



Programming in Java

Unit II



Unit II

Inheritance: Basic concepts - Types of inheritance - Member access rules - Usage of this and Super key word - Method Overloading - Method overriding - Abstract classes - Dynamic method dispatch - Usage of final keyword.

Packages: Definition- Access Protection –Importing Packages.

Interfaces: Definition–Implementation–Extending Interfaces.

Exception Handling: try – catch - throw - throws – finally – Built-in exceptions - Creating own Exception classes

Inheritance - Introduction

- It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones.
- In the terminology of Java, a class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Need for Inheritance

- Inheritance, a key feature of OOP provides us with following :
- Code reusability – Code written in super class is common for all the subclasses.
- Method Overriding – It is achievable only through Inheritance where the runtime polymorphism is achieved.
- Abstraction – The concept of abstract is achieved through inheritance.

Basic Concepts/Terminologies in Inheritance

- Class : a set of objects that shares common characteristics and properties. A class is a template or blueprint or prototype from which objects are created.
- Super Class/ Parent Class : The class whose features are inherited
- Sub Class/Child Class : The class that inherits other class.
- Syntax:

```
Class Subclass extends Superclass
{
    // fields and methods
}
```

Simple Program

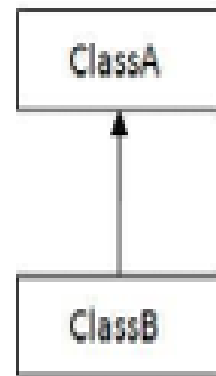
```
class Employee {  
    float salary=40000;  
}  
  
class Programmer extends Employee {  
    int bonus=10000;  
  
    public static void main(String args[]) {  
        Programmer p=new Programmer();  
  
        System.out.println("Programmer salary is:"+p.salary);  
        System.out.println("Bonus of Programmer is:"+p.bonus);  
    } }  
}
```

Output:

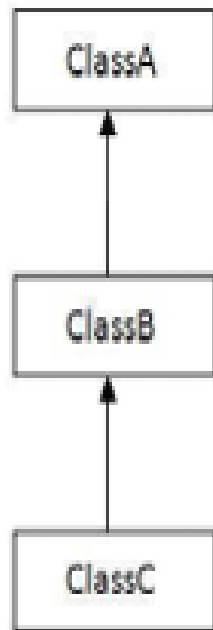
Programmer salary is:40000.0
Bonus of programmer
is:10000

Types of Inheritance

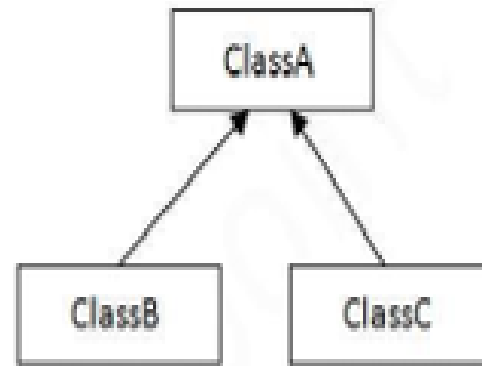
- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance



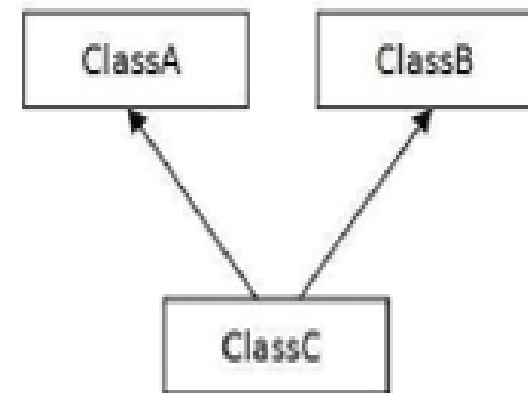
1) Single



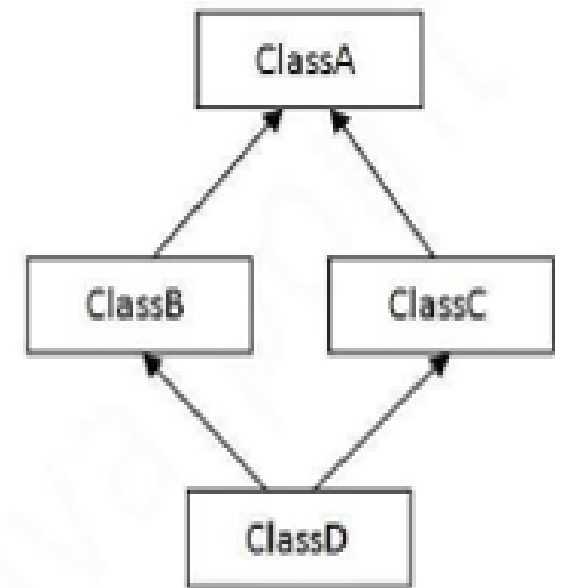
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

Single Inheritance

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

Output:

barking...eating...

Multilevel Inheritance

```
class Animal {  
    void eat() {  
        System.out.println("eating...");  
    }  
    class Dog extends Animal  
    {  
        void bark() {  
            System.out.println("barking...");  
        }  
        class BabyDog extends Dog  
        {  
            void weep()  
            {  
                System.out.println("weeping...");  
            }  
        }  
    }  
}
```

```
class TestInheritance2  
{  
    public static void main(String args[])  
    {  
        BabyDog d=new BabyDog();  
        d.weep();  
        d.bark();  
        d.eat();  
    }  
}
```

Output:

weeping...barking... eating...

Hierarchical Inheritance

```
class Animal
```

```
{
```

```
void eat()
```

```
{
```

```
System.out.println("eating...");
```

```
}
```

```
}
```

```
class Dog extends Animal
```

```
{
```

```
void bark()
```

```
{
```

```
System.out.println("barking...");
```

```
}
```

```
}
```

```
class Cat extends Animal
```

```
{
```

```
void meow()
```

```
{
```

```
System.out.println("meowing...");
```

```
}
```

```
}
```

```
class TestInheritance3
```

```
{
```

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

```
{
```

```
Cat c=new Cat();
```

```
c.meow();
```

```
c.eat();
```

```
}
```

```
}
```

Output:

meowing...eating...

