UNIT – III

INTERFACES, PACKAGES, EXCEPTIONS

Interfaces: Introduction,Extending Interfaces, Interfaces Vs Abstract Classes.

Packages: Creating Packages, ImportingPackages, Access Protection, Exploring java.io and java.util Packages.

Exceptions: Introduction, Exception Handling Techniques-Try…Catch, Throw, Throws, Finally Block, User Defined Exceptions.

Interfaces:

An interface is a **blueprint** for a **class** that defines a set of methods a class must implement. **Interfaces** provide **100% abstraction** (only abstract methods traditionally). Since Java 8, they can also contain **default** and **static** methods and since Java 9, **private** methods.
A class implements an **interface** using the **implements** keyword.

```
class Car implements Vehicle {
 // Implement methods
}
```

A class can implement **multiple interfaces**, allowing **multiple inheritance**

```
class Student implements Sports, Cultural {
// Implement methods
}
```

All methods are **public and abstract**by default (unless they are private, default, or static methods introduced in newer Java versions).
All interface variables are constants and are implicitlydefault(by :

- public
- static
- final

```
interface Test {
    int MAX = 100; // public static final
  }
```

A class implementing an interface must implement all abstract methods, or it must be declared abstract.

Interfaces support polymorphism.

```
Animal a = new Dog(); // Upcating
a.sound();
```

One interface can extend another interface.

```
interface A {
```

```
        void display();
    }
    interface B extends A
        { void show();
    }
```

Interfaces cannot be instantiated.

InterfaceNameobj = new InterfaceName(); // Compiler Error

Interfaces are commonly used in Java frameworks (such as collections, JDBC, and Spring) to define contracts between components.

Syntax:

```
interface interface_name
{
final varibales
abstract methods
}
```

Example 1:

```
interface I1
{
public static final int a=10, b=20;
public abstract void m1();
}
```

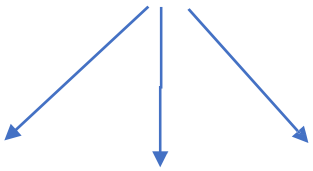
Example 2:

```
public interface I2
{
    public static final int a=10, b=20;
    public abstract void m1();
}
```

- By default,all the **variables** of the interface are **public,static and final**
- By default, all the **methods**of the interface are **public and abstract**
- By default, the **access modifier** of the interface is **<default>**
- We can use **public**access modifier for the interface explicitely

After defining the interface, there must be atleast one class which implements the interface and that class has to provide implementation to all the abstract methods of that interface. The class which implements the interface is called implementing class

Example:I1
interface I1{ }
class A implements I1 { }
class B implements I1 { }
class C implements I1 { }
A B C



- **For interfaces,** we can not create objects with new operator like abstract classes but we can only create reference variables.
- **Upcasting** is usedto access the members of the interface
- **Upcasting w.r.t interfaces:**implementing class object is assigned to interface reference variable,with thisreference variablewe can access all the members of interface except overridden methods. In case of overridden methods of the interface, implementing class overriding methods are executed.

Feature	Concrete Class	Abstract Class	Interface
Variables	Consists of both final and non-final variables	Consists of both final and non-final variables	Consists of only final variables
Methods	Consists of only concrete methods	Consists of both concrete and abstract methods	Consists of only abstract methods
Constructors	Can have constructors	Can have constructors	Can't have constructors
Objects	Objects can be created with **new** operator	Objects can't be created with **new** operator	Objects can't be created with **new** operator

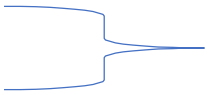
Differences betweenConcrete Class, Abstract Class and Interface

Programs on interfaces:

- **one class can implement one interface**
  P1: interface I1, class A implements I1
- **any number of classes can implement one interface**
  P2: interface I1, class A implements I1, class B implements I1
- **one class can implement any number of interfaces**
  P3: interface I1, interface I2, class A implements I1, I2(Multiple inheritance)
- **one class can extend one class and can implement any number of interfaces.**
  P4: interface I1, interface I2, class A, class B extends A implements I1,I2
- **interfaces can also be extended like classes.**
  P5: interface I1, interface I2 extends I1, class A implements I2
- **one interface can extend any number ofinterfaces (Multiple inheritance)**
  P6:interface I1,interface I2,interface I3 extends I1,I2,class A implements I3
- **abstract class can implement the interface**
  P7: interface I1,abstract class A implements I1, class B extends A

