Object Oriented Programming (with JAVA) (UNIT-2)

by
Dr. Partha Roy,
Associate Professor,
Bhilai Institute of Technology, Durg

UNIT- II

Concrete class, Abstract class, Interface, Inner classes. Aggregation, Composition and Inheritance, super method and reference. Method overloading and overriding. Singleton classes. Package concepts. Exception Handling: Inbuilt, User defined, Checked and Unchecked.

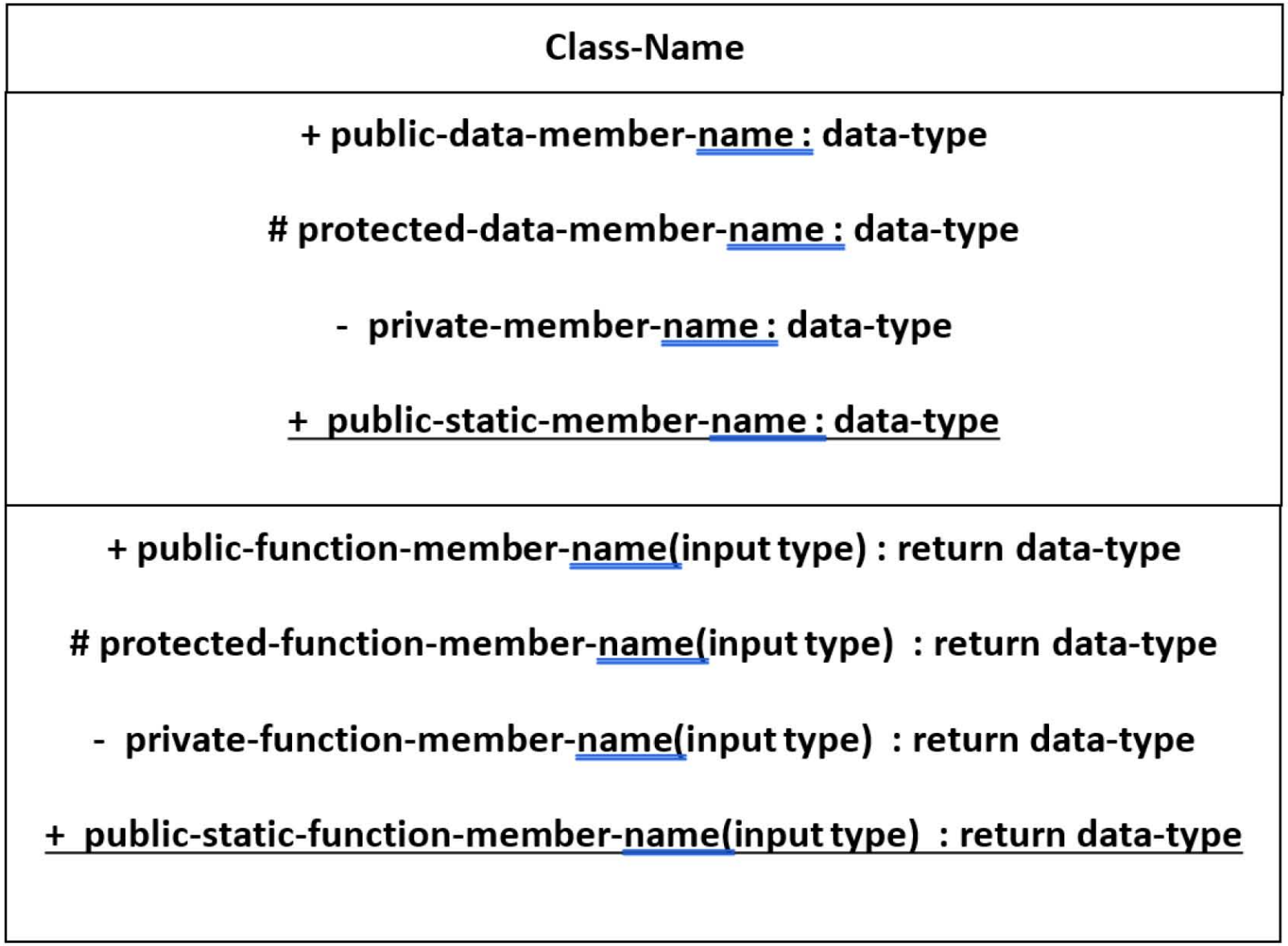
Class Notation (UML: Unified Modeling Language)

UML class is represented by the diagram.

The diagram is divided into three parts.

- The top section is used to name the class.
- The second one is used to show the attributes (data-members) of the class.
- The third section is used to describe the operations (methods or behaviour) performed by the class.

Class Notation (UML)



Concrete class

- Has data members and member methods.
- All member methods are defined.
- Has constructors.
- Can be instantiated using new operator and constructor methods.
- The keyword “extends” is used to inherit a concrete class into another class.
- Any access modifier can be associated with any member (data as well as methods).

Concrete class example

```
class A{
    private int n;
    public void setN(int n){ this.n=n; }
    public void getN(){ System.out.println("n= "+n); }
    public A(){ System.out.println("Default Constructor"); n=0; }
}
public class MyClass {
    public static void main(String args[]) {
      A ob1=new A();
      ob1.setN(100);
      ob1.getN();
    }
}
```

Output:

```
Default Constructor
n= 100
```

Activity:
Create a parameterized constructor and create instances using it.

Abstract class

- Has data members and member methods.
- The keyword “abstract” has to be used along with the class declaration.
- All or Some of the member methods may be defined. Member methods without definition body should be declared as abstract.
- If all the member methods are defined in the class declared as abstract, then the inherited child class does not become abstract.
- Can NOT be instantiated using new operator and constructor methods.

Abstract class

- Has constructors.
- Any access modifier can be associated with any member (data as well as methods).
- The keyword “extends” is used to inherit an abstract class.
- The child class of an abstract class also becomes abstract unless it overrides and defines all the inherited abstract methods.
- When a class inherits an abstract class and when the child class instance is created then inside it an unnamed instance of the parent abstract class is created which calls the constructor of the abstract parent class.
- The child class can use the “super()” to call the abstract parent class constructor.

Abstract class example

```
abstract class A //class containing abstract method so need to declare abstract
{
   private int n;
   public abstract void setN(int n);
        //missing method body hence declared abstract
   public void getN(){ System.out.println("n= "+n); }
   public A(){ System.out.println("Default Constructor"); n=0; }
}
public class MyClass {
   public static void main(String args[]) {
     //A ob1=new A(); //error: A is abstract; cannot be instantiated
     //ob1.getN();
   }
}
```

Note:
Remember how we created a pure virtual function in C++, abstract method in Java is same as that. Virtual keyword is not there in Java.

Interface

- An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a concrete class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that its sub-class implements.
- A class that implements the interface becomes abstract until all the methods of the interface are defined (over-ridden) in the child class to make the child class concrete.
- An interface is implicitly abstract. We do not need to use the abstract keyword when declaring an interface.

Interface

- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods and data members in an interface are implicitly public.
- The prototype-signature of the interface method should be maintained when overriding the methods of the interface in the child classes.
- However, an interface is different from a concrete class in several ways, including:
  - We cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All the member-methods in an interface are compulsorily ***public abstract***.
  - All the data-members in an interface are compulsorily ***public, static and final***.
  - To inherit an interface into a class, **implements** keyword is used.
  - An interface can **extend** multiple interfaces (**extends** keyword).

Interface example-1

```
interface Animal {
   public void eat();
   public void travel();
}

public class Mammal implements Animal{
   public void eat(){
      System.out.println("Mammal eats");
   }
   public void travel(){
      System.out.println("Mammal travels");
   }
   public int noOfLegs(){
      return 0;
   }
   public static void main(String args[]){
      Mammal m = new Mammal();
      m.eat();
      m.travel();
   }
}
```

Output:
Mammal eats
Mammal travels

Note:
implements keyword is used to inherit an interface into a class

Interface example-2

```
interface I1
{
    public void test();
}
interface I2
{
    public void test2();
}
interface I3 extends I1,I2
{
    public void test3();
}
```

Note:
extends keyword is used to inherit an interface into another interface

Comparison between Abstract class and Interface

Interface	Abstract class
➢ When only requirement specification is given but nothing is known about implementation	➢ When only partial implementation is described in the requirement specification.
➢ Every method can only be public and abstract, whether explicitly mentioned or not and nothing else	➢ There can be a mix of abstract and concrete methods with different access modifiers. Also it is NOT compulsory to have any abstract method.
➢ As methods are abstract so they cannot be final or static or synchronized or anything else	➢ Those methods which are concrete can be declared final or static or anything else.
➢ Every data member is always public static and final, whether explicitly mentioned or not and nothing else	➢ No restrictions on data members.
➢ As data members are final so initialization is compulsory	➢ Only data members declared as final need to be initialized.
➢ No instance-block and static-block allowed	➢ Instance and static blocks allowed
➢ No constructors allowed	➢ Constructors are allowed

Inner classes

- An inner class or nested class, is a class that is declared within the scope of another class or another method.
- There are four types of inner classes:
  - **1. Regular/normal inner class**
  - **2. Method local inner class**
  - **3. Anonymous inner class**
  - **4. Static nested class**
- Inner or nested classes demonstrate the concept of **Composition or Aggregation**.
- Example: Department cannot exist without College, so here we can make College as the outer class and Department as the inner class.

1. Regular/normal inner class

- Regular/normal inner class is a class which is defined inside the scope of another class.
- Syntax:

```
class Outer-class-name
{   //starting scope of outer class
    class Inner-class-name
    {

    }
} //ending scope of outer class
```

Compiling inner class

- If we have an outer class that has n-number of inner classes then after compilation we will get class files of each inner class separately in the format:

"**Outer-class$Inner-class.class**" also we will have the class file of the outer class "**Outer-class.class**"

- When we compile any java file, it creates class files of every class that we declare in that java file.

