithmetic exception
        catch (ArithmeticException e) {

            System.out.println(
                "catch : exception handled.");
        }

        // Always execute
        finally {

          System.out.println("finally : i execute
always.");
        }
    }
}
```

Output

```
inside try block
catch : exception handled.
finally : i execute always.
```

Case 3: When exception rise and not handled by the catch block

In this case, the program throws an exception but not handled by catch so finally block execute after the try block and after the execution of finally block program terminate abnormally, But finally block execute fine.

- Java

```
// Java program to demonstrate finally block
// When exception rise and not handled by catch

import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        try {
            System.out.println("Inside try block");

            // Throw an Arithmetic exception
            System.out.println(34 / 0);
        }

        // Can not accept Arithmetic type exception
        // Only accept Null Pointer type Exception
        catch (NullPointerException e) {

            System.out.println(
                "catch : exception not handled.");
        }

        // Always execute
        finally {

            System.out.println(
                "finally : i will execute always.");
        }
        // This will not execute
        System.out.println("i want to run");
    }
}
```

Output

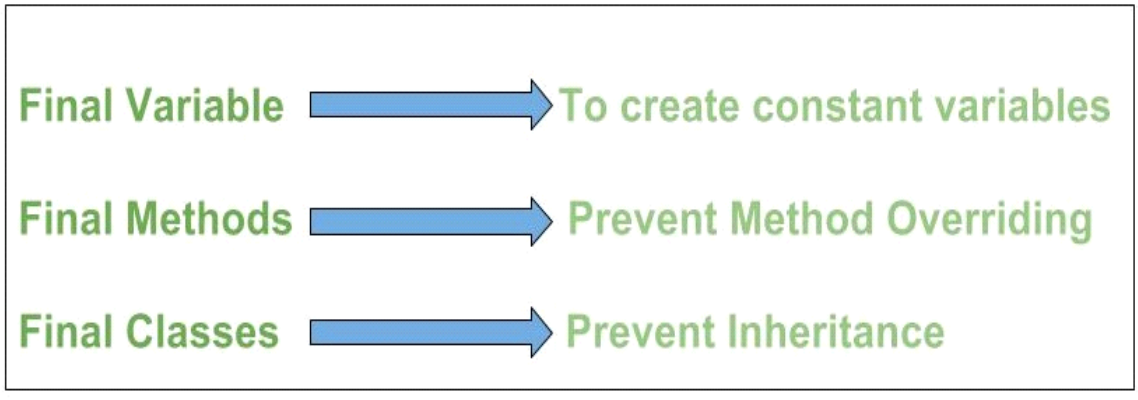
```
Inside try block
finally : i will execute always.
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at GFG.main(File.java:10)
```

From <https://www.geeksforgeeks.org/java-program-to-use-finally-block-for-catching-exceptions/>

final keyword in java

- Difficulty Level : Easy
- Last Updated : 17 Feb, 2021

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable **only to a variable, a method or a class**.Following are different contexts where final is used.



Final variables

When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object

pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Examples :

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a  blank final static  variable
static final double PI;
```

Initializing a final variable :

We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable :

36. You can initialize a final variable when it is declared.This approach is the most common. A final variable is called **blank final variable**,if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
37. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
38. A blank final static variable can be initialized inside static block.

Let us see above different ways of initializing a final variable through an example.

```
//Java program to demonstrate different
// ways of initializing a final variable

class Gfg
{
    // a final variable
    // direct initialize
    final int THRESHOLD = 5;

    // a blank final variable
    final int CAPACITY;

    // another blank final variable
    final int  MINIMUM;

    // a final static variable PI
    // direct initialize
    static final double PI = 3.141592653589793;

    // a  blank final static  variable
    static final double EULERCONSTANT;

    // instance initializer block for
    // initializing CAPACITY
    {
        CAPACITY = 25;
    }

    // static initializer block for
    // initializing EULERCONSTANT
    static{
        EULERCONSTANT = 2.3;
    }

    // constructor for initializing MINIMUM
    // Note that if there are more than one
    // constructor, you must initialize MINIMUM
    // in them also
    public GFG()
    {
        MINIMUM = -1;
```

```
    }

}
```