Between classes

Between interfaces



extends

Between class and interface→

implements

```
// P1: Program using interface I1, CLASS A IMPLEMENTS I1
interface I1 {
    public static final int a = 10;
    public abstract void m1();
    public abstract void m2();
}
class A implements I1
    { int b = 20;
    public void m1()
        { System.out.println("A's
        m1()");
    }
    public void m2()
        { System.out.println("A's
        m2()");
    }
    void m3() {
        System.out.println("A's m3()");
    }
}
class P1interfacedemo {
    public static void main(String[] args) {
        // UPCASTING
        I1 i1 = (I1) new A();
        // with UPCASTING only a, m1, m2 can be accessed
        i1.m1();
        i1.m2();
System.out.println(I1.a + " " + i1.a);
        A a1 = new A();
        a1.m3();
System.out.println(a1.b);
```

```
c:\nrcm>javac P1interfacedemo.java

c:\nrcm>java P1interfacedemo
A's m1()
A's m2()
10 10
A's m3()
20
```

```
// P2: Program using interface I1, CLASS A IMPLEMENTS I1, CLASS B IMPLEMENTS I1
interface I1 {
    public static final int a = 10;
    public abstract void m1();
    public abstract void m2();
}
class A implements I1
    { int b = 20;
    public void m1()
{ System.out.println("A's
m1()");
    }
    public void m2()
{ System.out.println("A's
m2()");
    }
    void m3()
{ System.out.println("A's
m3()");
    }
}
class B implements I1
    { int c = 30;
   public void m1()
{ System.out.println("B's
m1()");
    }
    public void m2()
{ System.out.println("B's
m2()");
    }
    void m4()
```

```
        I1 i1 = (I1) new A();    // UPCASTING
        i1.m1();// with UPCASTING only a, m1, m2 can be accessed
        i1.m2();
System.out.println(I1.a + " " + i1.a);
        A a1 = new A();
        a1.m3();
System.out.println(a1.b);
        B b1 = new B();
        b1.m4();
System.out.println(b1.c);
    }
}
```

```
c:\nrcm>javac P2interfacedemo.java

c:\nrcm>java P2interfacedemo
A's m1()
A's m2()
10 10
A's m3()
20
B's m4()
30
```

```
// P3: Program using interface I1, interface I2, Class A implements I1, I2 (Multiple Inheritance)
interface I1 {
    public static final int a = 10;
    public abstract void m1();
    public abstract void m2();
}
interface I2 {
    public static final int b = 20;
    public abstract void m3();
    public abstract void m4();
}
class A implements I1, I2 {
    int c = 50;    // A's own variable
    public void m1() {
System.out.println("A's m1()");
    }
    public void m2()
{ System.out.println("A's
m2()");
```

```
    public void m3()
{ System.out.println("A's
m3()");
    }
    public void m4()
{ System.out.println("A's
m4()");
    }
    void m5() {    // A's own method
System.out.println("A's m5()");
    }
}
class P3interfacedemo {
        a1.m1();       // Accessing interface methods
        a1.m2();
        a1.m3();
        a1.m4();
        a1.m5();       // Accessing class-specific members
System.out.println("A's variable c: " + a1.c);
System.out.println("I1.a = " + I1.a); // Accessing constants from interfaces
System.out.println("I2.b = " + I2.b);
    }
}
```

```
c:\nrcm>javac P3interfacedemo.java

c:\nrcm>java P3interfacedemo
A's m1()
A's m2()
A's m3()
A's m4()
A's m5()
A's variable c: 50
I1.a = 10
I2.b = 20
```

```
// P4: Program where one class extends another class and implements multiple interfaces
// Structure: interface I1, interface I2, class A, class B extends A implements I1, I2

interface I1 {
    public static final int a = 10;
    public abstract void m1();
```

```
    public abstract void m2();
}
interface I2 {
    public static final int b = 20;
    public abstract void m3();
    public abstract void m4();
}
// Parent class
class A {
    int d = 100;
    void parentMethod()
{ System.out.println("A's
parentMethod()");
    }
}
// Child class extending class A and implementing both interfaces I1 and I2
class B extends A implements I1, I2 {
    int c = 50; // B's own variable
    // Implementing methods from interface I1
    public void m1() {
System.out.println("B's m1()");
    }
    public void m2()
{ System.out.println("B's
m2()");
    }
    // Implementing methods from interface I2
    public void m3() {
System.out.println("B's m3()");
    }
    public void m4()
{ System.out.println("B's
m4()");
    }
    // B's own specific method
    void m5() {
System.out.println("B's m5()");
    }
}
class P4interfacedemo {
    public static void main(String[] args) {
        // 1. UPCASTING to Interface I1
        I1 i1 = (I1) new B();
        i1.m1();
```

```
        i1.m2();
System.out.println(I1.a + " " + i1.a);
```

```
        // 2. UPCASTING to Interface I2
        I2 i2 = (I2) new B();
        i2.m3();
        i2.m4();
System.out.println(I2.b + " " + i2.b);
        // 3. UPCASTING to Parent Class A
        A a1 = (A) new B();
        a1.parentMethod();
System.out.println("Parent class variable: " + a1.d);
        // 4. Accessing everything directly via Child Class B reference
        B b1 = new B();
        b1.m5();
System.out.println("B's own variable c: " + b1.c);
        // B can also access inherited components from A directly
        b1.parentMethod();
System.out.println("Inherited parentVar in B: " + b1.d);
    }
}
```

```
c:\nrcm>javac P4interfacedemo.java

c:\nrcm>java P4interfacedemo
B's m1()
B's m2()
10 10
B's m3()
B's m4()
20 20
A's parentMethod()
Parent class variable: 100
B's m5()
B's own variable c: 50
A's parentMethod()
Inherited parentVar in B: 100
```

```
// P5: Program where an interface extends another interface
// Structure: interface I1, interface I2 extends I1, class A implements I2
interface I1 {
    public static final int a = 10;
    public abstract void m1();
    public abstract void m2();
}
// Interface extending another interface
```

```
interface I2 extends I1 {
    public static final int b = 20;
    public abstract void m3();
    public abstract void m4();
}
// Class A implements the sub-interface I2.
// It must implement methods from both I1 and I2.
class A implements I2 {
    int c = 30; // A's own variable
    // Implementing methods from parent interface I1
    public void m1() {
System.out.println("A's m1()");
    }
    public void m2()
{ System.out.println("A's
m2()");
    }
    // Implementing methods from sub-interface I2
    public void m3() {
System.out.println("A's m3()");
    }
    public void m4()
{ System.out.println("A's
m4()");
    }
    // A's own specific method
    void m5() {
System.out.println("A's m5()");
    }
}

class P5interfacedemo {
    public static void main(String[] args) {
            // 1. UPCASTING to Parent Interface I1
        I1 i1 = (I1) new A();
        i1.m1();
        i1.m2();
System.out.println(I1.a + " " + i1.a);
        // 2. UPCASTING to Sub-Interface I2
        I2 i2 = (I2) new A();
        i2.m1();
        i2.m2();
        i2.m3();
        i2.m4();
```

```
System.out.println(I2.b + " " + i2.b + " " + i2.a);
```

```
        // 3. Direct access via Class A reference
        A a1 = new A();
        a1.m5();
System.out.println("A's own variable c: " + a1.c);
    }
}
```

```
c:\nrcm>javac P5interfacedemo.java

c:\nrcm>java P5interfacedemo
A's m1()
A's m2()
10 10
A's m1()
A's m2()
A's m3()
A's m4()
20 20 10
A's m5()
A's own variable c: 30
```

```
// P6: Program where one interface extends multiple interfaces (Multiple Inheritance)
// Structure: interface I1, interface I2, interface I3 extends I1, I2, class A implements I3

interface I1 {
    public static final int a = 10;
    public abstract void m1();
    public abstract void m2();
}
interface I2 {
    public static final int b = 20;
    public abstract void m3();
    public abstract void m4();
}
// Interface extending multiple interfaces using commas
interface I3 extends I1, I2 {
    public static final int c = 30;
    public abstract void m5();
}
// Class A implements the sub-interface I3.
// It must implement all methods from I1, I2, and I3.
```

```
class A implements I3 {
    int d = 50; // A's own variable
    // Implementing methods from interface I1
    public void m1() {
System.out.println("A's m1()");
    }
    public void m2()
{ System.out.println("A's
m2()");
    }
    // Implementing methods from interface I2
    public void m3() {
System.out.println("A's m3()");
    }
    public void m4()
{ System.out.println("A's
m4()");
    }
    // Implementing methods from sub-interface I3
    public void m5() {
System.out.println("A's m5()");
    }
    // A's own specific method
    void m6() {
System.out.println("A's m6()");
    }
}

class P6interfacedemo {
    public static void main(String[] args) {
        // 1. UPCASTING to Interface I1
        I1 i1 = (I1) new A();
        i1.m1();
        i1.m2();
System.out.println(I1.a + " " + i1.a);
        // 2. UPCASTING to Interface I2
        I2 i2 = (I2) new A();
        i2.m3();
        i2.m4();
System.out.println(I2.b + " " + i2.b);
        // 3. UPCASTING to Sub-Interface I3
        I3 i3 = (I3) new A();
          i3.m1();
        i3.m3();
```

```
i3.m5();
```

```
System.out.println(I3.a + ", " + I3.b + ", " + I3.c);
        // 4. Accessing everything directly via Class A reference
        A a1 = new A();
        a1.m6();
System.out.println("A's own variable d: " + a1.d);
    }
}
```

```
c:\nrcm>javac P6interfacedemo.java

c:\nrcm>java P6interfacedemo
A's m1()
A's m2()
10 10
A's m3()
A's m4()
20 20
A's m1()
A's m3()
A's m5()
10, 20, 30
A's m6()
A's own variable d: 50
```

```
// P7: Program demonstrating an interface implemented by an abstract class
// Structure: interface I1, abstract class A implements I1, class B extends A
interface I1 {
    public static final int a = 10;
    public abstract void m1();
    public abstract void m2();
}
// Abstract class implementing interface I1
// It does not need to implement m1() and m2() immediately.
abstract class A implements I1 {
    int b = 100; // A's own variable
    void parentMethod() {
System.out.println("Abstract Class A's parentMethod()");
    }
}
// Concrete class extending abstract class A
// Class B must implement the inherited abstract methods m1() and m2() from I1
class B extends A {
    int c = 50; // B's own variable
```

```
    // Implementing methods originating from interface I1
    public void m1() {
System.out.println("B's m1() [Implementation of I1]");
    }
    public void m2() {
System.out.println("B's m2() [Implementation of I1]");
    }
    // B's own specific method
    void m5() {
System.out.println("B's m5()");
    }
}
class P7interfacedemo {
    public static void main(String[] args) {
        // 1. UPCASTING to Interface I1
        I1 i1 = (I1) new B();
        i1.m1();
        i1.m2();
System.out.println(I1.a + " " + i1.a);
        // 2. UPCASTING to Abstract Class A
        A a1 = (A) new B();
        a1.m1();
        a1.m2();
        a1.parentMethod();
System.out.println("Variable from Abstract class A: " + a1.b);
        // 3. Accessing everything directly via Concrete Child Class B reference
        B b1 = new B();
        b1.m5();
System.out.println("B's own variable c: " + b1.c);
        // B inherits everything from abstract parent class A
        b1.parentMethod();
System.out.println("Inherited parentVar in B: " + b1.a);
    }
}
```

```
c:\nrcm>javac P7interfacedemo.java

c:\nrcm>java P7interfacedemo
B's m1() [Implementation of I1]
B's m2() [Implementation of I1]
10 10
B's m1() [Implementation of I1]
B's m2() [Implementation of I1]
Abstract Class A's parentMethod()
Variable from Abstract class A: 100
B's m5()
B's own variable c: 50
Abstract Class A's parentMethod()
Inherited parentVar in B: 10
```