Creating instance of inner class

- As inner class exists within outer class so we need to create and use the outer class object-instance to create the instance of inner class.
- **Syntax:**
  - **Firstly: Outer ob=new Outer();**
  - **Then: Outer.Inner ib=ob.new Inner();**
  - **Or: Outer.Inner ib=new Outer().new Inner();**

Members of outer and inner class

- We can have static declarations in Outer class.
- We cannot have static declarations for member methods in Inner classes.
- We can have static declarations for data members in inner classes but they should be declared **final**.
- The inner class has full access to all the members of outer class, but outer class does not have any access to inner class members.
- From inner class we can access all the outer class instance members using the following format:
  - **Member-name;** (when member name is unique in both outer and inner classes)
  - **Outer.this.member-name;** (when inner class has the same member name as in the outer class)
- Nesting of inner classes to any level is allowed.

Modifiers applicable

- **For Outer class:**
  - default
  - public
  - final
  - abstract
- **For Inner classes:**
  - default
  - public
  - final
  - abstract
  - *private*
  - *protected*
  - *static*

2. Method local inner class

- When we declare a class within a method is called method local inner class.
- The existence or accessibility of method local inner classes is limited within the execution of the method in which it is being declared.
- When the utility of a class is only limited to a method then we create method local inner classes to serve the purpose.
- It is very useful in saving memory as the lifetime of these classes is limited to the method body only. So when the method executes then only these classes are created and when method execution finishes, these classes also cease to exist.

Method local inner class

- We can define a method local inner class inside instance as well as static member methods.
- A method local inner class can access all the local final variables present inside the function body.
- Method local inner class **inside an instance member method** can access all the static as well as non-static members present in the outer class.
- Method local inner class **inside a static member method** can access **only** the static members present in the outer class and not the instance members.

3. Anonymous inner class

- The purpose of these classes is to instantly override member methods coming from parent class and using the overridden method in few statements.
- It saves memory as an explicit child class is not created, the anonymous class acts as a child class that has no name and its implementation in a local scope.
- Anonymous inner classes are child classes of an existing class.
- The anonymous class cannot define new member methods, it can only be is used to override existing member methods.

3. Anonymous inner class

- They are created while creating the instance of an existing class in the same statement.
- **Syntax:**
  - **Class-name object**=new Class-name(){
    //definition of anonymous inner class
    //overridden of member method(s)
    };
  - **Interface-name object**=new Interface-name(){
    //definition of anonymous inner class
    //overridden of member method(s)
    };
- The **object** created this way would become the child class object of the specified **Class-name** or **Interface-name**.
- Using the **object** we can call the overridden member methods.

4. Static inner class

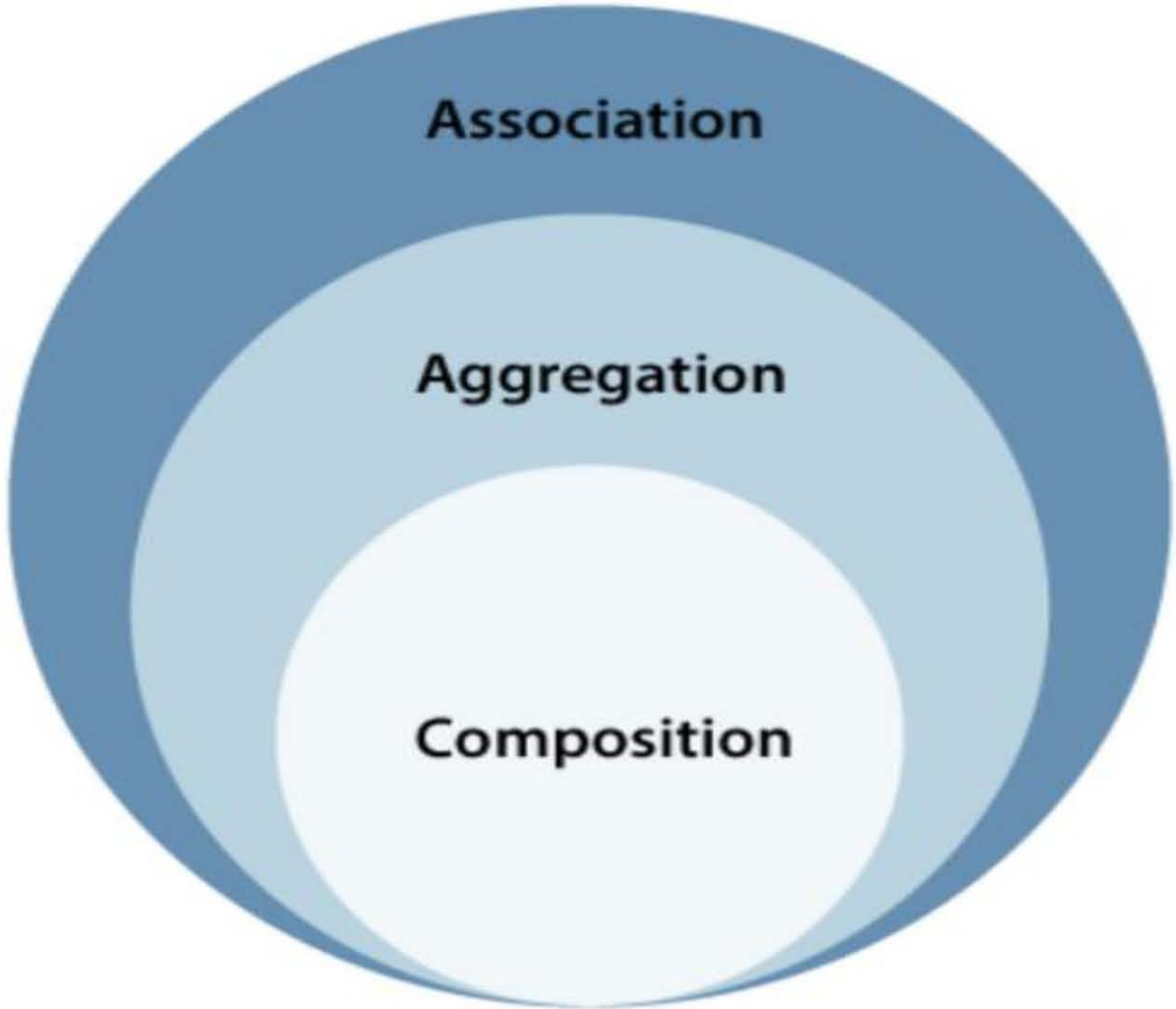
- Static inner class is a class which is defined inside the scope of another class declared as **static**.
- Syntax:

```
class Outer-class-name
{   //starting scope of outer class
    static class Inner-class-name
     {

     }
} //ending scope of outer class
```

- They can only access the static members of the outer class and they cannot access the instance members of the outer class.

Association, Aggregation and Composition



Association, Aggregation and Composition

Association	Aggregation	Composition
➢ Association relationship is represented using an arrow.	➢ Aggregation relationship is represented by a straight line with an empty diamond at one end.	➢ The composition relationship is represented by a straight line with a black diamond at one end.
➢ In UML, it can exist between two or more classes.	➢ It is a part of the association relationship.	➢ It is a part of the aggregation relationship.
➢ It incorporates one-to-one, one-to-many, many-to-one, and many-to-many association between the classes.	➢ It exhibits a kind of weak relationship.	➢ It exhibits a strong type of relationship.

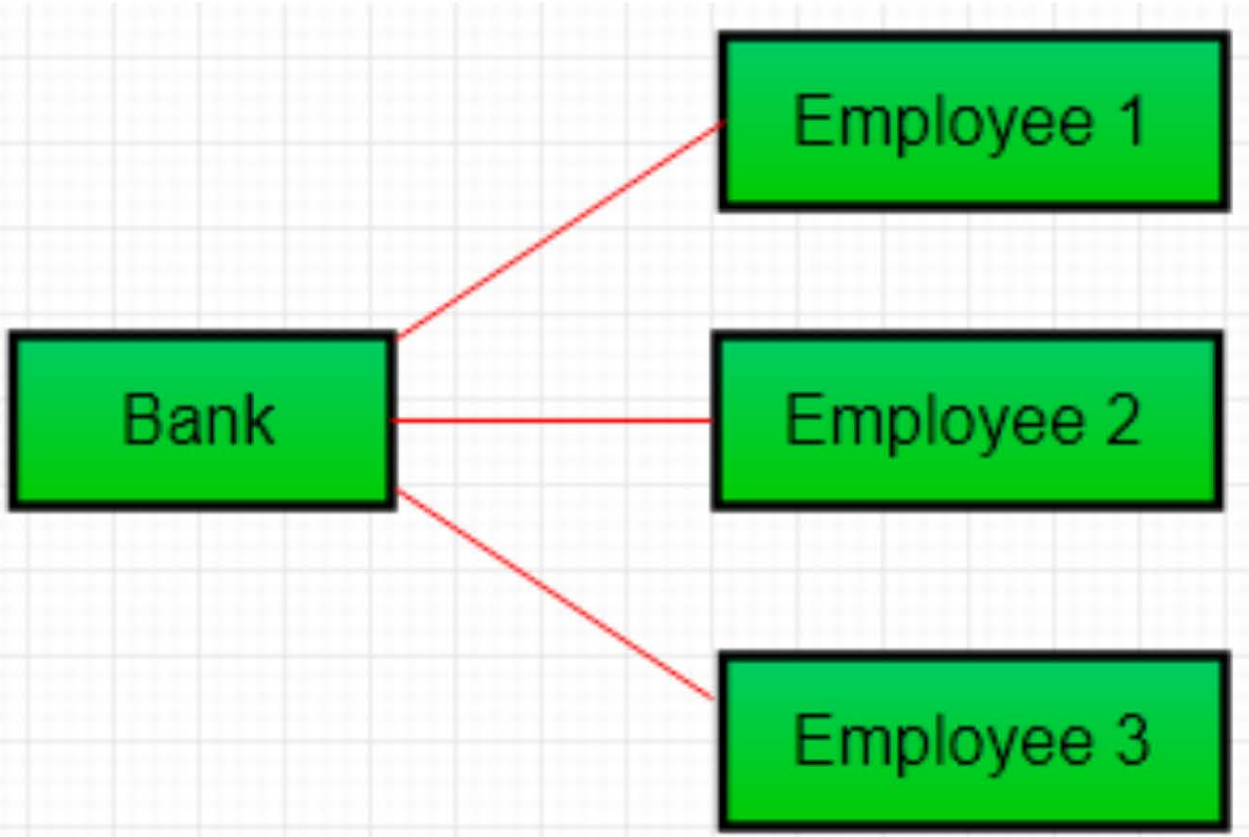
Association, Aggregation and Composition