When to use a final variable :

The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

Reference final variable :

When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like

```
final StringBuffer sb;
```

As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called *non-transitivity*. To understand what is mean by internal state of the object, see below example :

```
// Java program to demonstrate
// reference final variable

class Gfg
{
    public static void main(String[] args)
    {
        // a final reference variable sb
        final StringBuilder sb = new
StringBuilder("Geeks");

        System.out.println(sb);

        // changing internal state of object
        // reference by final reference variable sb
        sb.append("ForGeeks");

        System.out.println(sb);
    }
}
```

Output:

```
Geeks
GeeksForGeeks
```

The *non-transitivity* property also applies to arrays, because arrays are objects in java. Arrays with final keyword are also called final arrays.

Note :

39. As discussed above, a final variable cannot be reassign, doing it will throw compile-time error.

```
// Java program to demonstrate re-assigning
// final variable will throw compile-time error

class Gfg
{
    static final int CAPACITY = 4;

    public static void main(String args[])
    {
        // re-assigning final variable
        // will throw compile-time error
        CAPACITY = 5;
    }
}
```

Output

```
Compiler Error: cannot assign a value to final
variable CAPACITY
```

40. When a final variable is created inside a method/constructor/block, it is called local final variable, and it

must initialize once where it is created. See below program for local final variable

```
// Java program to demonstrate
// local final variable

// The following program compiles and runs fine

class Gfg
{
    public static void main(String args[])
    {
        // local final variable
        final int i;
        i = 20;
        System.out.println(i);
    }
}
```

Output:

```
20
```

41. Note the difference between C++ *const* variables and Java *final* variables. const variables in C++ must be assigned a value when declared. For final variables in Java, it is not necessary as we see in above examples. A final variable can be assigned value later, but only once.
42. *final* with foreach loop : final with for-each statement is a legal statement.

```
// Java program to demonstrate final
// with for-each statement

class Gfg
{
    public static void main(String[] args)
    {
        int arr[] = {1, 2, 3};

        // final with for-each statement
        // legal statement
        for (final int i : arr)
            System.out.print(i + " ");
    }
}
```

Output:

```
1 2 3
```

Explanation : Since the i variable goes out of scope with each iteration of the loop, it is actually re-declaration each iteration, allowing the same token (i.e. i) to be used to represent multiple variables.

Final classes

When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class :

43. One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer,Float etc. are final classes. We can not extend them.

```
final class A
{
     // methods and fields
}
// The following class is illegal.
class B extends A
{
    // COMPILE-ERROR! Can't subclass A
}
```

44. The other use of final with classes is to create an immutable class like the predefined String class.You can not make a class immutable without making it final.

Final methods

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden.
The Object class does this—a number of its methods are final.We must declare methods with final keyword for which we

required to follow the same implementation throughout all the derived classes. The following fragment illustrates final keyword with a method:

```
class A
{
    final void m1()
    {
        System.out.println("This is a final
method.");
    }
}
class B extends A
{
    void m1()
    {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

For more examples and behavior of final methods and final classes, please see Using final with inheritance

From <https://www.geeksforgeeks.org/final-keyword-java/>

What Is an Exception?

The term *exception* is shorthand for the phrase "exceptional event."

> **Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

From <https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

Exceptions and errors both are subclasses of Throwable class. The error indicates a problem that mainly occurs due to the lack of system resources and our application should not catch these types of problems. Some of the examples of errors are system crash error and out of memory error. Errors mostly occur at runtime that's they belong to an unchecked type.
Exceptions are the problems which can occur at runtime and compile time. It mainly occurs in the code written by the developers. Exceptions are divided into two categories such as checked exceptions and unchecked exceptions.

Sr. No.	Key	Error	Exception
1	Type	Classified as an unchecked type	Classified as checked and unchecked
2	Package	It belongs to java.lang.error	It belongs to java.lang.Exception
3	Recoverable/ Irrecoverable	It is irrecoverable	It is recoverable
4		It can't be occur at compile time	It can occur at run time compile time both
5	Example	OutOfMemoryError , IOError	NullPointerException , SqlException

Example of Error