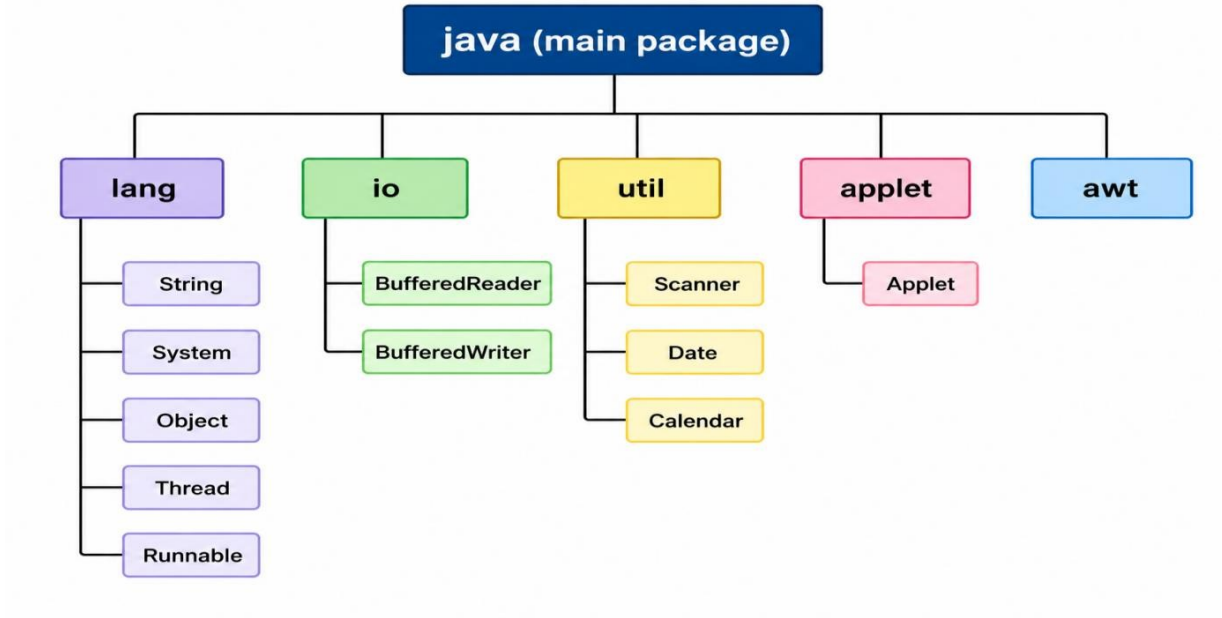
Package

A package is a collection of classes and interfaces.

Packages are of 2 types

1. Pre-defined Packages
2. User-Defined Packages

Pre-defined Packages: Java provides a rich set of built-in packages structured under the core java main package namespace



User-Defined Packages

Developers can create their own custom packages to organize application code components.

Syntax:

package packagename; // Must be the first statement in the file

Example:

package pack1;

```
class A {
  // ...
}

class B {
    // ...
}
interface i1 {
    // ...
}
interface i2 {
    // ...
}
```

pack 1 is a collection of 2 classes & 2 interfaces.

Structure for Java Program & Writing Rules

Structure for Java Program:

1. There should be at most 1 package statement in the program.
2. The package statement must be the first non-comment statement.
3. After package statement, we can write any number of import statements.
4. After import statements, we can write any number of classes and interfaces.

Example Frameworks:

```
// Struct 1
package pack1;
import java.lang.*;
import java.util.*;
class A { }
class B { }
interface i1 { }

// Struct 2
import java.lang.*;
import java.util.*;
class A { }
class B { }
interface i1 { }
```

Rule for writing packages:

If multiple classes are there in a package/file, then, there should be only **one public class** and the name of the file must be the same as the name of the public class.

Example:

```
package pack1;
public class A {
    // ...
```

```
}
class B {
   // ...
}
class C {
   // ...
}
```
Save this file as A.java since, A is a public class.

Package

```
package pack1;
public class A {
        public static void main(String[] args)
                { System.out.println("Hello
                package");
         }
}
```

- **Save**: A.java (name of the public class)
- **Compile**: javac -d . A.java
  - -d directs all class files into package.
- **Execution**: java pack1.A [Package.classname]
- **O/P**: Hello package

Access Modifiers: Java provides 4 explicit levels of visibility constraints to manage object-oriented access boundary rules

- Classes : Top-level classes can only be declared with public or <default> visibility access. They cannot be explicitly marked as private or protected.

- Members : Structural elements like variables and methods can utilize all 4 visibility constraints.

Scope Hierarchy: private ,< default >, protected , public

Example:

```
class A {
  int a, b; // Default access modifier (Package-private scope)
     void m1() {
            // Method accessible only inside this package
}
  int m2() {
return 0;
  }
}

public class B {
int a, b; // Accessible within package boundary
 public void m1() {
  // Globally accessible method scope
```

```
}
```

}

Package Implementation Scenarios

Programs on Packages:

1. Single package
2. Import one package into another package (2 packages)
    - a. import all the classes of a package.
    - b. import only some classes.
    - c. Use fully qualified name.
3. Sub packages (2 packages min)

Prog 1: Single package

```
package pack;
public class A {
 public static void main(String[] args)
{ System.out.println("Welcome to
package");
 }
}
```

Without packages:

A.java

|

A.class pack1

With packages:

A.java

|

|

A.class

Prog 2: package with 2 classes

```
// P8: Program demonstrating packages in Java
// Structure: package pack1, class A (public), class B (default)
package pack1;
public class A {
       public static void main(String[] args)
{ System.out.println("Welcome to package");
       }
}
class B {
      void m1()
{ System.out.println("B's
m1()");
     }
}
```

A.java -> pack1 ->A.class, B.class (new folder)

```
c:\nrcm>javac -d . A.java

c:\nrcm>java pack1.A
Welcome to package
```

Multiple Public Classes in the Same Package

Prog 3:

We can put any number of public classes in the same package. For that, we need to write separate prog for each public class.

Put 2 public classes A & B in the same package pack1. Then write 2 separate progs.

P1:

```
package pack1;
public class A {
  public static void main(String[] args)
        { System.out.println("Welcome to public class A");
 }
}
```

P2:

```
package pack1;
public class B {
        public static void main(String[] args)
                { System.out.println("Public class B");
  }
}
```

Directory:

```
 pack1
   |-- A.class
   +— B.class
```

Inter-Package Import Implementation

Prog:

// importing packages.We can import public classes only of one package into another package.

```
package pack1;
public class A {
            public void m1()
                { System.out.println("A-
                m1()");
        }
}
```

```
package pack2;
import pack1.A;
```

```
public class B {
        public static void main(String[] args)
                { A a1 = new A();
                a1.m1();
        }
}
```

```
c:\nrcm>javac -d . A.java

c:\nrcm>javac -d . B.java

c:\nrcm>java pack2.B
A-m1()
```

pack2
└─ B.class

➔ Without using import statement, we can implement by using fully qualified name.
pack1.A a = new pack1.A();

Page 8: Sub-Package Configurations
How do we create sub-packages
Prog on sub packages

```
// P8: Program demonstrating packages and sub-packages in Java
// Main package
package  pack1;
public class A {
   public static void main(String[] args)
{ System.out.println("main package: pack1");
}
}
```

```
// Sub-package
package pack1.pack2;
public class B {
 public static void main(String[] args)
{ System.out.println("Sub Package: Pack
2");
}
```

OUTPUT:
Save: B.java
Compile: javac -d . B.java
Execute: java pack1.pack2.B

```
c:\nrcm>javac -d . A.java

c:\nrcm>javac -d . B.java

c:\nrcm>java pack1.pack2.B
Sub Package: Pack 2
```

pack1
└─ pack2
└─ B.class

Access Modifiers for Variables & Methods

Explain access modifiers in Java variables & methods.

