UNIVERSITY OF MISKOLC
FACULTY OF MECHANICAL ENGINEERING
AND INFORMATICS

# Java programming

GEIAL31A-B2a

---

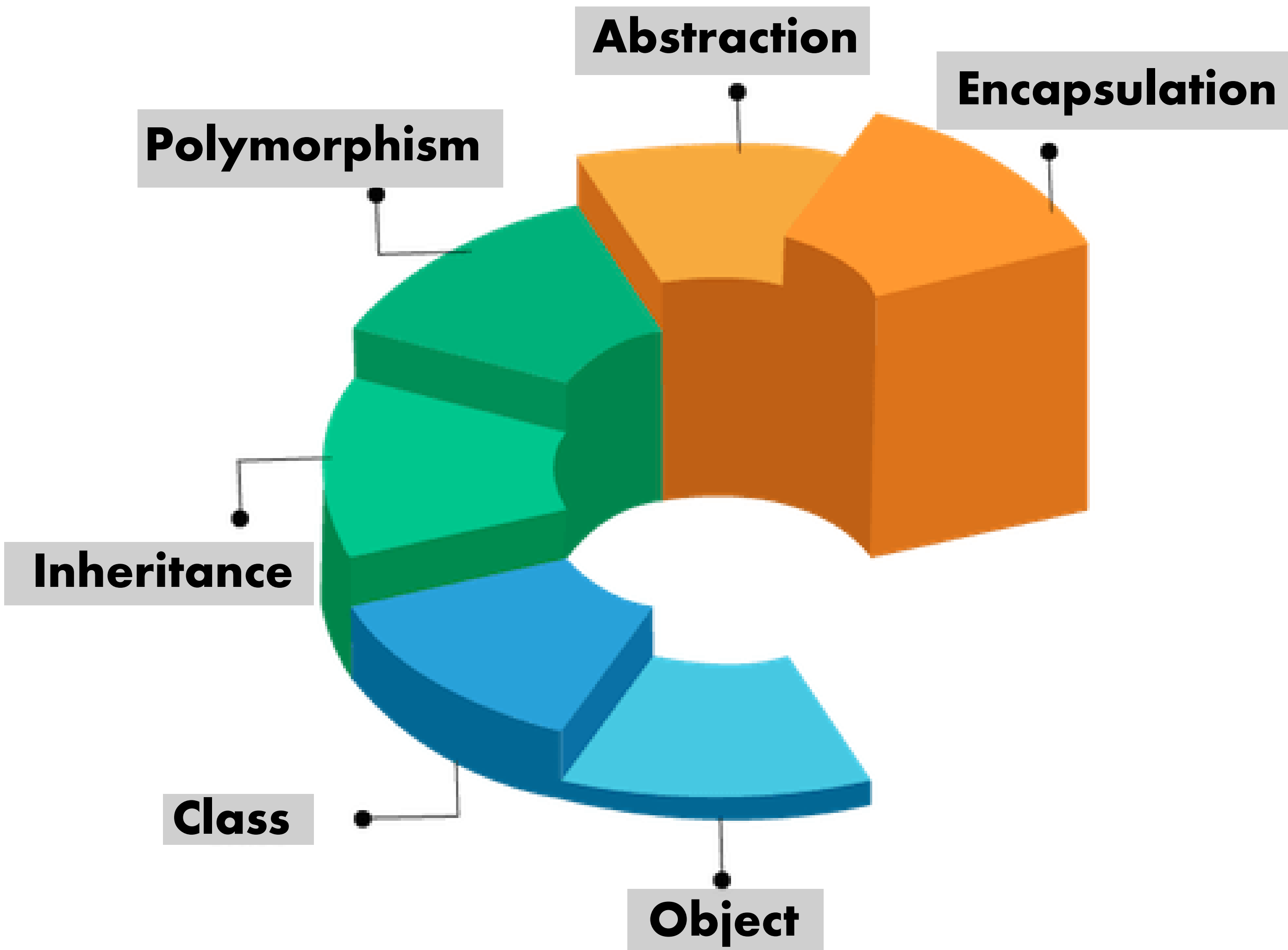
## Java language features II.
## Principles, Inheritance, Binding, Exception