Association	Aggregation	Composition
➢ It can associate one or more objects together. **(Member objects)**	➢ In an aggregation relationship, the associated objects exist independently within the scope of the system. **(Member objects)**	➢ In a composition relationship, the associated objects cannot exist independently within the scope of the system. **(Member objects)**
➢ In this, objects are linked together.	➢ In this, the linked objects are independent of each other.	➢ Here the linked objects are dependent on each other.
➢ It may or may not affect the other associated element if one element is deleted.	➢ Deleting one element in the aggregation relationship does not affect other associated elements.	➢ It affects the other element if one of its associated element is deleted.
➢ Example: A tutor can associate with multiple students, or one student can associate with multiple teachers.	➢ Example: A car needs a wheel for its proper functioning, but it may not require the same wheel. It may function with another wheel as well.	➢ Example: If a file is placed in a folder and if that folder is deleted then the file residing inside that folder will also get deleted at the time of folder deletion.

Association

Association Code

For example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.

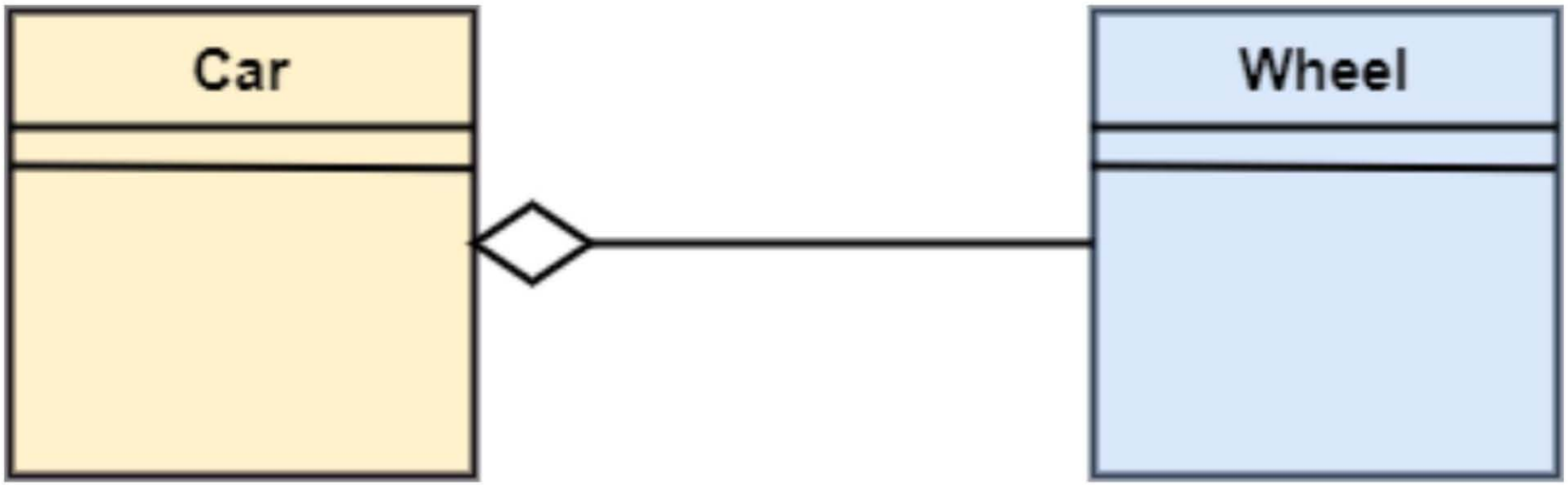


Aggregation

Aggregation Code

In the case of Aggregation, A uses B e.g. A Car uses Wheels.

For example A car cannot move without a wheel. But the wheel can be independently used with the bike, scooter, cycle, or any other vehicle. The wheel object can exist without the car object, which proves to be an aggregation relationship.

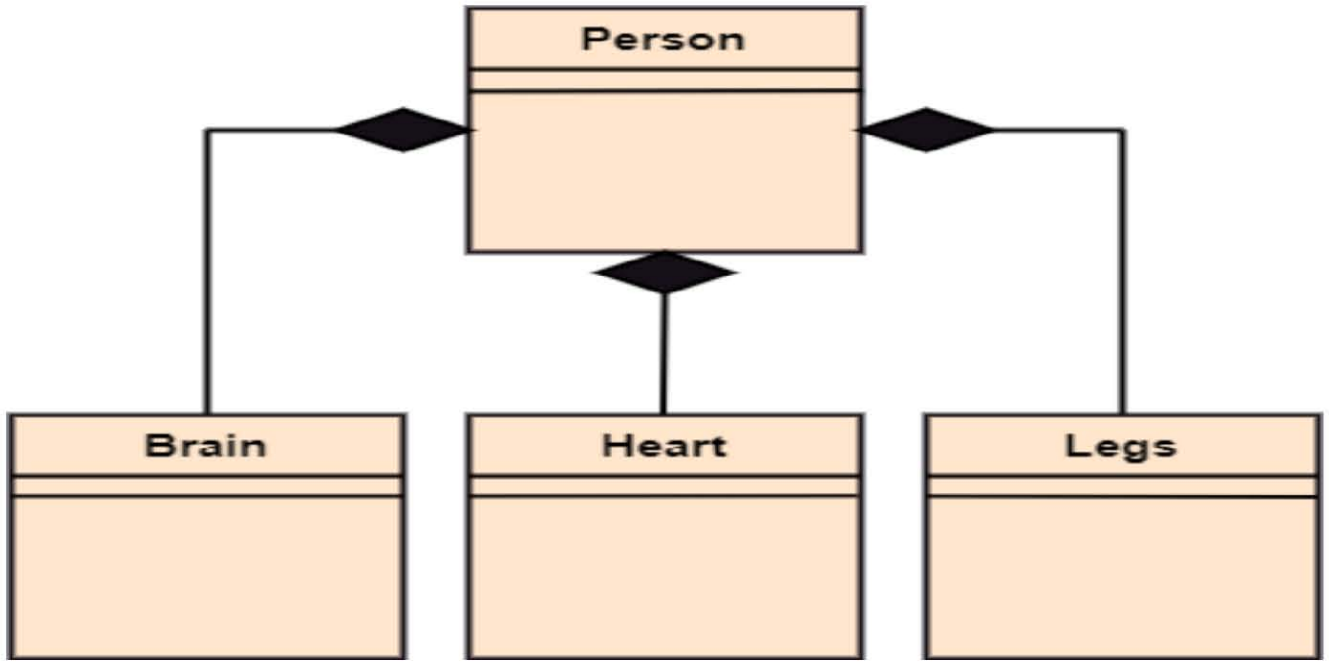


Composition

Composition Code

In the case of Composition A owns B or B is integral part of A e.g. Person is the owner of his Hand, Mind and Heart.

For example the composition association relationship connects the Person class with Brain class, Heart class, and Legs class. If the person is destroyed, the brain, heart, and legs will also get discarded.



Inheritance

- •Reusability can be implemented using Inheritance.
- •Inheritance can be defined as the process where one class acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order. It is also called **IS-A** relation.
- •Where the parent class properties get collected in to the child classes along with the additional individual properties of the child classes then it is called Inheritance. (UML notation is empty triangle shape)
- •The most commonly used keywords for implementing inheritance is **extends** and **implements**.