Member Access Rule

- Often an instance variable of a class will be declared private to prevent its unauthorized use or tampering.
- Inheriting a class does not overrule the private access restriction.
- Thus, even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared private.

```
// Private members are not inherited.

// This example will not compile.

// A class for two-dimensional objects.
class TwoDShape {

    private double width; // these are
    private double height; // now private

    void showDim() {
        System.out.println("Width and height are " +
                           width + " and " + height);
    }
}

// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    String style;

    double area() {
        return width * height / 2; // Error! can't access
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

Can't access a **private** member of a superclass.

this keyword

- The 'this' keyword refers to the current object in a method or constructor.
- The most common use of the 'this' keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).
- this can also be used to:
 - Invoke current class constructor
 - Invoke current class method
 - Return the current class object
 - Pass an argument in the method call
 - Pass an argument in the constructor call

'this' - example

```
public class Main
{
    int x;
    // Constructor with a parameter
    public Main(int x)
    {
        this.x = x;
    }
    // Call the constructor
    public static void main(String[] args)
    {
        Main myObj = new Main(5);
        System.out.println("Value of x = " + myObj.x);
    }
}
```

Output:
Value of x = 5

super keyword

- The super keyword refers to superclass (parent) objects.
- It is used to call superclass methods, and to access the superclass constructor.
- The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

‘super’ keyword

```
class Animal
{
    // Superclass (parent)
    public void animalSound()
    {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal
{
    // Subclass (child)
    public void animalSound()
    {
        super.animalSound(); // Call the superclass method
        System.out.println("The dog says: bow wow");
    }
}

public class Main
{
    public static void main(String args[])
    {
        Animal myDog = new Dog(); // Create a Dog object
        myDog.animalSound(); // Call the method on the Dog object
    }
}
```

Output:
The animal makes a sound
The dog says: bow wow

Method Overloading

- In Java, Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.
- Method overloading in Java is also known as Compile-time Polymorphism, *Static Polymorphism*, or Early binding.
- In Method overloading compared to the parent argument, the child argument will get the highest priority.

Method Overloading - Example

```
public class Sum {  
    public int sum(int x, int y)  
    {  
        return (x + y);  
    }  
    public int sum(int x, int y, int z)  
    {  
        return (x + y + z);  
    }  
    public double sum(double x, double y)  
    {  
        return (x + y);  
    }  
    public static void main(String args[])  
    {  
        Sum s = new Sum();  
        System.out.println(s.sum(10, 20));  
        System.out.println(s.sum(10, 20, 30));  
        System.out.println(s.sum(10.5, 20.5));  
    }  
}
```

Output

30

60

31.0

Different Ways of Method Overloading in Java

- Changing the Number of Parameters.
- Changing Data Types of the Arguments.
- Changing the Order of the Parameters of Methods

Method Overriding

- Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
- Method overriding is one of the way by which java achieve Run Time Polymorphism.
- **Usage of Java Method Overriding**
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.

Rules for Java Method Overriding

- The method must have the same name as in the parent class.
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

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

class Bike2 extends Vehicle
{
    void run()
    {
        System.out.println("Bike is running safely");
    }

    public static void main(String args[])
    {
        Bike2 obj = new Bike2();//creating object
        obj.run();//calling method
    } }
```

OUTPUT

Bike is running safely

Method Overloading Vs. Method Overriding

No.	Method Overloading	Method Overriding
1.	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2.	Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3.	In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
4.	Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
5.	Return type can be same or different in method overloading	Return type must be same or covariant in method overriding.

Abstract Classes

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either abstract classes or interfaces.
- The abstract keyword is a non-access modifier, used for classes and methods:
 - **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
 - **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Rules for Java Abstract class



1

An abstract class must be declared with an abstract keyword.

2

It can have abstract and non-abstract methods.

3

It cannot be instantiated.

4

It can have final methods

5

It can have constructors and static methods also.

Syntax:

```
public abstract class Shape {  
    // Abstract method  
    public abstract double area();  
    // Concrete method  
    public void display() {  
        System.out.println("This is a shape.");  
    }  
}
```

Example:

```
abstract class Bank{  
    abstract int getRateOfInterest();  
}
```

```
class SBI extends Bank{  
    int getRateOfInterest()  
    {return 7;}  
}
```

```
class PNB extends Bank{  
    int getRateOfInterest()  
    {return 8;}  
}
```

```
class TestBank{  
    public static void main(String args[]){  
        Bank b;  
        b=new SBI();  
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");  
        b=new PNB();  
        System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");  
    }  
}
```

Output:

Rate of Interest is: 7 %

Rate of Interest is: 8 %

Dynamic method dispatch

- Dynamic method dispatch or run-time polymorphism is the mechanism through which the correct version of an overridden method is called at runtime.
- When a subclass overrides a method from its superclass, the overridden method in the subclass is executed when called on an instance of the subclass, even if the reference to the object is of the superclass type.

Example:

```
class Animal {  
    void makeSound() {  
        System.out.println("Generic Animal Sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void makeSound() {  
        System.out.println("Bark");  
    }  
}  
  
class Cat extends Animal {  
    @Override  
    void makeSound() {  
        System.out.println("Meow");  
    }  
}  
  
public class DynamicMethod {  
    public static void main(String[] args) {  
        Animal[] animals = {new Dog(), new Cat()};  
        for (Animal animal : animals) {  
            animal.makeSound();  
        }  
    }  
}
```

Output:

Bark
Meow

- In the Main class, we have created an array of Animal objects called animals and populate it with an instance of Dog and an instance of Cat.
- We then iterate through the animals array using a for-each loop and call the makeSound() method on each element.
- Due to dynamic method dispatch, the appropriate version of makeSound() from either Dog or Cat will be executed based on the actual type of the object.

Usage of final keyword

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
 - variable
 - method
 - class

Final Keyword	Description	Program	Output
Java final variable	If you make any variable as final, you cannot change the value of final variable(It will be constant).	<pre> class Bike9 { final int speedlimit=90; // final variable void run() { speedlimit=400; } public static void main(String args[]) { Bike9 obj=new Bike9(); obj.run(); } } //end of class </pre>	<p>Compilation Error</p> <p>Main.java:4: error: cannot assign a value to final variable speedlimit</p> <pre> speedlimit=400; ^ 1 error </pre>
Java final method	If you make any method as final, you cannot override it	<pre> class Bike { final void run() {System.out.println("running");} } class Honda extends Bike { void run() {System.out.println("running safely with 100kmph");} public static void main(String args[]) { Honda honda= new Honda(); honda.run(); } } </pre>	<p>Compilation Error</p> <p>Main.java:6: error: run() in Main cannot override run() in Bike</p> <pre> void run() {System.out.println("running safely with 100kmph");} ^ overridden method is final 1 error </pre>