	Private	<default>	Protected	Public
1. Within the class	Yes	Yes	Yes	Yes
2. Subclasses of same package	No	Yes	Yes	Yes
3. Non-sub classes of same package	No	Yes	Yes	Yes
4. Subclasses of the other package	No	No	Yes	Yes
5. Non-sub classes of other package	No	No	No	Yes

Access Modifiers - Variables, Methods

- **Private**: Private members are accessible only within the same class. Outside the class they are not accessible or not visible.
- **Default**: Default members can be accessible only within the same package. i.e., both sub-classes & non-sub classes of the same package can use default members.
- **Protected**: Protected members can be accessed by both sub-classes & non-sub classes of the same package and also sub-classes of the other package.
- **Public**: (All can be accessed) Public members can be accessed by sub-classes & non-sub classes of same package and also sub-classes & non-sub classes of other package.

Private Access Modifier Scope

Access modifiers - classes:

For classes we can use only two access modifiers — default & public.

- **Default**: Default classes can be accessed only within the same package. Outside the package, default classes cannot be accessed.

- **Public**: Public classes can be accessed within the same package & outside the package also.

Program on access modifiers — variables, methods

Private members of a class can be accessed only within the same class. Outside the class they are not visible.

```
// P9: Program demonstrating private access modifier in Java
// Private members are accessible only within the same class.

class Student
   { private int sno =
   1;
private int marks = 59; // private instance variables
    private void display() { // private method
System.out.println("sno is " + sno);
System.out.println("marks are " + marks);
}
}
class Test {
public static void main(String[] args)
{ Student s1 = new Student();
       // Trying to access private members outside the class
System.out.println(s1.sno + " " + s1.marks); // Compiler error
s1.display(); // Compiler error
}
```

```
c:\nrcm>javac Test.java
Test.java:16: error: sno has private access in Student
                System.out.println(s1.sno + " " + s1.mark
s); // Compiler error
                                   ^
Test.java:16: error: marks has private access in Student
                System.out.println(s1.sno + " " + s1.mark
s); // Compiler error
                                                       ^
Test.java:17: error: display() has private access in Stude
nt
                s1.display(); // Compiler error
                  ^
3 errors
```

To use this we have to not declare the method as private (display method) in student class.

O/P:

sno is 1

marks are 59

Default Access Modifier

Default members of a class can be accessed by sub-classes and non-sub classes of same package. To explain default access modifier, we need to take one package.

```
// P10: Program demonstrating default access modifier in Java
// Default members are accessible only within the same package.

package pack1;
class A {
   int a = 10, b = 20; // default access variables

   void display() { // default method
System.out.println(a + " " + b);
   }
}
class B extends A { // subclass of A
   public static void main(String[] args) {
      A a1 = new A();
System.out.println(a1.a + " " + a1.b);
   }
}
```

```
c:\nrcm>javac -d . B.java

c:\nrcm>java pack1.B
10 20
```

Package Compilation & Protected Setup

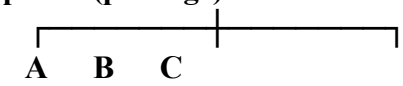
```
// P11: Program demonstrating default access modifier in Java
// Non-subclass accessing default members within the same package.

package pack1;

class C { // non-subclass of A
   public static void main(String[] args)
      { A a2 = new A();
      a2.display();
System.out.println(a2.a + " " + a2.b);
   }
```

```
}
```

- **Save: A.java (or) B.java (or) C.java**
- **Compiler: javac -d . A.java**
- **Execute: java pack1.B OR java pack1.C**

```
c:\nrcm>javac -d . A.java

c:\nrcm>javac -d . B.java

c:\nrcm>javac -d . C.java

c:\nrcm>java pack1.C
10 20
10 20

c:\nrcm>java pack1.B
10 20
```

pack1 (package)



A B C

(all default classes, one public)

Protected Access Modifier

Protected members of a class can be accessed by sub-classes and non-sub classes of the same package and also by the sub-classes of other package. To explain protected access modifier, we need to take 2 packages.

```
// P12: Program demonstrating protected access modifier in Java
// Protected members are accessible within the same package.

package pack1;

public class A { // public so it can be imported in other packages
 protected int a = 10, b = 20;
    protected void display()
{ System.out.println(a + " " + b);
  }
}
```

```
package pack1;

class B extends A {
public static void main(String[] args) {
A a1 = new A();
System.out.println(a1.a + " " + a1.b);
a1.display();
}
}
```

```
package pack1;

class C {
  public static void main(String[] args)
   { A a2 = new A();
a2.display();
System.out.println(a2.a + " " + a2.b);
}
}
```

```
// P14: Program demonstrating protected access modifier across packages
package pack2;
import pack1.A; // class A is available here

class D extends A { // subclass of A in another package
    public static void main(String[] args) {
        D a3 = new D();
System.out.println(a3.a + " " + a3.b);
        a3.display();
```

```
  }
}
```

```
c:\nrcm>javac -d . A.java

c:\nrcm>javac -d . B.java

c:\nrcm>javac -d . C.java

c:\nrcm>javac -d . D.java

c:\nrcm>java pack2.D
10 20
10 20
```

Public Members Implementation

Public members of a class can be used by sub-classes and non-sub classes of the same package and also by subclasses and non-subclasses by another package.

```
// P15: Program demonstrating public access modifier in Java
// Public members are accessible everywhere, across packages and classes.
package pack1;
public class A {
   public int a = 10, b = 20;
    public void display() {
System.out.println(a + " " + b);
}
}
class B extends A {
 public static void main(String[] args)
{ A a1 = new A();
System.out.println(a1.a + " " + a1.b);
a1.display();
}
}
class C {
  public static void main(String[] args)
  { A a2 = new A();
a2.display();
System.out.println(a2.a + " " + a2.b);
}
}
```

```
c:\nrcm>javac -d . A.java

c:\nrcm>java pack1.B
10 20
10 20

c:\nrcm>java pack1.C
10 20
10 20
```

Cross Package Public & Class Modifiers

```
// P16: Program demonstrating public access modifier across packages
// Public members are accessible everywhere, including non-subclasses in other packages.
package pack2;
import pack1.A;
class D extends A { // subclass of A in another package
  public static void main(String[] args) {
   A a3 = new A();
System.out.println(a3.a + " " + a3.b);
a3.display();
}
}
class E { // non-subclass in another package
public static void main(String[] args) {
A a4 = new A();
System.out.println(a4.a + " " + a4.b);
a4.display();
}
}
```

```
c:\nrcm>javac -d . A.java

c:\nrcm>javac -d . D.java

c:\nrcm>java pack2.D
10 20
10 20
```

Access modifiers for classes:

For classes access modifiers are default and public.

Difference b/w default & public:

- Default classes can be used only within the same package, outside the package they cannot be used.

- Public classes can be used within the same package & outside also.
- Default classes of one package cannot be imported into another package, whereas public classes can be imported.