**Tamás Tompa, PhD**
assistant professor
Department of Information Technology

*Based on materials prepared by Miklós Szűcs*

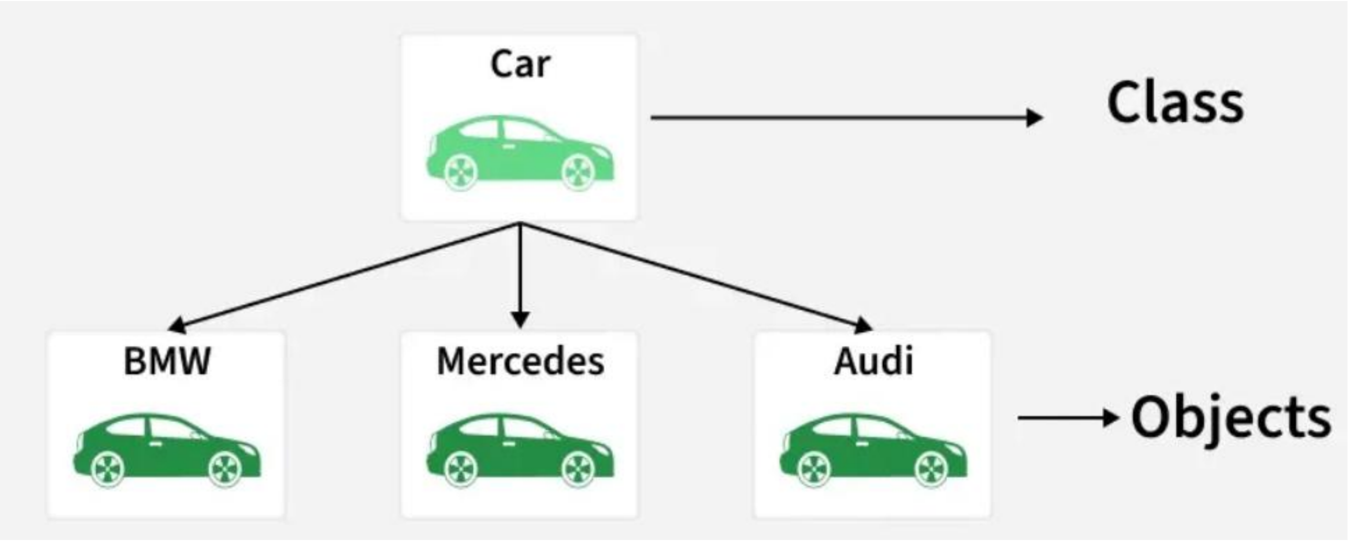
Miskolc, 2026

# OOP principles



# OOP principles

- **Class**
  - a Class is a **user-defined blueprint** or prototype from which objects are created
  - it represents the **set of properties** or methods that are common to all objects of one type
  - using classes, can be create multiple objects with the same behavior instead of writing their code multiple times