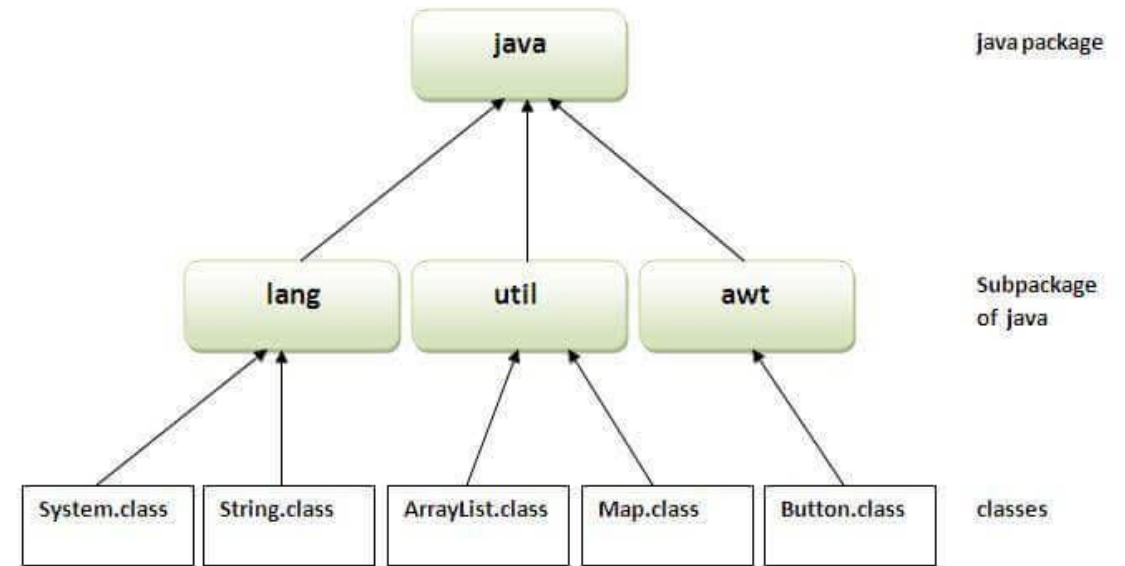
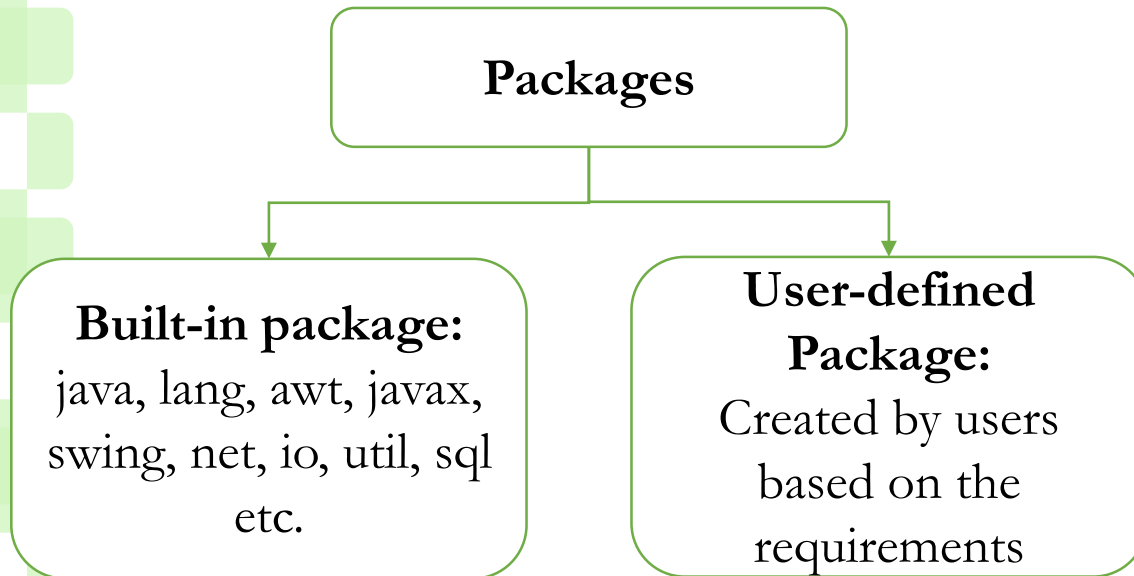
Final Keyword	Description	Program	Output
Java final class	If you make any class as final, you cannot extend it.	<pre> final class Bike {} class Honda1 extends Bike { void run() {System.out.println("running safely with 100kmph");} public static void main(String args[]) { Honda1 honda= new Honda1(); honda.run(); } } </pre>	<p>Compilation Error</p> <p>Main.java:3: error: cannot inherit from final Bike</p> <p>class Main extends Bike {</p> <p>^</p> <p>1 error</p>



Packages

Packages - Definitions

- A **java package** is a group of similar types of classes, interfaces and sub-packages.



Simple Package program

Syntax to create a package:

```
package packagename;
```

```
//save as Simple.java
```

```
package mypack;
```

```
public class Simple{
```

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

```
        System.out.println("Welcome to package");
```

```
    }
```

```
}
```

To compile java package:(without IDE)

```
javac -d directory javafilename
```

Example:

```
javac -d Simple.java
```

To run java package:

```
java packagename.javafilename
```

Example:

```
java mypack.Simple
```

The -d switch specifies the destination where to put the generated class file.

Access Protection

- In java, the access modifiers define the accessibility of the class and its members.
- Java has four access modifiers, and they are default, private, protected, and public.
- In java, the package is a container of classes, sub-classes, interfaces, and sub-packages.
- The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages.
- In java, the accessibility of the members of a class or interface depends on its access specifiers.

Access Protection

Access Specifier \ Accessibility Location	Same Class	Same Package		Other Package	
		Child Class	Non-Child Class	Child Class	Non- Child Class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

- The **public** members can be accessed everywhere.
- The **private** members can be accessed only inside the same class.
- The **protected** members are accessible to every child class (same package or other packages).
- The **default** members are accessible within the same package but not outside the package.

Access Protection : Example Program

```
class ParentClass {
    int a = 10;
    int b = 20;
    protected int c = 30;
    private int d = 40;
    void showData()
    {
        System.out.println("Inside ParentClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}

class ChildClass extends ParentClass {
    void accessData() {
        System.out.println("Inside ChildClass");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        //System.out.println("d = " + d); // private member can't be accessed
    }
}

public class AccessModifiersExample {
    public static void main(String[] args) {
        ChildClass obj = new ChildClass();
        obj.showData();
        obj.accessData();
    }
}
```

Output:

Inside ParentClass

a=10

b=20

c=30

d=40

Inside Child Class

a=10

b=20

c=30

Importing Packages

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. fully qualified name.