Importing Visibility Boundaries

```
package pack1;
class A { // default
   // ...
}
public class B { // public
   // ...
}
class C extends A { // default
   // ...
}
package pack2;
import pack1.A; // Compiler error
import pack1.B;
class C {
   // ...
}
```

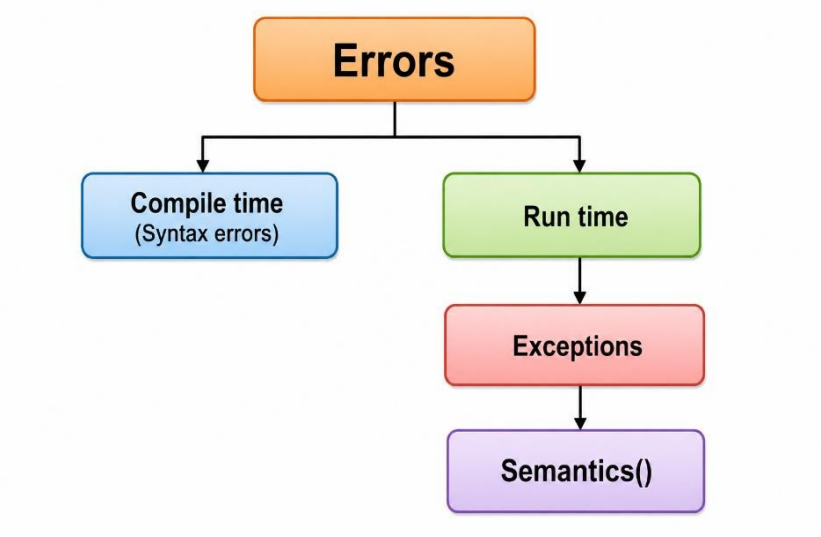
For multiple public classes:

```
import pack1.*;
```

OR

```
import pack1.A.B;
```

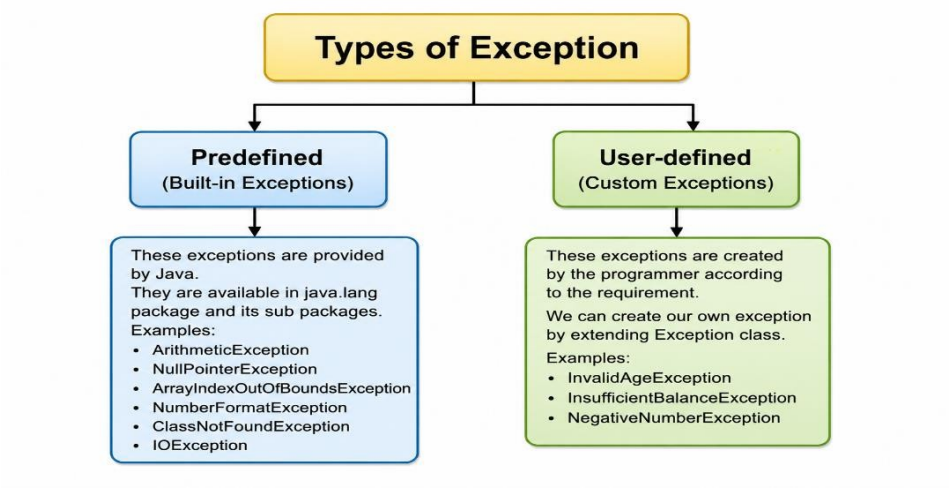
Exception Handling:



What is an exception?

An unwanted or unexpected event that disturbs / disrupts the normal flow of the prog is called an exception.

Types of Exception:



In Java, all exceptions are classes.

Predefined exceptions examples:

- Array index out of bound exception.
- Arithmetic exception.
- Class cast exception.
- Null pointer exception.

Purpose and Core Keywords

Purpose of exception handling:

It is always a good programming practice to handle the exceptions. The main objective of exception handling is graceful termination of the program or to avoid the abnormal termination.

Meaning of exception handling:

Exception handling does not mean repairing the exception. We have to provide an alternative way to continue the rest of the program normally.

For exception handling, we use the following 5 Keywords:

1. try
2. catch
3. finally
4. throw
5. throws

In Java, try, catch & finally are blocks.

In try block, we write risky code

```
try {
   // risky code
}
```

Inside the catch codeblocks we write handling code.

```
catch (Classnamerefvariable) { // we send exception class object as parameters to catch block.
   // handling code
```

}

Clean-up Block & Program Demonstration

Inside the finally block, we write clean-up code.

```
finally {
   // clean-up code
}
```

➔ We will handle the exceptions in java by using try, catch blocks.

Prog: Without exception handling

(abnormal termination occurs)

```
// P17: Program demonstrating runtime exception in Java
// Division by zero causes ArithmeticException at runtime.

class ExceptionDemo1 {
  public static void main(String[] args)
{ System.out.println("Stmt1");
System.out.println("Stmt 2");
System.out.println(10 / 0); // runtime error - exception
System.out.println("Stmt 3");
System.out.println("Stmt 4");
   }
}
```

```
c:\nrcm>javac ExceptionDemo1.java

c:\nrcm>java ExceptionDemo1
Stmt1
Stmt 2
Exception in thread "main" java.lang.ArithmeticException:
/ by zero
        at ExceptionDemo1.main(ExceptionDemo1.java:8)
```

(rest of the prog will not be executed due to Arithmetic exception: / by zero)

➔ The exception is handled by JVM by default exception handling.

Prog 2: With exception handling - using try, catch blocks

↳ Graceful termination. To avoid abnormal termination.

```
// P18: Program demonstrating exception handling in Java using try-catch
// Division by zero is caught and handled gracefully.

class ExceptionDemo2 {
```

```
  public static void main(String[] args)
{ System.out.println("Stmt1");
System.out.println("Stmt2");
      try
{ System.out.println("Stmt3");
System.out.println(10 / 0); // ArithmeticException
System.out.println("Stmt4"); // not executed
 }
catch (ArithmeticException ae)
{ System.out.println("Exception is
handled");
}
System.out.println(10 / 2);
System.out.println("Stmt5");
  }
```

```
c:\nrcm>javac ExceptionDemo2.java

c:\nrcm>java ExceptionDemo2
Stmt1
Stmt2
Stmt3
Exception is handled
5
Stmt5
```

➔ Here, the exception is handled by JVM by **Customized Exception handling**.

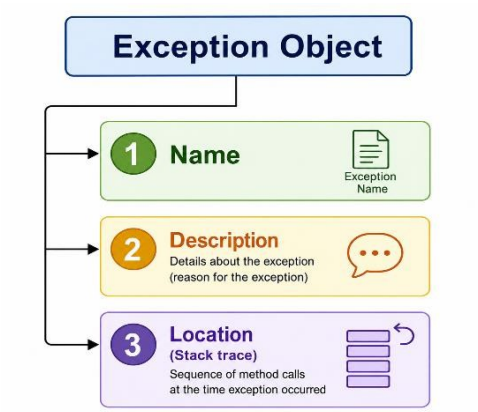
Rules of Exception Handling Mechanism

- Inside the try block, always put only the risky code, i.e., the length of the try block should be as min as possible.
- The code which is there after the exception statement inside the try block will not be executed.

Default Exception handling mechanism in Java (JVM will use)

If we won't handle the exception then, JVM will use the following procedure to give the output.

Step 1: If the exception is raised in any method then, that method is responsible for creating the exception object by including the following information.



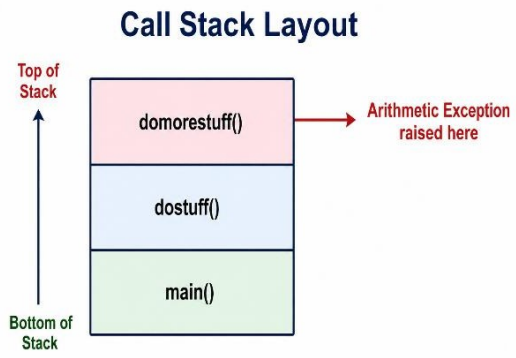
Step 2: After creation of the exception, the method hand overs that obj to JVM.
Step 3: JVM will check for exception handling code in that method. If the method does not contain exception handling code, then, JVM will terminate that method abnormally and removes corresponding entry from the stack.
Step 4: JVM identifies calling or call method and checks for exception handling code in the calling method. If the calling method also does not contain exception handling code then, JVM terminates the calling method also abnormally and removes the corresponding entry also from the stack.
Step 5: JVM continues the same procedure till the main method is reached. Once the main method is reached, JVM verifies exception handling code in the main method. If the main method also does not contain exception handling code, then, JVM terminates the main method abnormally and removes the corresponding entry from the stack. Just before terminating the main method from the stack, JVM handover the exception object to default exception handler.
Step 6: Default exception handler is a part of JVM which prints the object information on the console as output.