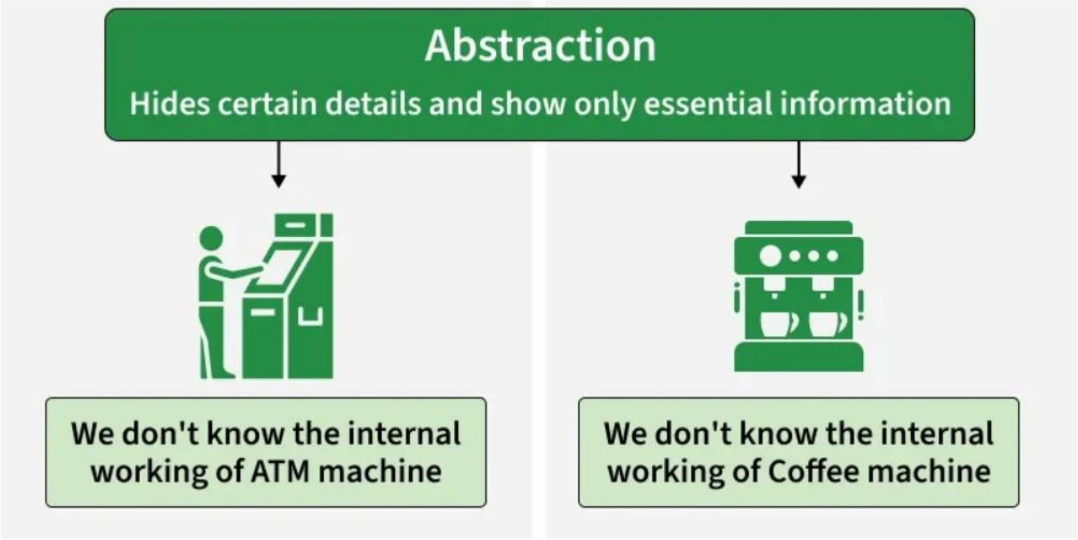
# OOP principles

- **Object**
  - an Object is a **basic unit of Object-Oriented Programming** that represents real-life entities
  - a typical **Java program creates many objects**, which interact by invoking methods
  - An object mainly consists of:
    - **State**: it is represented **by the attributes of an object**. It also reflects the properties of an object
    - **Behavior**: it is represented **by the methods of an object**. It also reflects the response of an object to other objects
    - **Identity**: it is a **unique name given to an object** that enables it to interact with other objects
    - **Method**: a method is a **collection of statements** that perform some specific task and return the result to the caller

# OOP principles

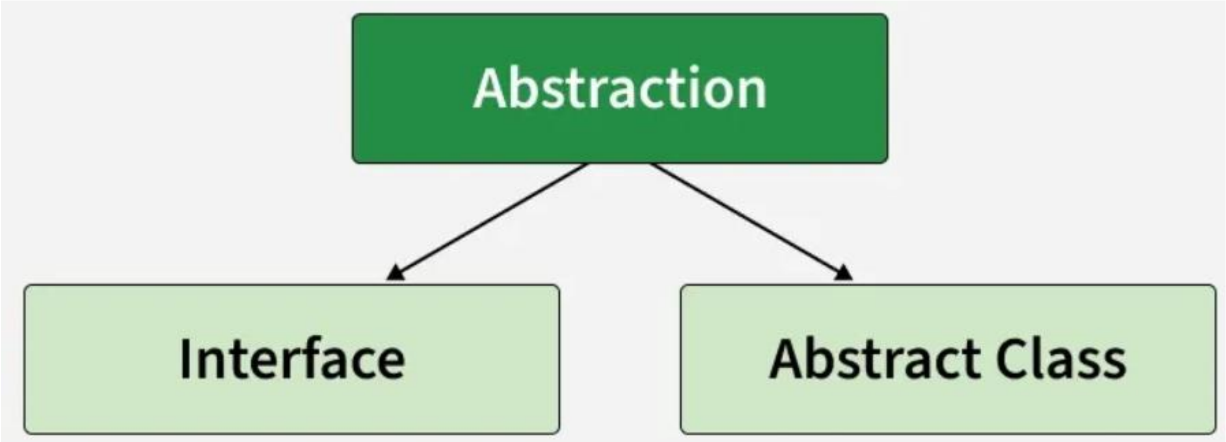
## Abstraction

- abstraction in Java is the process of **hiding implementation details** and showing only the essential features of an object
- hides complexity: internal implementation details are hidden from the user
- improves maintainability: changes in implementation do not affect the user code
- enhances flexibility: supports loose coupling through abstract classes and interfaces