```
public class ErrorExample {
  public static void main(String[] args){
    recursiveMethod(10)
  }
  public static void recursiveMethod(int i){
    while(i!=0){
      i=i+1;
      recursiveMethod(i);
    }
  }
}
```

Output

```
Exception in thread "main" java.lang.StackOverflowError
  at ErrorExample.ErrorExample(Main.java:42)
```

Example of Exception

```
public class ExceptionExample {
  public static void main(String[] args){
    int x = 100;
```

```
    int y = 0;
    int z = x / y;
  }
}
```

Output

```
java.lang.ArithmeticException: / by zero
  at ExceptionExample.main(ExceptionExample.java:7)
```

From <https://www.tutorialspoint.com/difference-between-exception-and-error-in-java>

Exception Handling in Java

1. Exception Handling
2. Advantage of Exception Handling
3. Hierarchy of Exception classes
4. Types of Exception
5. Exception Example
6. Scenarios where an exception may occur

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition.
In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
Features of Java - Javatpoint

What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

45. statement 1;
46. statement 2;
47. statement 3;
48. statement 4;
49. statement 5;//exception occurs
50. statement 6;
51. statement 7;
52. statement 8;
53. statement 9;
54. statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

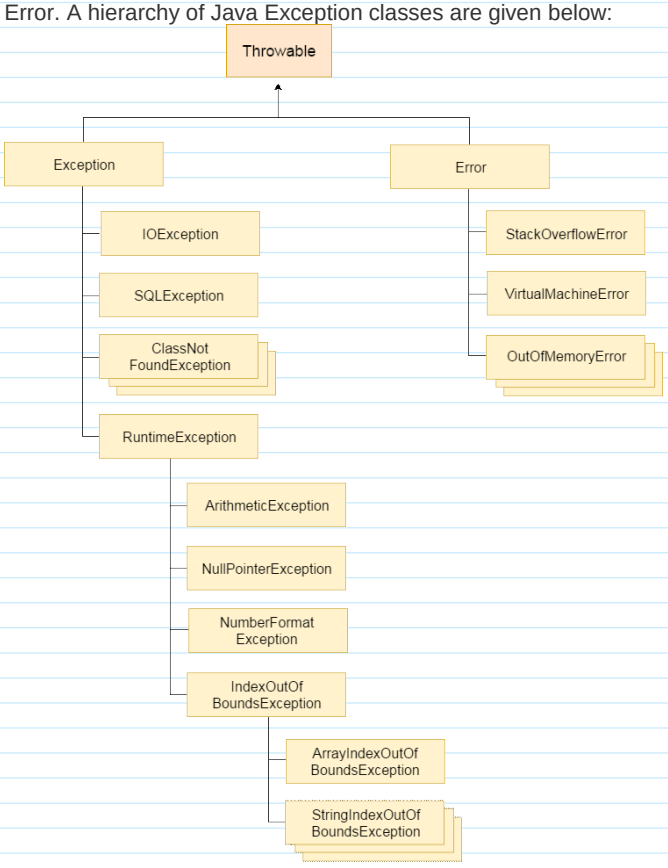
Do You Know?

- What is the difference between checked and unchecked exceptions?
- What happens behind the code int data=50/0;?
- Why use multiple catch block?
- Is there any possibility when finally block is not executed?
- What is exception propagation?
- What is the difference between throw and throws keyword?
- What are the 4 rules for using exception handling with method overriding?

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and

Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions

e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```
55. public class JavaExceptionExample{
56.   public static void main(String args[]){
57.    try{
58.       //code that may raise exception
59.       int data=100/0;
60.    }catch(ArithmeticException e){System.out.println(e);}
61.    //rest code of the program
62.    System.out.println("rest of the code...");
63.   }
64. }
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
65. int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
66. String s=null;
67. System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur

NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exceptions Index

1. Java Try-Catch Block
2. Java Multiple Catch Block
3. Java Nested Try
4. Java Finally Block
5. Java Throw Keyword
6. Java Exception Propagation
7. Java Throws Keyword
8. Java Throw vs Throws
9. Java Final vs Finally vs Finalize
10. Java Exception Handling with Method Overriding
11. Java Custom Exceptions

From <https://www.javatpoint.com/exception-handling-in-java>