Prog: On Default exception handling mechanism.

```
// P19: Program demonstrating exception propagation in Java
// Exception created in domorestuff() propagates up the call stack.

class ExceptionDemo {
 public static void main(String[] args)
{ dostuff();
}
   static void dostuff()
{ domorestuff();
}
   static void domorestuff()
{ System.out.println("Hello");
System.out.println(10 / 0); // ArithmeticException created here
}
```

```
}
```

```
c:\nrcm>javac ExceptionDemo.java

c:\nrcm>java ExceptionDemo
Hello
Exception in thread "main" java.lang.ArithmeticException:
/ by zero
        at ExceptionDemo.domorestuff(ExceptionDemo.java:13
)
        at ExceptionDemo.dostuff(ExceptionDemo.java:9)
        at ExceptionDemo.main(ExceptionDemo.java:6)
```



Direct Proximity Constraints & Alternate Stack Traces

Note: In between try and catch blocks, don't write any statements. i.e., catch block must be immediately followed by try block.

```
// Valid Structure
try {
   // ...
}
catch (Exception e) {
   // ...
}
// Invalid Structure (Compiler Error)
try {
   // ...
}
System.out.println(" "); // Compiler error occurs here
catch (Exception e) {
   // ...
}
```

Prog: Alternative Scenario

Note: In between try and catch blocks, don't write any statements. i.e., catch block must be immediately followed by try block.

```
try {
      // ...
   }
catch() {
       // ...
   }
// Compiler Error Scenario
   try {
      // ...
   }
   S.O.P(" "); ──► Compiler error occurs here
catch() {
      // ...
   }
```

Exception Instantiation & Metadata Capture

Note: Inside the try block always put the only risky code i.e., length of the try block should be as minimum as possible.

The code which is there after exception statement in try block will not be executed.

Default Exception handling mechanism in Java (JVM uses this procedure)

If we won't handle Exception, then JVM will use the following procedure to give the output.

Step 1: If the Exception is raised in any method, then that method is responsible for creating the exception object by including the following information:

1. Name of Exception
2. Description of Exception
3. Location of Exception (Stack trace)

Exception Propagation & Stack Frames Unwinding

Step 2: After creation of the exception object, the method handovers that object to JVM.

Step 3: JVM will check for Exception handling code in the method. If that method does not contain exception handling code, then JVM will terminate that method abnormally and removes the corresponding entry from stack.

Step 4: JVM identifies the calling method and checks for Exception handling code in the calling method / caller method. If the calling method also does not contain exception handling code then JVM terminates the calling method also abnormally and removes the corresponding entry from stack.

Step 5: JVM continues the same procedure till the main method is reached. Once the main method is reached, JVM verifies exception handling code in main method. If the main method also does not contain Exception handling code, then JVM terminates the main method abnormally and removes the corresponding entry from stack.

Just before terminating the main method from the stack, JVM handovers that Exception object to default exception handler.

Step 6: Default Exception handler is a part of JVM which prints the object information on console as output.

Predefined Exceptions

1. ArithmeticException
2. NullPointerException
3. ArrayIndexOutOfBoundsException
4. ClassCastException
5. StringIndexOutOfBoundsException

Comparison of Mechanisms

Default Exception Handling:

- The JVM handles the exception.
- Results in **abnormal termination**.
- Occurs when the programmer does not write try and catch blocks (does not handle the exception).

Customized Exception Handling:

- The programmer creates try and catch blocks.
- The programmer handles the exception explicitly.
- Results in **graceful termination**.

Customized Exception Handling — ArithmeticException

Program 1: ArithmeticException Handling

Syntax Structure Layout:

```
class Test {
   public static void main(String[] args)
{ System.out.println("stmt1");
      try {
         // Risky code here
      }
      catch (Exception e) {
         // Handling code here
      }
System.out.println("stmt4");
   }
}
```

Implementation Code:

```
// P20: Program demonstrating exception handling in Java
// ArithmeticException is caught and handled inside catch block.

class Test {
 public static void main(String[] args)
{ System.out.println("Stmt1");
```

```
      try
{ System.out.println("Stmt2");
System.out.println(10 / 0); // Exception occurs here
System.out.println("Stmt3"); // not executed
 }
 catch (ArithmeticException ae)
{ System.out.println("Arithmetic Exception is handled");
System.out.println(10 / 2);
}
   }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
Stmt1
Stmt2
Arithmetic Exception is handled
5
```

Program 2: NullPointerException (NPE)

Scenario 1: Reference pointing to null

Student s1; ⟶ s1 is null

Runtime Error: NullPointerException (NPE)

Scenario 2: Empty String method call

```
String s = null;
System.out.println(s);          // Prints: null
System.out.println(s.length());  // Runtime Error: NullPointerException
```

NullPointerException Code

```
// P21: Program demonstrating NullPointerException handling in Java
// Attempting to call a method on a null reference triggers NPE.

class Test {
public static void main(String[] args)
 { try {
String s1 = null;
System.out.println(s1.length()); // Exception raised here
}
 catch (NullPointerExceptionnpe)
{ System.out.println("NPE is handled");
}
```

```
}
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
NPE is handled
```

Array Overruns (ArrayIndexOutOfBoundsException)

Program : ArrayIndexOutOfBoundsException

```
// P22: Program demonstrating ArrayIndexOutOfBoundsException handling in Java
// Attempting to access invalid array index triggers exception.

class Test {
public static void main(String[] args)
{ try {
int[] a = new int[4]; // valid indices: 0, 1, 2, 3
System.out.println(a[4]); // bounds violation (or a[-1])
  }
 catch (ArrayIndexOutOfBoundsExceptionaioobe)
{ System.out.println("AIDOBE is handled");
 }
}
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
AIDOBE is handled
```

String Traversals (StringIndexOutOfBoundsException)

Program 5: StringIndexOutOfBoundsException

```
// P23: Program demonstrating StringIndexOutOfBoundsException handling in Java
// Attempting to access invalid string index triggers exception.

class Test {
public static void main(String[] args)
{ try {
 // String indices: 0('R'), 1('a'), 2('m'), 3('u')
String s1 = new String("Ramu");
```

```
System.out.println(s1.charAt(4)); // index 4 is out of bounds
  }
 catch (StringIndexOutOfBoundsExceptionsioobe)
{ System.out.println("Exception is handled");
}
System.out.println("Stmt");
 }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
Exception is handled
Stmt
```

Type Casting Diagnostics (ClassCastException)
Program 6: ClassCastException

```
// P24: Program demonstrating ClassCastException handling in Java
// Invalid downcasting triggers ClassCastException at runtime.

class Test {
public static void main(String[] args)
{ System.out.println("Stmt1");

try {
Object o1 = new Object();
String s1 = (String) o1; // invalid downcasting
}
catch (ClassCastExceptioncce)
{ System.out.println("CCE is handled");
 }
System.out.println("Stmt2");
 }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
Stmt1
CCE is handled
Stmt2
```