1. import package.*;

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java  
package pack;  
public class A{  
    public void msg() {System.out.println("Hello");}  
}
```

```
//save by B.java  
package mypack;  
import pack.*;  
  
class B{  
    public static void main(String args[]) {  
        A obj = new A();  
        obj.msg();  
    }  
}
```

Output:Hello

2. import package.classname;

- If you import package.classname then only declared class of this package will be accessible.

//save by A.java

```
package pack;  
public class A{  
    public void msg(){System.out.println("Hello");}  
}
```

//save by B.java

```
package mypack;  
import pack.A;  
  
class B{  
    public static void main(String args[]){  
        A obj = new A();  
        obj.msg();  
    }  
}
```

Output:Hello

3. fully qualified name.

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

Output:Hello



Interfaces



Interfaces

- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
 - Interfaces specify what a class must do and not how. It is the blueprint of the class.
 - An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) `move()`. So it specifies a set of methods that the class has to implement.
 - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
 - It is used to achieve abstraction and multiple inheritance in Java.
 - To declare an interface, use `interface` keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default.

Use of Java Interface

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes? The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

Syntax:

```
interface <interface_name>
{
    // declare constant fields
    // declare methods that
    //abstract by default.
}
```

Example:

```
public interface A1{
    public int a = 4, b = 5;
    public double f = 5.5, d = 4.5;
}
class B implements A1{
    public void add1(){
        int i;
        i = a + b;
        System.out.println("Addition of two
integer number 'a + b' :"+ i);
    }
}
```

```
class C implements A1{
    public void add2(){
        double s;
        s = f + d;
        System.out.println("Addition of two
float number 'f + d' :"+ s);
    }
}
public static void main(String[] args) {
    B in = new B();
    C ij = new C();
    in.add1();
    ij.add2();
}
```

output:

Addition of two integer number 'a + b' :9

Addition of two float number 'f + d' :10.0

Extending Interfaces

- An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class.
- An interface extends another interface like a class implements an interface in interface inheritance.

Example Program