Inheritance

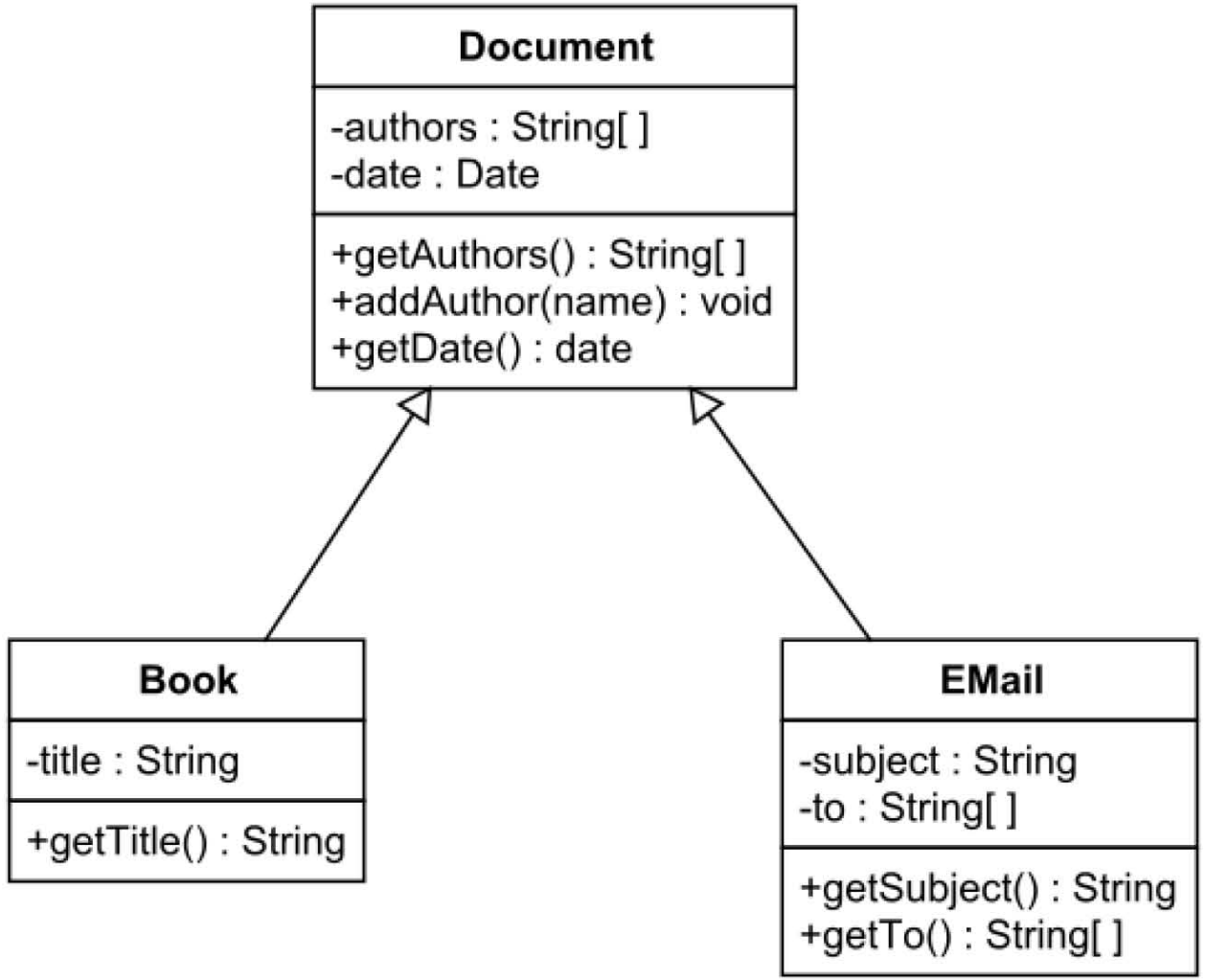
- When class-A is a parent of class-B then we say that class-B **Is-A** child or **Is-A** type of class-A.
- With the use of **extends** keyword the subclasses will be able to inherit all the properties of the super-class except for the private properties of the super-class.
- A very important fact to remember is that Java supports only single class inheritance. This means that a class cannot extend more than one class. So it would be **wrong** to write:

  class Creature extends Animal, Mammal { .... }

- However, a class can implement one or more interfaces. This helps in multiple inheritance.

Inheritance

A class extends another class. For example, the Book class might extend the Document class, which also might include the Email class. The Book and Email classes inherit the fields and methods of the Document class (possibly modifying the methods), but might add additional fields and methods.



Inheritance examples

Inheriting a class

Inheriting an interface

Member access within and outside package

Parent Class in package P1	Private members	Protected members	Default members	Public members
Child-Class present in package P1 and inheriting Parent class from package P1 (Inheritance)	Not Available	Accessible as protected	Accessible as default	Accessible as public
Class present in package P2 and using an object of Parent class from package P1 (Association)	Not Available	Not Available	Not Available	Accessible as public
Child-Class present in package P2, inheriting Parent class from package P1 (Inheritance)	Not Available	Accessible as protected	Not Available	Accessible as public

Basic Inheritance Examples

Inheritance within a Package

Inheritance across Packages

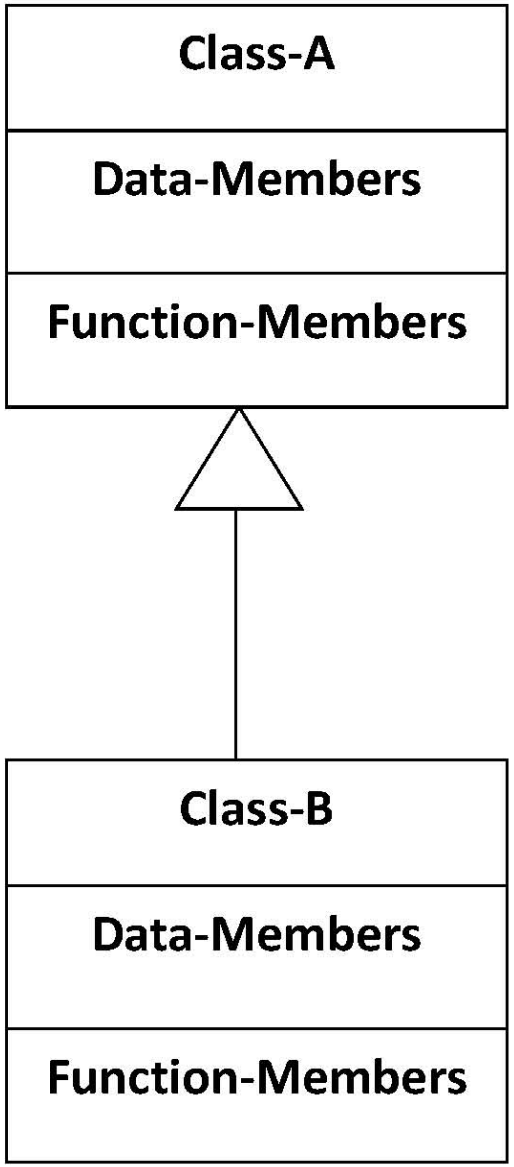
The types of Inheritance supported by Java

1. **Single Level Inheritance.**
2. **Multi Level Inheritance.**
3. **Hierarchical Inheritance.**
4. **Multiple Inheritance** (combining interface and class)

In JAVA the protected and default members can be accessed outside the class but within the package. But in C++ the protected members were not allowed access outside the class.

Single Level Inheritance

Parent class as Concrete class	Parent class as Abstract class	Parent class as Interface
class A {...} class B extends A {...}	abstract class A {...} class B extends A {...}	interface A {...} class B implements A {...}



Single Level Inheritance Code

Static members during inheritance

- During inheritance all the static members get inherited into child class.
- The concept of overriding is followed for static members.
- A static member method in parent class cannot be defined as non-static in child class.

Static members during inheritance

```
class P{
    static int n=100;
    static void test(){System.out.println("P-STATIC-TEST");}
}
class C extends P{
    static int n=200;
    //void test(){}
//compilation error: test() in C cannot override test() in P

  static void test()
   {System.out.println("C-STATIC-TEST");}

    void test2(){
        System.out.println("C-TEST2");
        P.test();
    }
}
```

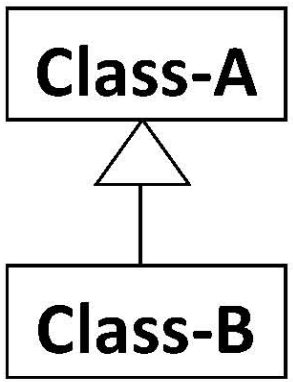
```
public class NewClass {
    public static void main(String arg[]){
        System.out.println(C.n);
//o/p: 200  //overriding the parent class 'n'
        C.test();
            //o/p: C-STATIC-TEST
            //overriding the parent class test()
        C ob1=new C();
        ob1.test2();
            //o/p: C-TEST2
            //       P-STATIC-TEST
    }
}
```

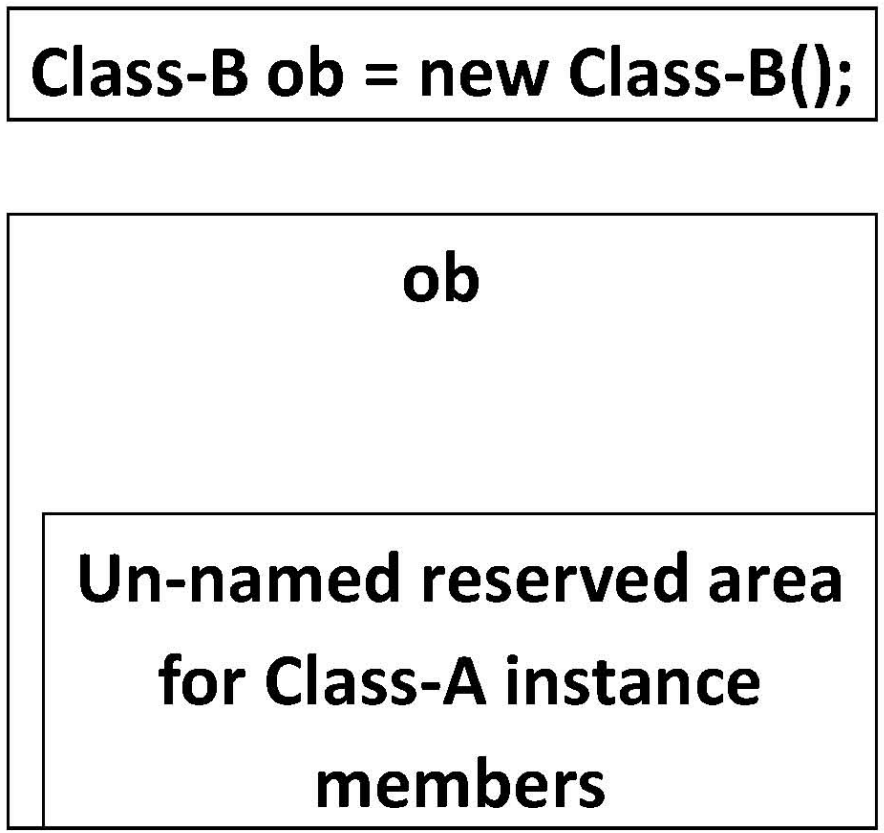
Using super reference and super() method during inheritance

- •During inheritance when we create an instance of the child class then inside the child class instance an un-named area is reserved to store parent class instance members.
- •This un-named reserved area can be accessed by using super reference.
- •Also by using super() method we can invoke any of the parent class constructors.
- •The super() method can only be inside the child class constructor code and also it should be the first statement in the constructor code of the child class.
- •If super() is used to invoke any of the constructors other than the default constructor then the default constructor of the parent class would not be used, instead the parameterized constructor called by the super() would be used to initialize the instance members present in the un-named reserved area.