Class Hierarchy Casting Controls

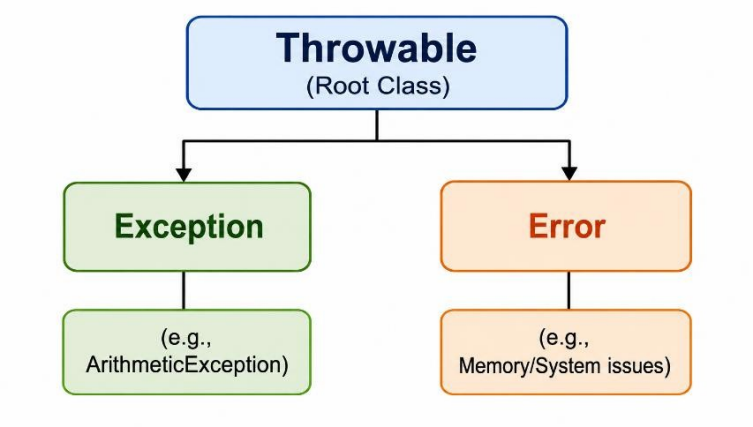
```
// P25: Program demonstrating ClassCastException in Java with inheritance
// Invalid runtime cast from superclass to subclass triggers exception.

class Animal
 { int a;
}
class Dog extends Animal {
 // subclass properties
}
class Test {
public static void main(String[] args)
{ System.out.println("Stmt1");
       try {
Animal a1 = new Animal();
 Dog d1 = (Dog) a1; // invalid runtime cast
   }
 catch (ClassCastException cce)
{ System.out.println("Exception is handled");
}
System.out.println("Stmt2");
  }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
Stmt1
Exception is handled
Stmt2
```

Multi-Catch Block Order Rules & Inheritance Root

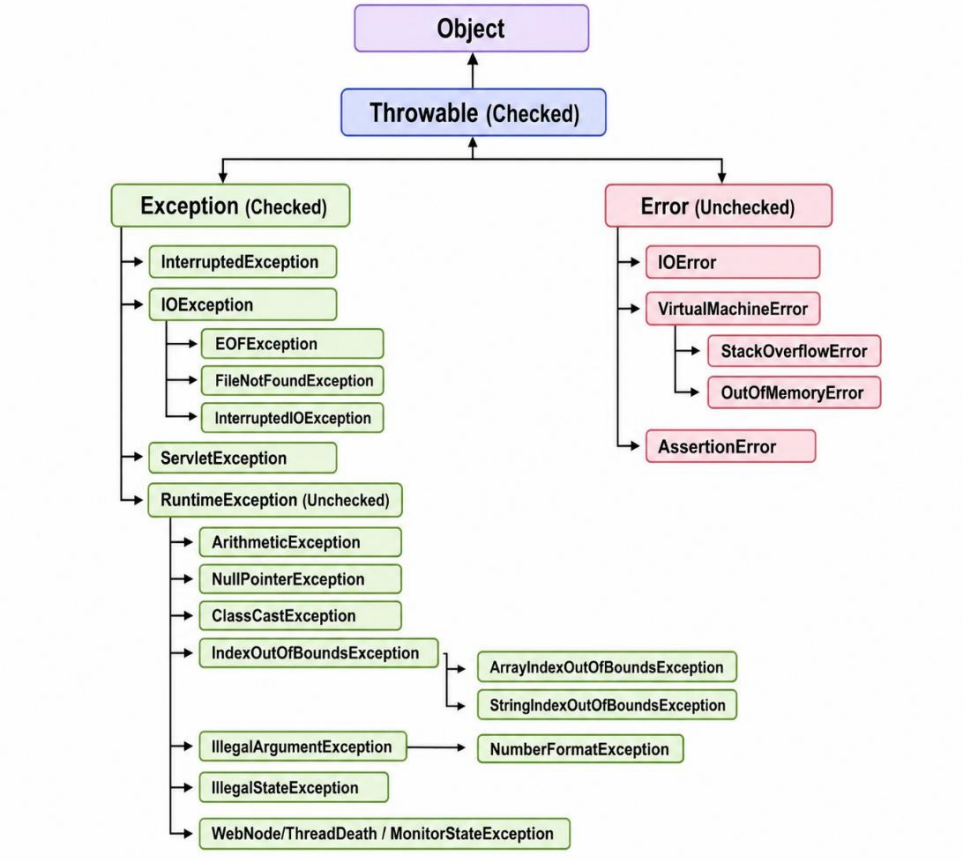
Try with Multiple Catch Blocks

- A single try block can be followed by any number of catch blocks.
- Crucial Ordering Rule: When structuring multiple catch blocks, the type hierarchy order must always go from Child to Parent.
  Predefined Exception Class Hierarchy



Exception: Created by the program execution flow and is generally **recoverable**.
Error: Typically caused by external environment or system dependencies and is **not recoverable**.

Predefined Exception Class Hierarchy Diagram



Types of Exceptions

Exceptions in Java are broadly divided into two main categories:

1. **Checked Exception**
2. **Unchecked Exception**

Checked Exceptions

- These are exceptions that are checked by the compiler at compile-time to ensure smoother execution at runtime.

- The programmer must handle these exceptions explicitly using try-catch blocks or declare them using the throws keyword, otherwise, the program will fail to compile.
- Examples:
  - InterruptedException
  - IOException
  - FileNotFoundException

Unchecked Exceptions

- These exceptions are not checked by the compiler at compile-time. They occur purely during runtime.
- RuntimeException and all of its subclasses are categorized as unchecked exceptions.
- **Total Count:** There are exactly **18 unchecked exceptions** in the standard exception hierarchy. Any exception class outside of these 18 is classified as a checked exception.
- Additionally, **Error and all its subclasses** are also classified as unchecked exceptions because they represent critical conditions outside the application's control.

Try with Multiple Catch Blocks:

Thewayofhandlinganexceptionisvariedfromexceptiontoexception. Henceforevery exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.

```
try {
   // Risky code that may throw multiple types of exceptions
}
catch (ExceptionType1 e1) {
   // Handling code 1
}
catch (ExceptionType2 e2) {
   // Handling code 2
}
```

If try with multiple catch blocks present then order of catch blocks is very important. Itshouldbefromchildtoparent bymistakeifwearetakingfrom parent to child then we will get Compile time error saying.

Example:

```
// P26: Program demonstrating multiple catch blocks in Java
// Specific child exceptions first, followed by general parent exception.

class Test {
public static void main(String[] args)
  { try {
 String s = null;
System.out.println(s.length()); // throws NullPointerException
 }
```

```
 // 1. Specific child exceptions first
catch (ArithmeticException ae)
{ System.out.println("AE is
handled");
  }
 catch (ArrayIndexOutOfBoundsExceptionaiobe)
{ System.out.println("AIOBE is handled");
}
  catch (NullPointerException npe) {
System.out.println("NPE is handled"); // executes successfully
}
 // 2. General parent exception last (acts as backup)
catch (Exception e) {
System.out.println("Any other Exception is handled");
 }
 }
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
NPE is handled
```

Finally Block

- It isnotrecommendedtotakecleanupcodeinsidetryblockbecausethereisno guarantee for the execution of every statement inside a try.
- Itisnotrecommendedtoplacecleanupcodeinsidecatchblockbecauseifthere is no exception then catch block won't be executed.
- Werequiresomeplacetomaintaincleanupcodewhichshouldbeexecuted always irrespective of whether exception raised or not raised and whether handledornot handled. Suchtypeofbestplaceisnothingbutfinallyblock.
- Hencethemainobjectiveoffinallyblockistomaintaincleanupcode.

Syntax:

```
try
{
riskycode
}
catch(xe)
{
handlingcode
}
```

finally
{
cleanupcode
}
Thespecialityoffinallyblockisit willbeexecuted alwaysirrespectiveofwhetherthe exception raised or not raised and whether handled or not handled.