```
interface A {  
    void funcA();  
}  
interface B extends A {  
    void funcB();  
}  
class C implements B {  
    public void funcA() {  
        System.out.println("This is funcA");  
    }  
    public void funcB() {  
        System.out.println("This is funcB");  
    }  
}  
public class Demo {  
    public static void main(String args[]) {  
        C obj = new C();  
        obj.funcA();  
        obj.funcB();  
    }  
}
```

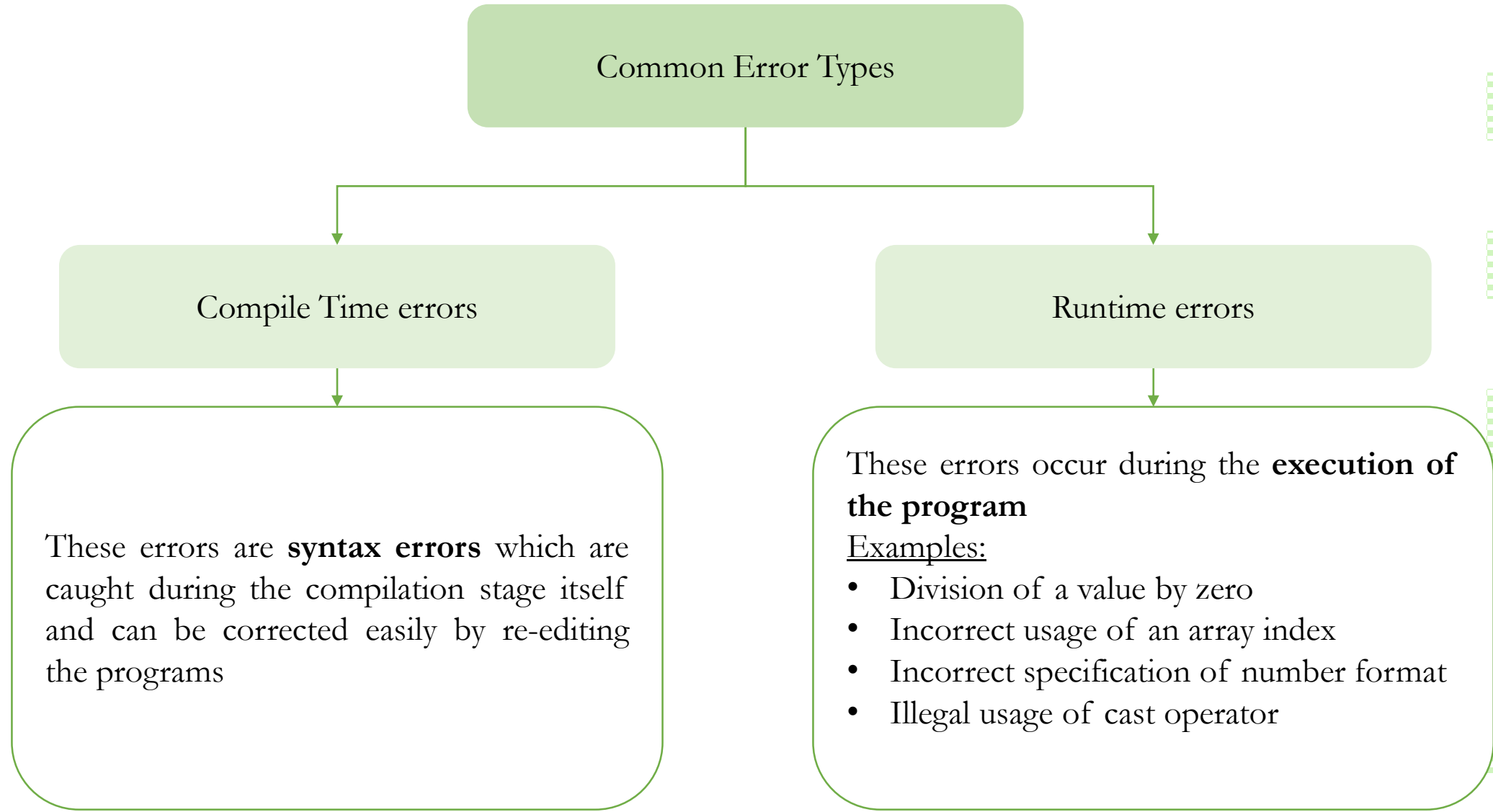
Output:

This is funcA

This is funcB

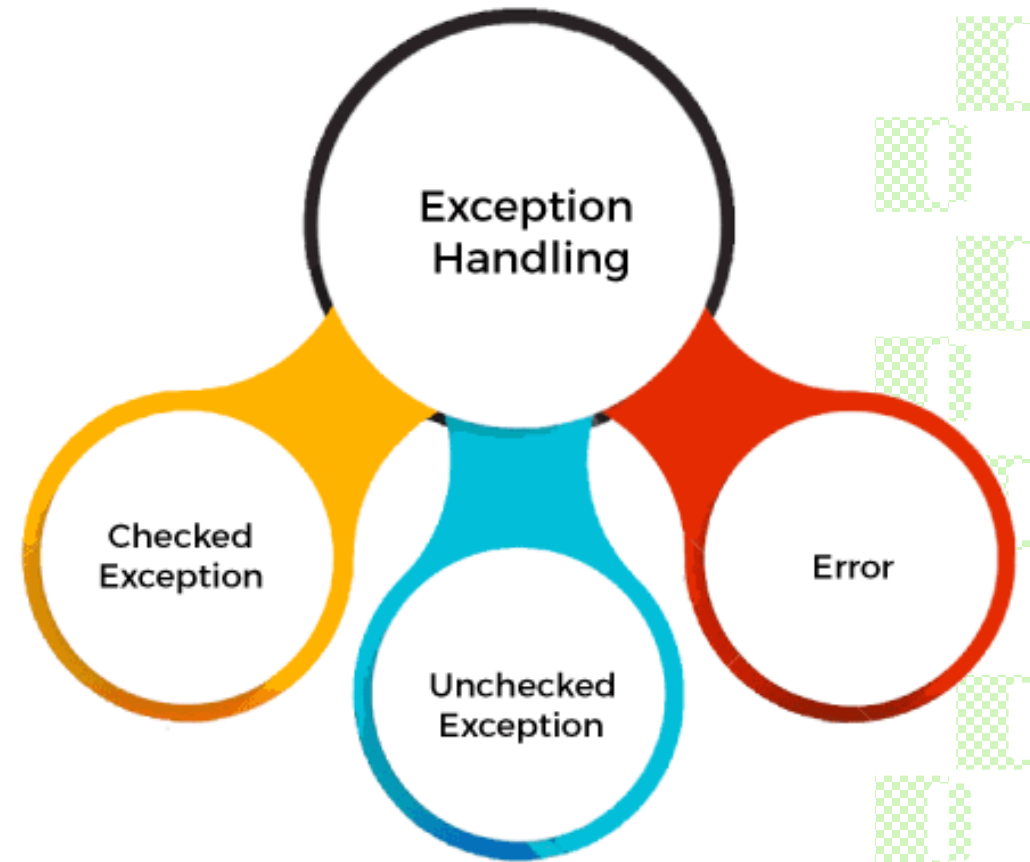
Exception Handling

- In Java, an exception is an event that disrupts the normal flow of the program.
- It is an object which is thrown at runtime.
- The **Exception Handling** in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.
- Errors such as `ClassNotFoundException`, `IOException`, `SQLException`, `RemoteException`, etc. are handled




Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked.
- An error is considered as the unchecked exception.
- However, according to Oracle, there are three types of exceptions namely:
 - Checked Exception
 - Unchecked Exception
 - Error




Difference between Checked, Unchecked and Error



Checked Exception

Checked

- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.
- For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.



Unchecked Exception

Unchecked:

- The classes that inherit the RuntimeException are known as unchecked exceptions.
- For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

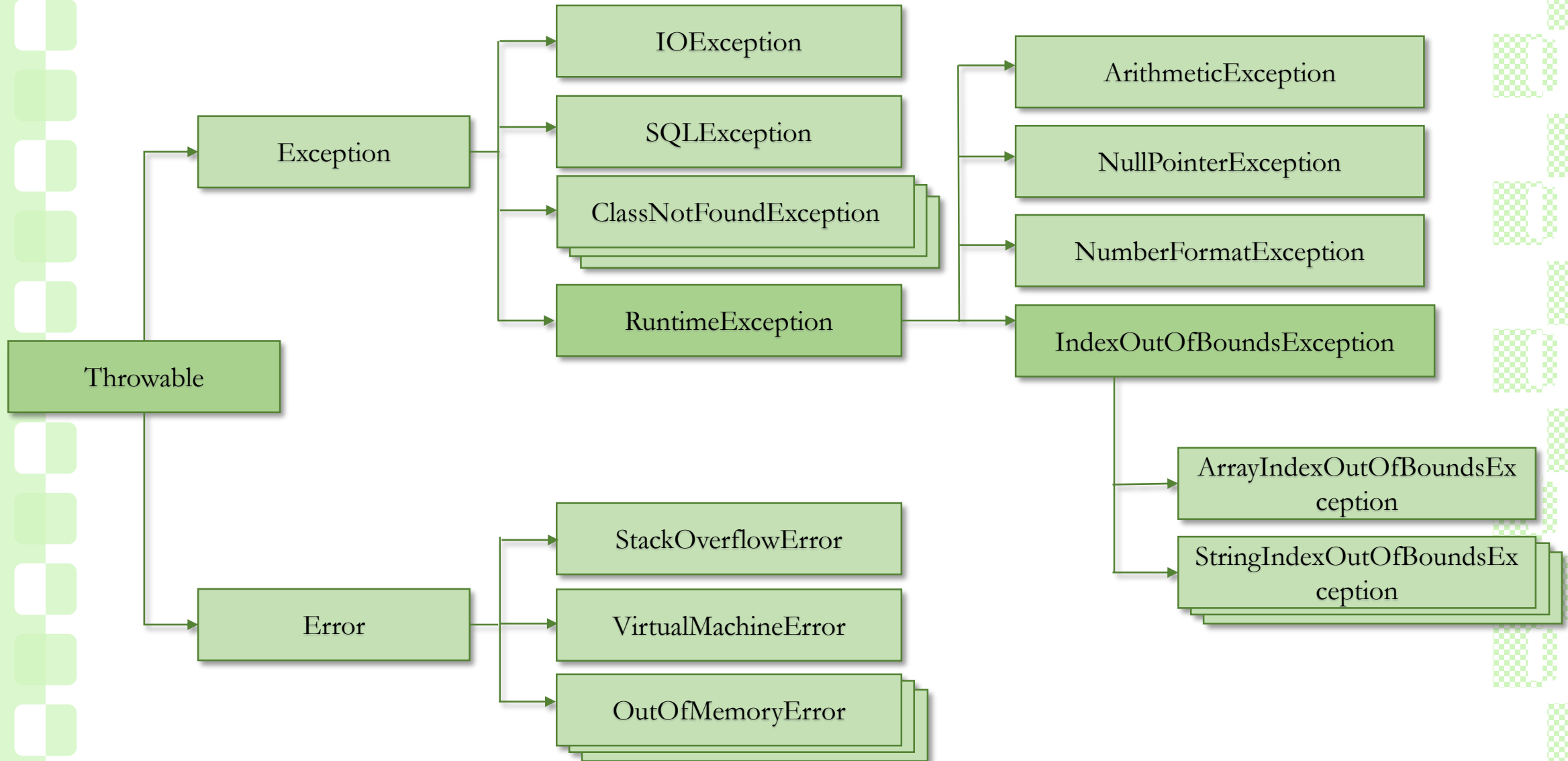


Error

Error:

- Error is irrecoverable.
- Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Hierarchy of Java Exception Handling



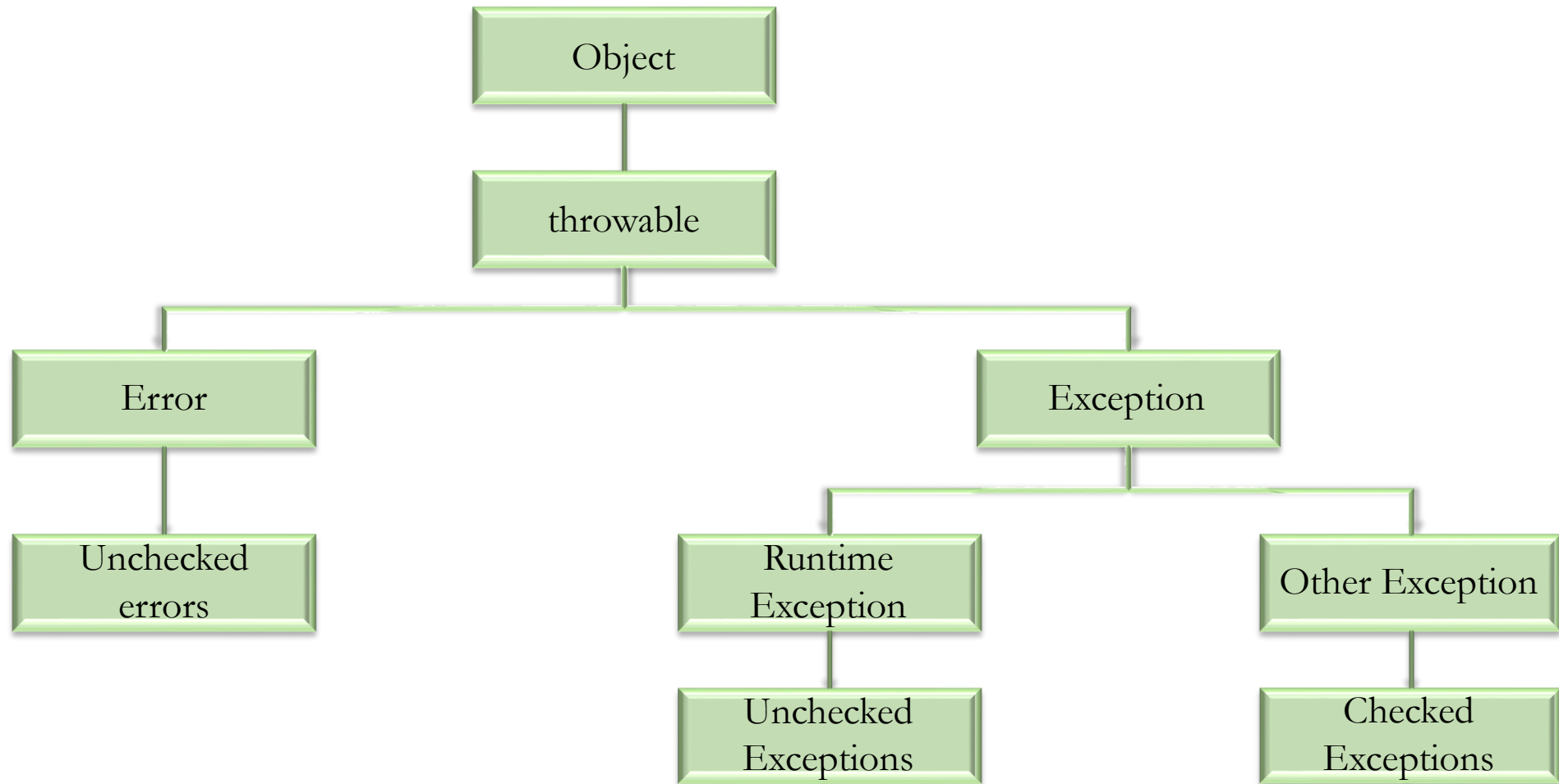
Details of Hierarchy:

- The **Throwable** class is the root class of all error classes.
- Two immediate subclasses of this class are **Exception class** and **Error class**