Using super reference and super() method during inheritance

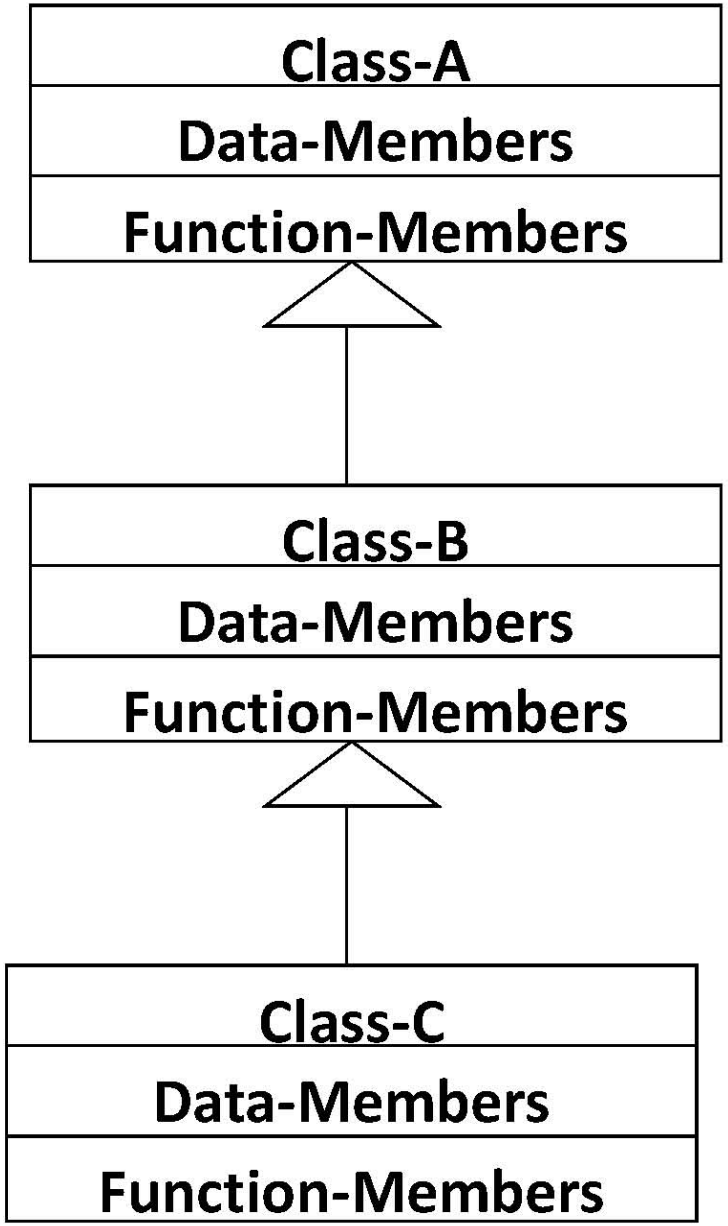
[Code example for super() and super reference]





Multi-level Inheritance

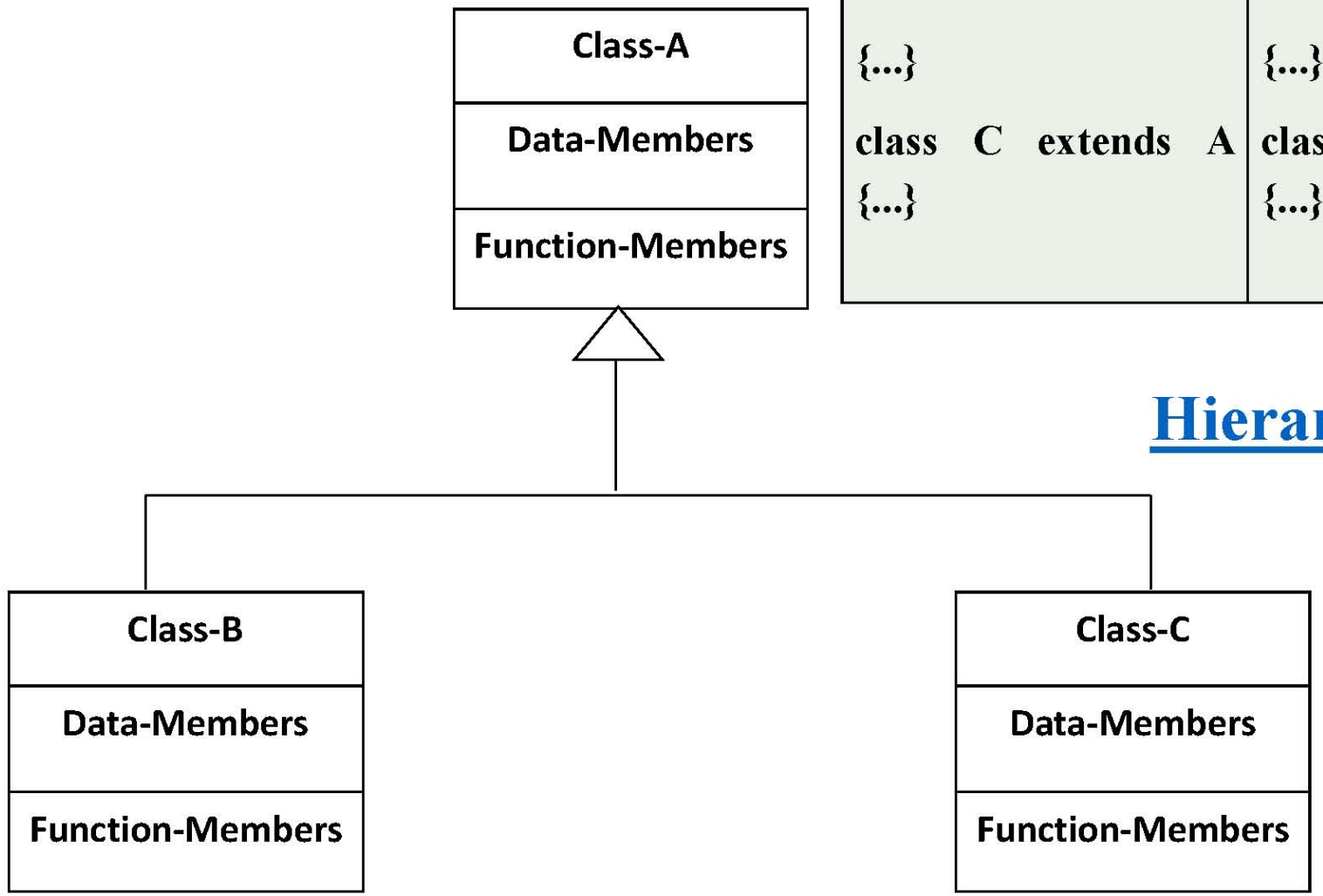
Parent class as Concrete class	Parent class as Abstract class	Parent class as Interface
class A {...} class B extends A {...} class C extends B {...}	abstract class A {...} class B extends A {...} class C extends B {...}	interface A {...} interface B extends A {...} class C implements B {...}



[Multi Level Inheritance Code]