Prog 1: IfthereisnoException

```
// P27: Program demonstrating try-catch-finally in Java
// Finally block always executes regardless of exception occurrence.

class Test {
  public static void main(String[] args)
 { try {
System.out.println("try block executed");
}
catch (ArithmeticException e)
{ System.out.println("catch block executed");
 }
finally {
System.out.println("finally block executed");
   }
}
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
try block executed
finally block executed
```

Prog2: Ifanexceptionraisedbutthecorrespondingcatchblock matched:

```
// P28: Program demonstrating try-catch-finally in Java with exception
// Division by zero triggers ArithmeticException, catch handles it, finally always executes.

class Test {
 public static void main(String[] args)
   { try {
System.out.println("try block executed");
System.out.println(10 / 0); // ArithmeticException
```

```
 }
catch (ArithmeticException e)
{ System.out.println("catch block
executed");
 }
 finally {
System.out.println("finally block executed");
}
 }
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
try block executed
catch block executed
finally block executed
```

Prog 3: Ifanexceptionraisedbutthecorrespondingcatchblock not matched:

```
// P29: Program demonstrating try-catch-finally in Java
// Catch block does not match thrown exception, but finally still executes.

class Test {
 public static void main(String[] args)
 { try {
System.out.println("try block executed");
System.out.println(10 / 0); // ArithmeticException
}
 catch (NullPointerException e)
{ System.out.println("catch block executed"); // not
executed
}
finally {
System.out.println("finally block executed");
}
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
try block executed
finally block executed
Exception in thread "main" java.lang.ArithmeticException:
/ by zero
        at Test.main(Test.java:8)
```

Nested try-catch Blocks: Inside a try, catch, or finally block, we can write another complete try-catch-finally block.

- Rule 1 (Outer Exception): If an exception is raised in the outer try block, then only the outer catch block can handle it.
- Rule 2 (Inner Exception): If an exception is raised in the inner try block, it can be handled by either the inner catch or the outer catch block.
- Priority Rule: First priority is always given to the inner catch. If the inner catch block does not match the exception type, it falls back to use the outer catch block.

Program 1: Exception Raised in Outer Try Block
In this scenario, an exception occurs in the outer try block before the runtime execution ever reaches the inner try block.

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

      try {  // Outer try
System.out.println("Outer try block executed");

System.out.println(10 / 0);   // Exception occurs here

         // These statements are never executed
         try {
System.out.println("Inner try block executed");
         }
         catch (ArithmeticException e)
{ System.out.println("Inner catch block executed");
         }
         finally {
System.out.println("Inner finally block executed");
         }

      }
      catch (ArithmeticException e) {  // Outer catch
System.out.println("Outer catch block executed");
```

```
    }
    finally {  // Outer finally
System.out.println("Outer finally block executed");
    }
  }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
Outer try block executed
Outer catch block executed
Outer finally block executed
```

Program 2: Exception Raised in Inner Try Block

In this scenario, the outer try block runs fine initially, and the exception is thrown inside the inner try block. The inner catch block successfully matches and handles it.

```
// P30: Program demonstrating nested try-catch-finally in Java
// Inner try-catch-finally executes first, then control moves outward.

class Test {
public static void main(String[] args)
{ try
{ // Outer try
System.out.println("Outer try");
        try
{ // Inner try
System.out.println("Inner try");
System.out.println(10 / 0); // throws ArithmeticException
}
 catch (ArithmeticException ae) { // Inner catch handles it
System.out.println("Inner catch");
} finally { // Inner finally executes before moving out
System.out.println("Inner finally");
}
} catch (ArithmeticException ae) { // skipped since already handled
System.out.println("Outer catch");
    }
```

```
  }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
Outer try
Inner try
Inner catch
Inner finally
```

The throw Keyword

- The throws keyword in Java is used in a method signature to declare that the method has a chance of raising one or more specific exceptions during its execution.
- Instead of handling the exception locally within the method using a try-catch block, the method uses throws to delegate the responsibility of exception handling to its calling method.
- If the exception travels all the way down the call stack to the main method and is still declared via throws rather than caught, it is delegated to the JVM's default exception handler, which prints the exception information and terminates the program.

A. Implicit vs. Explicit Throwing
When a runtime exception happens naturally (like dividing by zero), the JVM implicitly creates the exception object and passes it to the default exception handler.
Implicit Exception (By JVM):

```
class Test {
   public static void main(String[] args) {
System.out.println(10 / 0); // JVM automatically creates and throws ArithmeticException
   }
}
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Explicit Exception (Using throw): You can mimic this behavior by manually creating the object with the new keyword and passing a custom message string.

```
class Test {
   public static void main(String[] args) {
      throw new ArithmeticException("/ by zero"); // Replaces implicit creation
   }
}
```

Unreachable Code Error

If you place any statements directly after a throw statement within the same execution path, they will never be reached. The Java compiler catches this and throws an error.

```
class Test {
   public static void main(String[] args) {
      throw new ArithmeticException("/ by zero");
System.out.println("Hello"); // Compile-time Error: unreachable statement
   }
}
```

Exception Rules & The throws Keyword

Java divides exceptions into two primary types, regulated by specific rules:
Comparison Table: Checked vs. Unchecked Exceptions

	CHECKED EXCEPTIONS	UNCHECKED EXCEPTIONS
Definition	Checked by the compiler at compile-time.	Not checked by the compiler at compile-time; occurs at runtime
Hierarchy	Subclasses of Exception except RuntimeException and its descendants	RuntimeException, Error, and all of their respective subclasses
Compiler Enforcement	Mandatory. The compiler forces you to address them.	Optional. The compiler does not force you to handle or declare them.
Handling Rule	Must be either handled (try-catch) or declared (throws keyword).	Can be optionally handled using try-catch, but not required.
Common Examples	InterruptedException IOException SQLException	ArithmeticException NullPointerException ArrayIndexOutOfBoundsException
Typical Cause	External factors outside the program's direct control (e.g., missing files).	Programming logic flaws or bugs (e.g., dividing by zero).

Priority Rule: If a method both handles (via try-catch) and declares (via throws) the same checked exception, handling via the catch block takes full priority.

Handling vs. Declaring Checked Exceptions

The Thread.sleep(long millis) method throws a checked exception named InterruptedException. If you do not address it, the code will fail to compile.

The "Calling" vs. "Called" Method Relationship

When working with checked exceptions across multiple methods, the relationship is defined as follows:

- **Called Method:** The method containing the code that inherently triggers or explicitly throws the checked exception.
- **Calling Method:** The method initiating the execution of the called method.

Whenever a **called method** signs its method declaration with a throws clause, the **calling method** must consciously choose to either catch that exception using try-catch or re-declare it using its own throws clause.

Programming Examples

Example A: The Problem (Compile-Time Error)

The built-in method Thread.sleep(long ms) is defined inside the Thread class with a throws InterruptedException signature. Because InterruptedException is a checked exception, calling it blindly causes a compilation failure:

```
class Test {
   public static void main(String[] args)
{ Thread.sleep(1000);
   }
}
```

```
c:\nrcm>javac Test.java
Test.java:3: error: unreported exception InterruptedExcept
ion; must be caught or declared to be thrown
             Thread.sleep(1000);
                          ^
1 error
```

Example B: Fixing via Declaration (throws)

By adding throws InterruptedException to the main method signature, the error is resolved. The main method effectively delegates the exception back to the JVM.

```
class Test {
   // Declaring the exception to avoid compiler error
   public static void main(String[] args) throws InterruptedException
         { Thread.sleep(1000);
   }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
```

Example C: Exception Propagation Across the Call Stack

When an exception isn't caught locally, it propagates down through successive calling methods if every method in the execution path declares it using the throws keyword.

```
class Test {
   // 4. main method delegates the exception to the JVM
   public static void main(String[] args) throws InterruptedException
         { doStuff();
   }
   // 3. doStuff delegates the exception to main
   static void doStuff() throws InterruptedException
         { doMoreStuff();
   }
   // 2. doMoreStuff delegates the exception to doStuff
   static void doMoreStuff() throws InterruptedException {
      // 1. Thread.sleep throws InterruptedException
         Thread.sleep(1000);
   }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
```

Example D: Priority Rule (Both Handled and Declared)

If a method contains a throws declaration but wraps the risky call in a try-catch block anyway, the local catch block takes over execution.

```
class Test {
   public static void main(String[] args) throws InterruptedException
      { try {
                  Thread.currentThread().interrupt();
                  Thread.sleep(1000);
      }
      catch (InterruptedExceptionie) {
                  System.out.println("InterruptedException is handled");
      }
   }
}
```

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
InterruptedException is handled
```