- The subclasses of the **Exception class** (Ex. **Arithmetic exception** and **NumberFormatException**) describe the exceptions that should be caught by the user programs
- The subclasses of the **error class** describe the conditions (**stack overflow, memory exhaustion and illegal access**) that should not be caught by the user programs.

Constructors and Methods of Throwable class

- Two **important constructors** of **Throwable** class
 - Throwable()
 - Throwable(String message)
- Two **important methods** of **Throwable** class
 - getMessage()
 - printStackTrace()
- Two **important constructors** of **Exception** class
 - Exception()
 - Exception(String message)

Throwable Class



Error Classes

Name of the class	Description of the Error Condition
ClassNotFoundException	A class is not found
InstantiationException	An attempt was made to instantiate an interface or an abstract class
Run-timeException	A run-time error condition has occurred
ArrayIndexOutOfBoundsException	A non-existent array position is accessed through a wrong index value
ArithmeticException	An arithmetic error, such as division-by-zero, has occurred
ClassCastException	An invalid casting is done
IOException	An input/output related error has occurred
EOFException	The 'end-of-file' condition has been encountered
FileNotFoundException	Specified file could not be located
NullPointerException	A reference to a non-existent object is made
NumberFormatException	A wrong conversion is made between a string and a numerical value

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

The try-catch block

- One of the primary mechanisms for handling exceptions in Java is the try-catch block.
- The try block contains the code that may throw an exception, and the catch block is used to handle the exception if it occurs.

- To handle multiple exceptions, we can use multiple catch blocks that allows to tailor the exception handling logic on some specific types of exceptions thrown.

Syntax:

```
try {  
    // Code that may throw an exception  
} catch (ExceptionType e) {  
    // Exception handling code  
}
```

Syntax:

```
try {  
    // Code that may throw an exception  
} catch (IOException e) {  
    // Handle IOException  
} catch (NumberFormatException e) {  
    // Handle NumberFormatException  
} catch (Exception e) {  
    // Handle any other exceptions  
}
```

finally block

- In addition to try and catch, Java also provides a finally block, which allows you to execute cleanup code, such as closing resources, regardless of whether an exception occurs or not.
- The finally block is typically used to release resources that were acquired in the try block.

Syntax:

```
try {  
    // Code that may throw an exception  
} catch (Exception e) {  
    // Exception handling code  
} finally {  
    // Cleanup code  
}
```

throw Statement

- JVM generated an Exception object when an error condition occurs in a program
- Similarly , user can generate an Exception object inside a catch block as shown below

```
catch ( ArithmeticException e)
{
    System.out.println("Arithmetic error has occurred");
    throw e;
}
```

Here **e** represents an Exception object of type `ArithmeticException`

The **throw e ;** statement generates the Exception object **e** once again.

- We shall also generate an **Exception** object within a try block

```
try
{
...
...
throw new ArrayIndexOutOfBoundsException();
...
...
}
```

The statements that follow a throw statement in the same catch block or try block will not be executed

throws clause

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception.
- So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Syntax of Java throws

```
return_type method_name() throws exception_class_name{  
    //method code  
}
```

//Program to illustrate 'throws' clause

```
import java.io.IOException;
class Testthrows1 {
    void m() throws IOException {
        throw new IOException("device error");//checked exception
    }
    void n() throws IOException {
        m();
    }
    void p() {
        try {
            n();
        } catch (Exception e) {System.out.println("exception handled");}
    }
    public static void main(String args[]) {
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:
exception handled
normal flow...

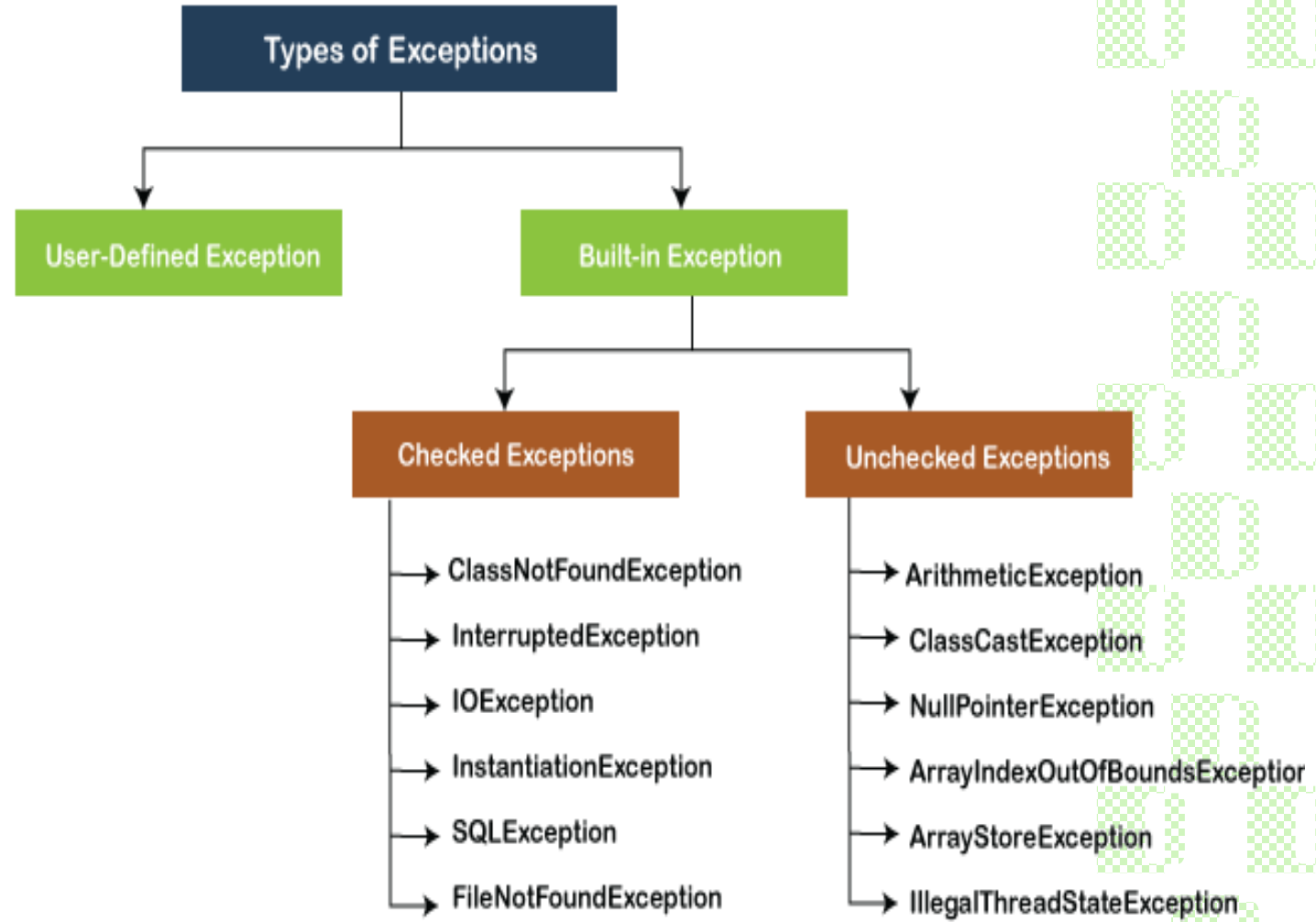
throw Vs. throws

Feature	throw	throws
Definition	It is used to explicitly throw an exception.	It is used to declare that a method might throw one or more exceptions.
Location	It is used inside a method or a block of code.	It is used in the method signature.
Usage	It can throw both checked and unchecked exceptions.	It is only used for checked exceptions. Unchecked exceptions do not require throws.
Responsibility	The method or block throws the exception.	The method's caller is responsible for handling the exception.
Flow of Execution	Stops the current flow of execution immediately.	It forces the caller to handle the declared exceptions.
Example	<code>throw new ArithmeticException("Error");</code>	<code>public void myMethod() throws IOException {}</code>

Types of Exceptions

Exceptions can be categorized into two ways:

- Built-in Exceptions
 - Checked Exception or Compile time error
 - Unchecked Exception or run time error
- User-Defined Exceptions