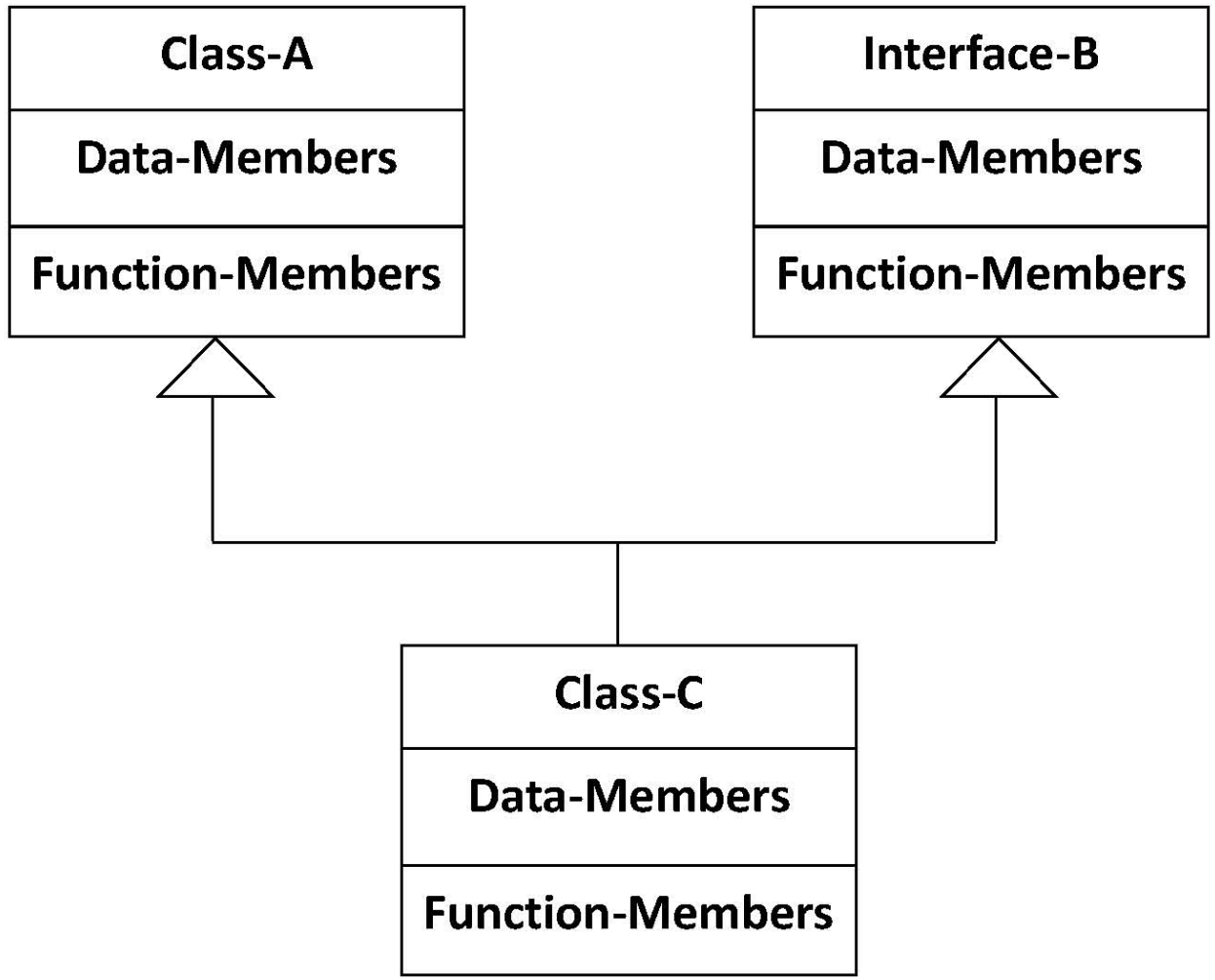
Hierarchical Inheritance

Parent class as Concrete class	Parent class as Abstract class	Parent class as Interface
class A {...} class B extends A {...} class C extends A {...}	abstract class A {...} class B extends A {...} class C extends A {...}	interface A {...} class B implements A {...} class C implements A {...}



[Hierarchical Inheritance Code]

Multiple Inheritance (using class and interface both)

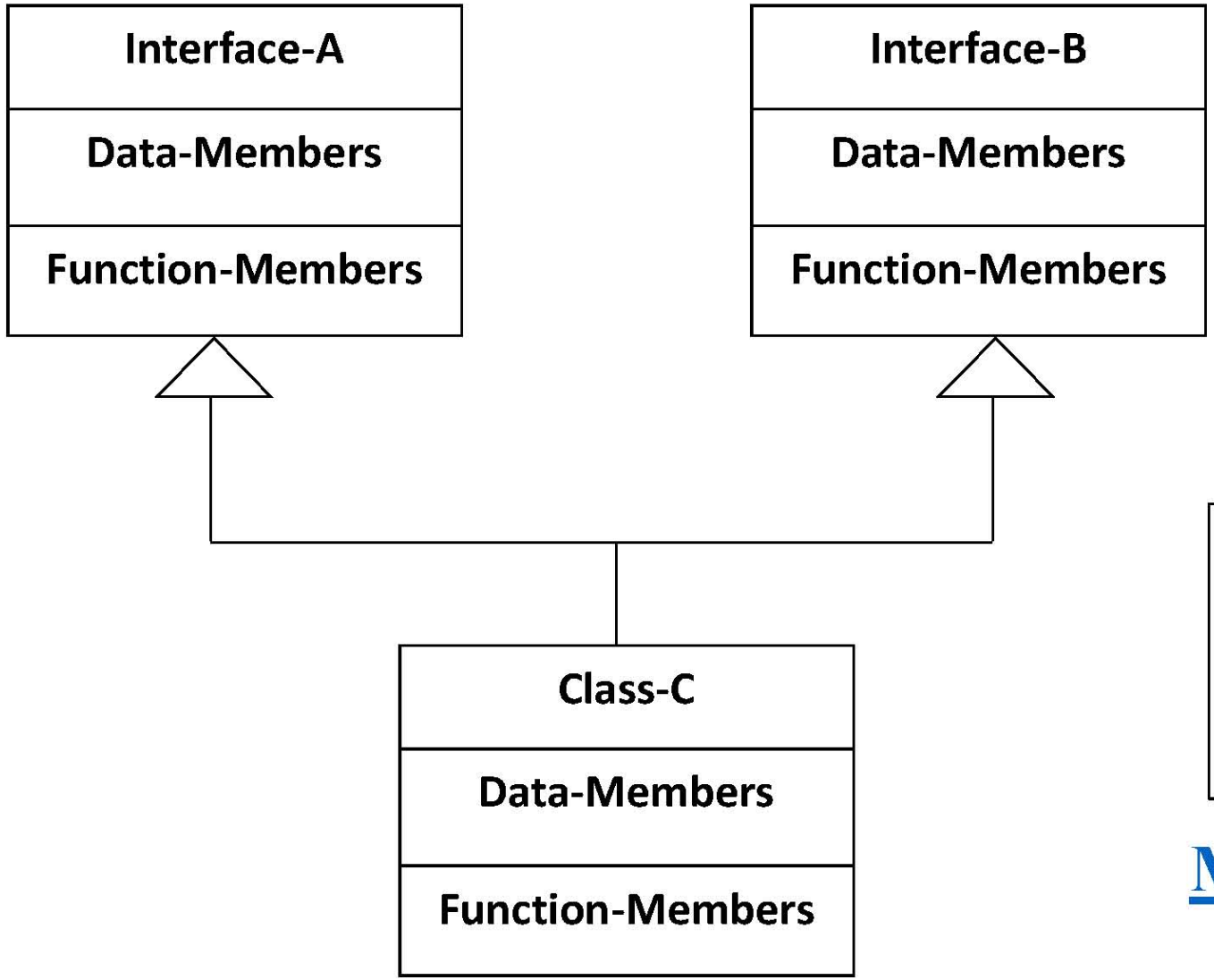


Syntax:

```
class A {...}
interface B {...}
class C extends A implements B
{...}
```

Multiple Inheritance Code-1

Multiple Inheritance (using interfaces)



Syntax:
interface A {...}
interface B {...}
class C implements A, B {...}

Multiple Inheritance Code-2

Method Overloading

- The member methods of class can have the same name but different input argument types or number of arguments should be different.
- Depending upon the actual arguments provided during calling of the method, the appropriate method's definition is invoked.

Method Overloading Example

Method Overriding

- •This can be seen during inheritance only.
- •The child class method overshadows parent class method.
- •The parent class contains a method and at the same time the child class has the same method with the same name, input arguments list and the return type is also same.
- •When the child class object calls that method then the child class version is always executed.
- •In order to call the parent class version of the method, the super keyword need to be used.

Method Overriding

```
class A
{
   public void test()
   {
      System.out.println("test() from Class-A");
   }
}
class B extends A
{
   public void test()
   {
      System.out.println("test() from Class-B");
      super.test();
            //this would help in calling the parent class test() method
            // otherwise the parent class test() could never be called
            // through class-B object.
   }
}
```

```
public class MethodOverriding
{
  public static void main(String arg[])
  {
      B ob=new B();
      ob.test();
  }
}
```

Comparing Overloading and Overriding

Property	Overloading	Overriding
Location	same class	child class
Method names	must be same	must be same
Argument types	must be different or order should be different	must be same including order
Number of arguments	must be different if data type is same and in same order	must be same
Method signature or prototype	must be different	must be same
Return type	does not matter (not considered)	must be same or can be child types
private and final methods	can be overloaded	cannot be overridden

Comparing Overloading and Overriding

Property	Overloading	Overriding
static methods	can be overloaded	the static methods perform method hiding, as they hide the parent class static methods
access modifiers	does not matter (not considered)	should be same as parent or increase the scope of the modifier in the child class, i.e protected in parent then public in child allowed but default or private not allowed in child class
throws clause	does not matter (not considered)	child class method should throw the same or child class exception compared to the parent class version of the method.
polymorphism	static or compile time	dynamic or runtime

Dynamic Binding (Runtime Polymorphism)

- When there is overriding and compiler cannot decide at compile time which method call to be associated with which method definition, then dynamic binding is applied by the compiler.
- The compiler at runtime associates the appropriate method call with its appropriate definition by taking into account the calling object type and input argument types.
- It is like switch-case, here which case the user would select cannot be decided at compile time.
- Inheritance is mandatory.
- The object of the parent class is used to invoke the overridden methods in child classes.

Dynamic Binding Example

Singleton classes

- •When we are allowed to create only a single instance of a class then that class is called singleton class.
- •When several processes having the same requirement then a single object-instance can be created that can be used by all the processes, here singleton class is useful.
- •Memory utilization improves, as many object-instances are not created but a single instance is reused.

Singleton classes

- •Here constructor cannot be used to create instances outside the class, instead factory methods are used.
- •A **factory method** is a member method of the singleton class that returns the same existing instance of the singleton class.
- •Every constructor is declared **private**, so that any instance cannot be created from outside the class.

Singleton classes

- •There has to be **private static** instance of the same singleton class as data-member and it should be initialized by any of the private constructors, usually the default constructor is used.
- •There has to be a **public static factory method** that should return the same static singleton class instance declared as the data member in the class.
- •This factory method will be used to get the singleton class instance.

Singleton classes

- •Ex: ***Runtime class in Java***

```
Runtime r1=Runtime().getRuntime();
                                    //factory method getRuntime()
Runtime r2=Runtime().getRuntime();
                                    //factory method getRuntime()
:
Runtime rn=Runtime().getRuntime();
                                    //factory method getRuntime()
```

- Here only one instance of Runtime will be made available by getRuntime() method, that is assigned to all the objects r1 to rn.
- •Two approaches can be used to create a customized singleton class.