Built-in Exception

- The two types of Built-in Exceptions are
- Checked Exception or Compile time error
 - Unchecked Exception or run time error

Handling Checked Exception - FileNotFoundException

```
// Java Program to Illustrate Handling of Checked Exception
// Importing required classes
import java.io.*;
import java.util.*;
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
        throws FileNotFoundException
    {
        // Assigning null value to object of FileInputStream
        FileInputStream GFG = null;
        // Try block to check for exceptions
        try {
            // Giving path where file should exists to read
            // content
            GFG = new FileInputStream("/home/mayur/GFG.txt");
        }
        // Catch block to handle exceptions
        catch (FileNotFoundException e) {

            // Display message when exception occurs
            System.out.println("File does not exist");
        }
    }
}
```

OUTPUT:
File does not exist

Handling Unchecked Exception - ArrayIndexOutOfBoundsException

```
// Importing Classes/Files
import java.io.*;

public class GFG {
    // Main Driver Method
    public static void main(String[] args)
    {
        // Inserting elements into Array
        int a[] = { 1, 2, 3, 4, 5 };
        // Try block for exceptions
        try {
            // Forcefully trying to access and print
            // element/s beyond indexes of the array
            System.out.println(a[5]);
        }
        // Catch block for catching exceptions
        catch (ArrayIndexOutOfBoundsException e)
        {
            // Printing display message when index not
            // present in a array is accessed
            System.out.println("Out of index please check your code");
        }
    }
}
```

Out of index please check your code

User Defined Exception Handling

- You can create your own exceptions in Java.
 - All exceptions must be a child of Throwable.
 - If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the **Exception** class.
 - If you want to write a runtime exception, you need to extend the **RuntimeException** class.

Syntax

Syntax of Java try-catch

```
try{  
    //code that may throw an exception  
}catch(Exception_class_Name ref)  
{  
}
```

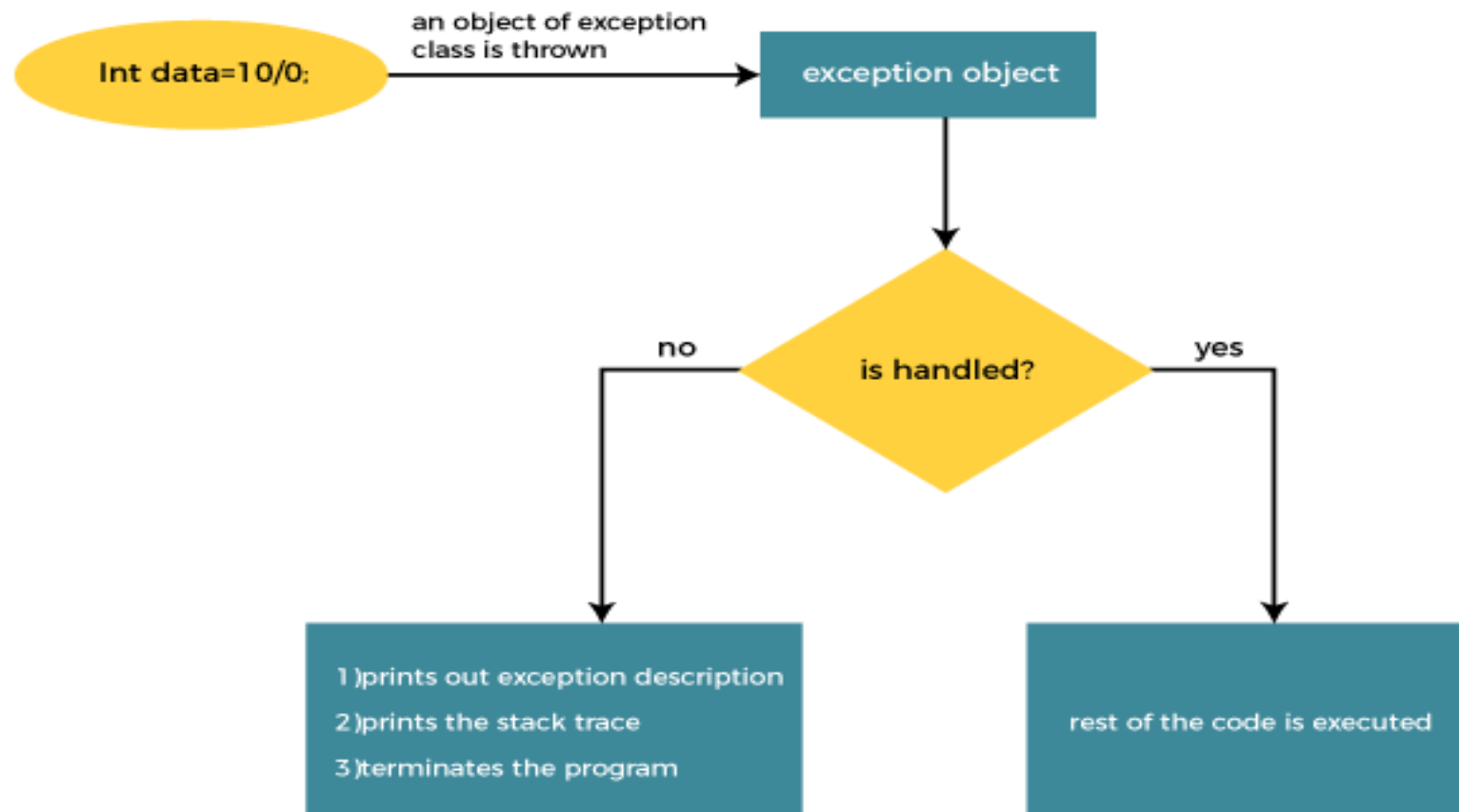
Java catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., Exception) or the generated exception type.
- However, the good approach is to declare the generated type of exception.

Syntax of try-finally block

```
try{  
    //code that may throw an exception  
}finally  
{  
}
```

Internal Working of Java try-catch block



User-Defined Exception - Example

```
class EmployeeException extends Exception {  
    public EmployeeException(String s) {  
        super(s);  
    }  
}  
  
class SampleEmp {  
    void empIDCheck(int EmpID) throws EmployeeException {  
        if (EmpID <= 0 || EmpID > 999) {  
            throw new EmployeeException("Invalid Employee ID");  
        }  
    }  
  
    public static void main(String args[]) {  
        SampleEmp emp = new SampleEmp();  
        try {  
            emp.empIDCheck(0);  
        }  
        catch (EmployeeException e) {  
            System.out.println("Exception caught");  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Result Size: 668 x 508

```
Exception caught  
Invalid Employee ID
```