Singleton classes

```
//Approach-1:
class Test
{
     private static Test t1=new Test();
     private Test(){}
     public static Test getTest(){return t1;}
          //factory method
}
class Main {
     public static void main(String arg[]) {
          Test t1=Test.getTest();
          Test t2=Test.getTest();
          :
          Test tn=Test.getTest();
     }
}
```

Singleton classes

// Approach-2:

```
class Test{
	private static Test t1=null;
	private Test(){}
	public static Test getTest() { //factory method
		if(t1==null){t1=new Test();}
	//if t1 is not null then new instance will not be created

		return t1;
	}
}
class Main{
	public static void main(String arg[]) {
		Test t1=Test.getTest();
		Test t2=Test.getTest();
		:
		Test tn=Test.getTest();
	}
}
```

Packages in Java

- •A package is a namespace that organizes a set of related classes and interfaces.
- •Conceptually we can think of packages as being similar to different folders in a computer.
- •Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.
- •The Java platform provides an enormous class library (a set of packages) suitable for use in our own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming.

Exception Handling in Java

- •An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- •When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred.
- •Creating an exception object and handing it to the runtime system is called throwing an exception. After a method throws an exception, the runtime system attempts to find something to handle it.

Exception Handling in Java

- •A program can catch exceptions by using a combination of the **try**, **catch**, and **finally** blocks.
- The **try** block identifies a block of code in which an exception may occur.
- The **catch** block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The **finally** block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.
- •The **try** statement should contain at least one **catch** block or a **finally** block and may have multiple **catch** blocks.

Exception Handling in Java

There are three categories of exceptions:

- **Checked exceptions**
- **Unchecked exception or Runtime exception**
- **Errors**

Checked exceptions

- A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer.
- For example, if a file is to be opened, but the file cannot be found, an exception occurs.
- They are the exceptions that are checked at **compile time**.
- If some code within a method throws a checked exception, then the method must either handle the exception using **try-catch block** or it must specify the exception using **throws** keyword.

Unchecked exception or Runtime exception

- Unchecked are the exceptions that are not checked at compiled time.
- It is up to the programmers to specify or catch the exceptions.
- In Java, exceptions coming under **Error** and **RuntimeException** classes are **unchecked** exceptions, everything else under **Throwable** class is **checked**.

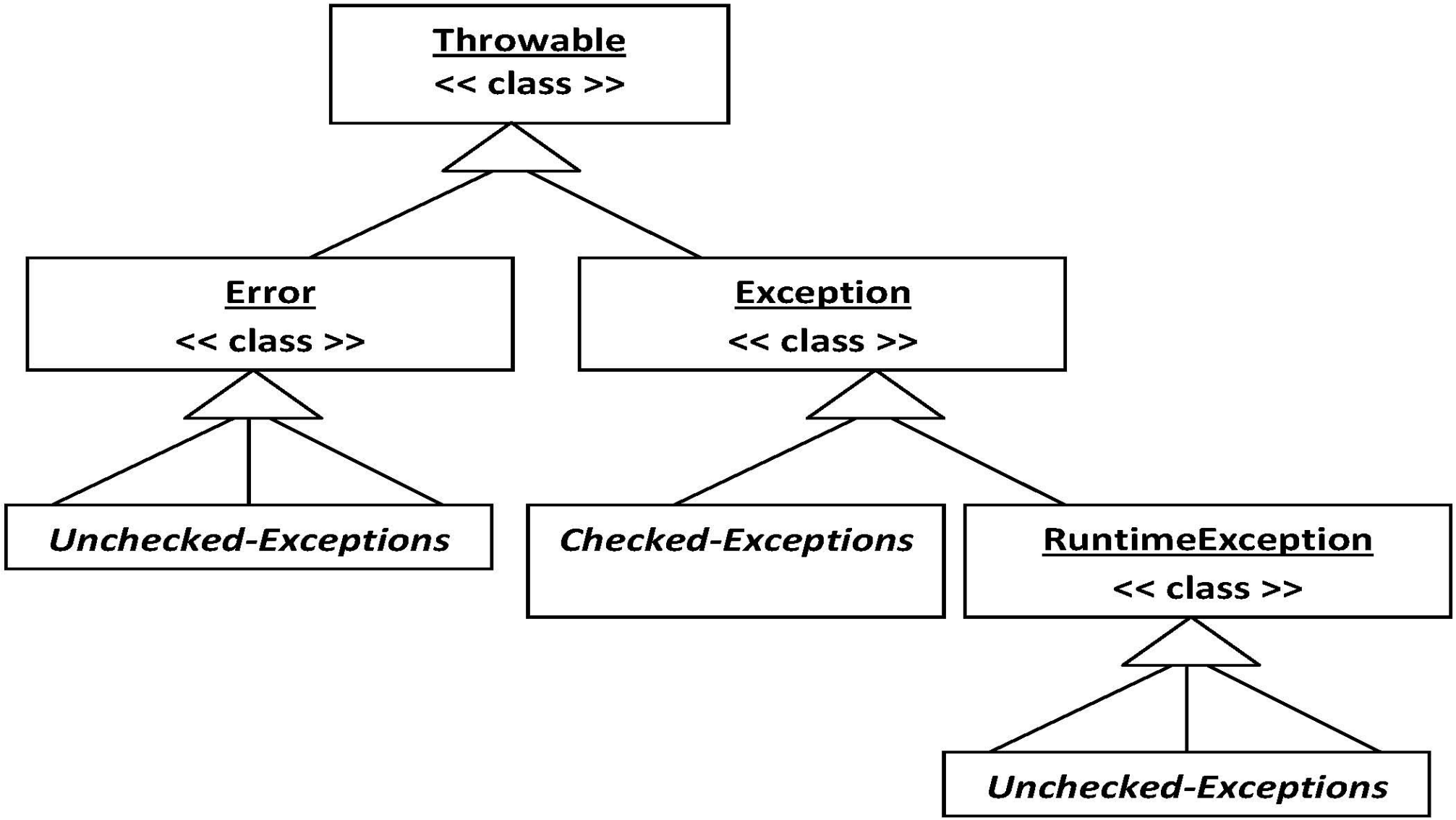
Errors

- These are not exceptions, but problems that arise beyond the control of the user or the programmer.
- Errors are typically ignored in our code because we can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise.
- They are also ignored at the time of compilation.

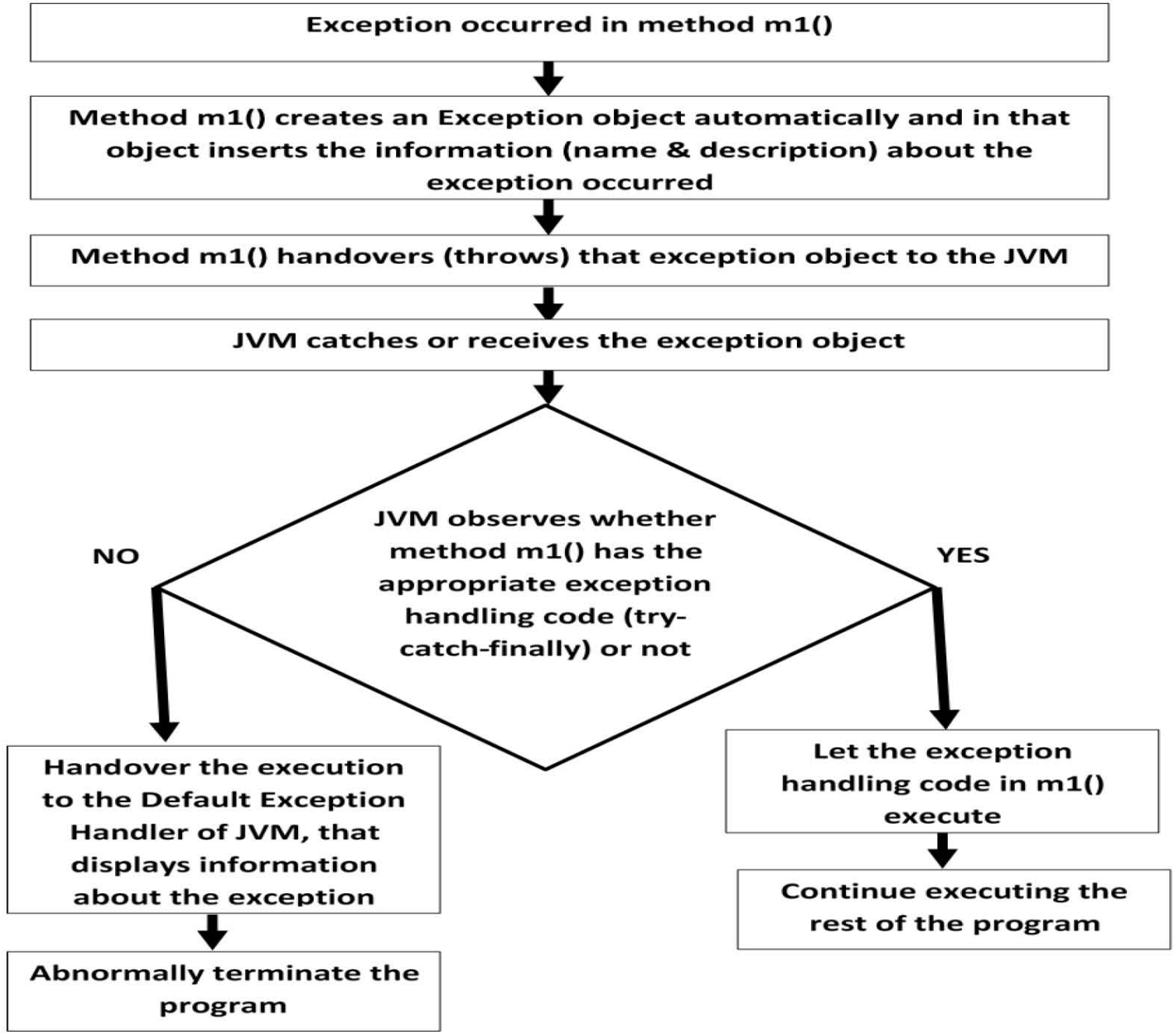
Throwable class

- All exception classes are subtypes of the **java.lang.Exception** class.
- The Exception class is a subclass of the **Throwable** class.
- Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Exception hierarchy



The Exception handling process



About try-catch-finally blocks

- **try-block**: to maintain code that might generate an exception.
- **catch-block**: to maintain exception handling code.
- **finally-block**: to maintain clean-up code.
- **throw-clause**: to handover our created exception object to JVM manually.
- **throws-clause**: to delegate the responsibility of exception handling to the caller.

About try-catch-finally blocks

- A **catch** block cannot exist without a **try** statement.
- It is not compulsory to have **finally** block whenever a **try/catch** block is present.
- The **try** block cannot be present without either **catch** clause or **finally** block.
- Even if an exception occurs or not, the **finally** block is definitely executed.

Exception handling Common Syntax

```
try
{
     //Code that might generate an exception
}catch(ExceptionType1 e1){
     //Catch block code
}catch(ExceptionType2 e2){
     //Catch block code
}catch(ExceptionType3 e3){
     //Catch block code
}finally{
     //The finally block always executes and is also
optional.
}
```

Examples of Inbuilt exceptions

- Inbuilt checked exception example
- Inbuilt un-checked exception example

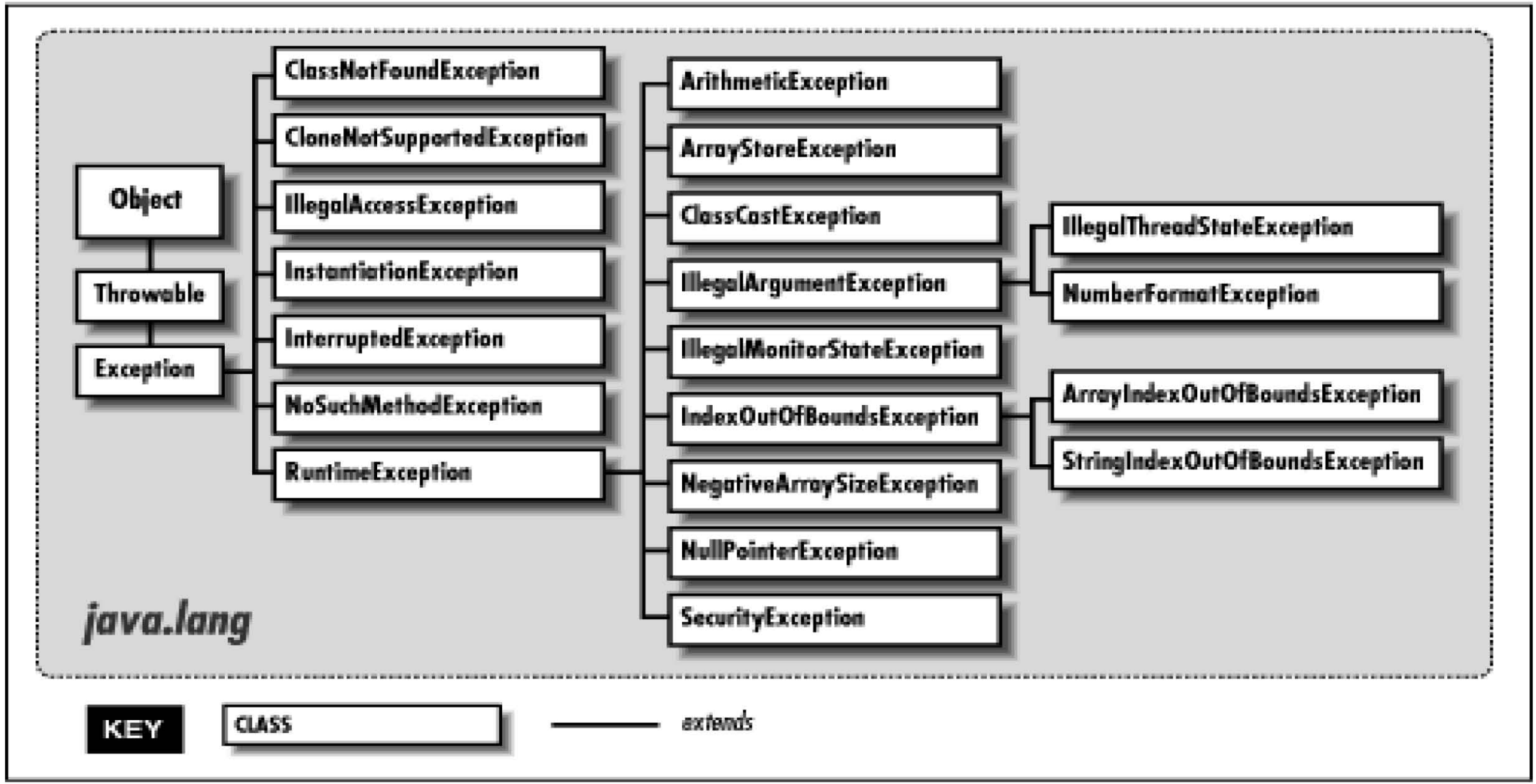
Examples of exception handling

- Example when an exception occurs
- Example when no exception occurs
- Example when an exception occurs which is handled by the JVM

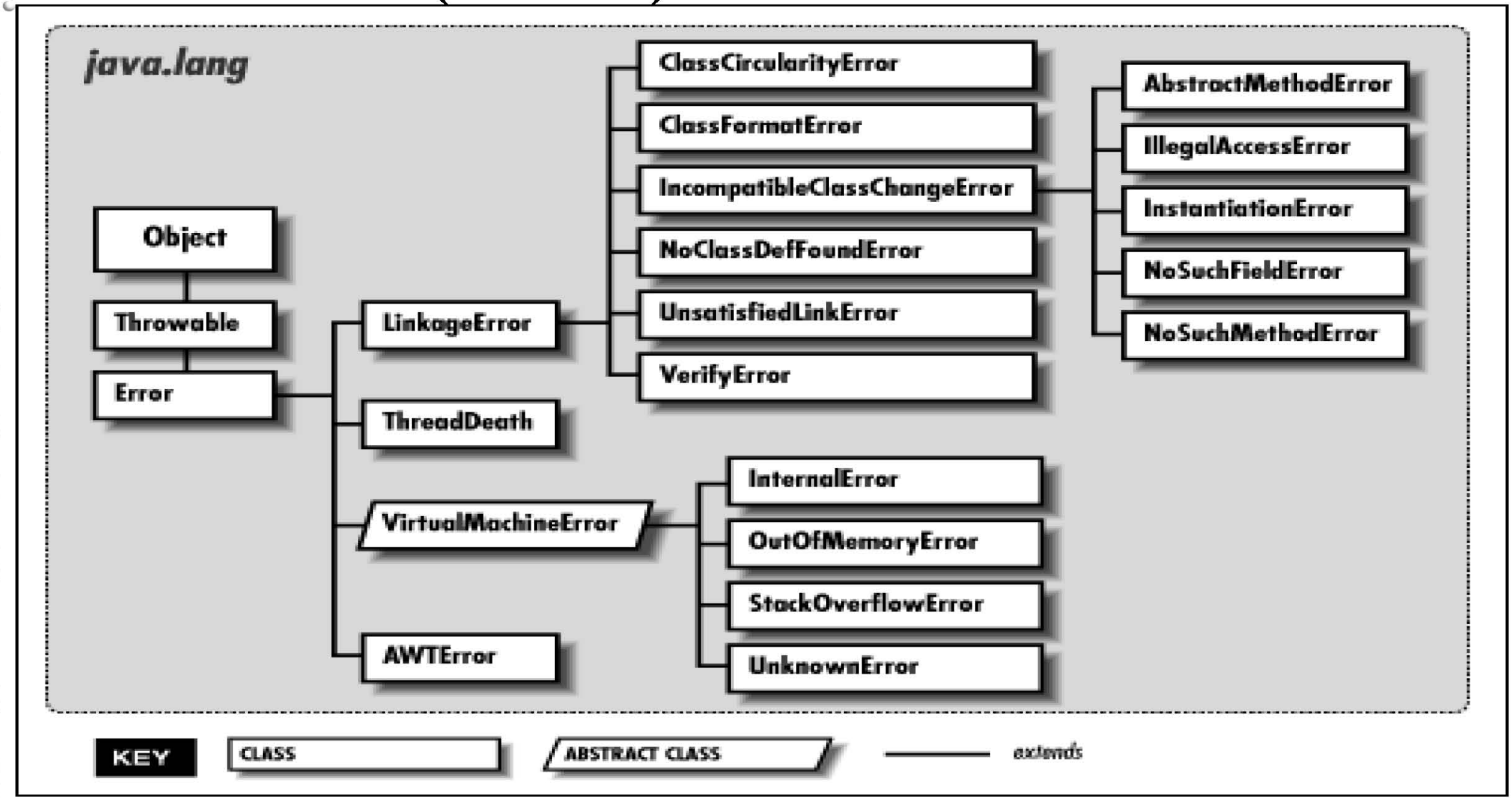
Some of the Built-in Exceptions in Java

Runtime Exceptions or Un-Checked Exceptions defined in java.lang package	
Exception	**Description**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
Checked-Exceptions defined in java.lang package	
Exception	**Description**
ClassNotFoundException	Class not found.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Exception classes (inbuilt) in Java



Error classes (inbuilt) in Java



Creating Custom (User-Defined) exceptions

- ❖•User defined exceptions in java are also known as Custom exceptions.
- ❖•Most of the times when we are developing an application in java, we often feel a need to create and throw our own exceptions.
- ❖•These exceptions are known as User defined or Custom exceptions.
- ❖•We can create our own exception sub class simply by extending Exception class or Throwable class.
- ❖•We can define a constructor for our Exception sub class (not compulsory).
- ❖•And also we can override the toString() function to display our customized message when caught using catch.

Examples of Customised (User-Defined) exceptions

- Customized checked exception example
- Making a Customized checked Exception to an un-checked exception example
- Customized un-checked exception example

End of UNIT-II