# OOP principles

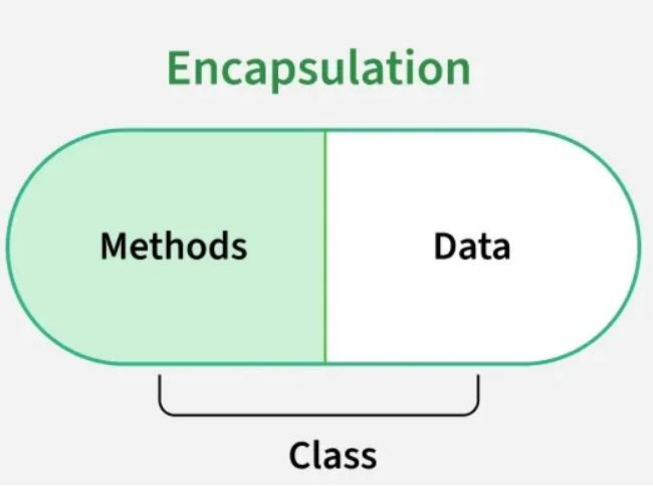
- **Abstraction**
  - it is achieved in Java using abstract classes and interfaces



# OOP principles

## Encapsulation

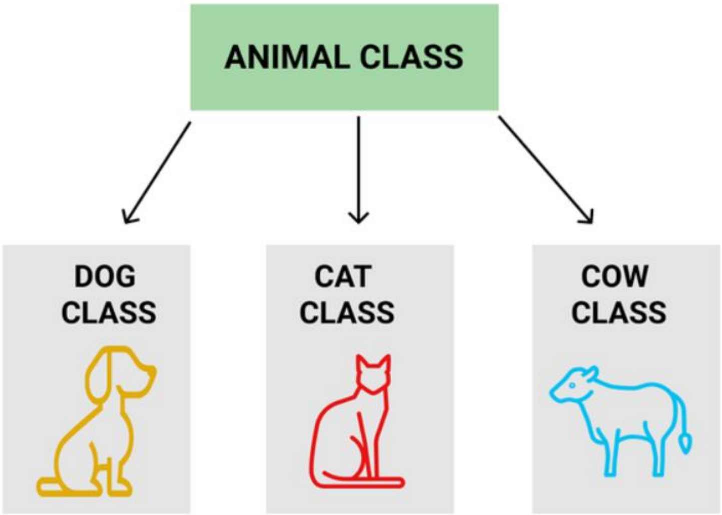
- encapsulation is **the process of wrapping data and methods into a single unit**, usually a class, and restricting direct access to the data
- it acts as a protective shield that prevents data from being accessed directly from outside the class
- **data members are hidden** using the private access modifier
- **access to data is provided through public getter and setter methods**
- it improves data security, maintainability, and controlled access

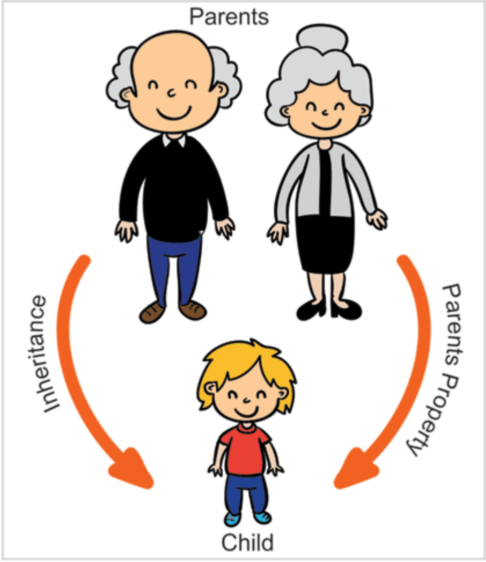


# OOP principles

## Inheritance

- inheritance is a core OOP concept in Java that allows **one class to acquire the fields and methods of another class using the extends keyword**
- it represents an **“is-a” relationship between classes**
- the class being inherited is called the **superclass**, and the inheriting class is the **subclass**
- a subclass can use existing features of the superclass and also add its own
- inheritance promotes **code reusability** and reduces redundancy

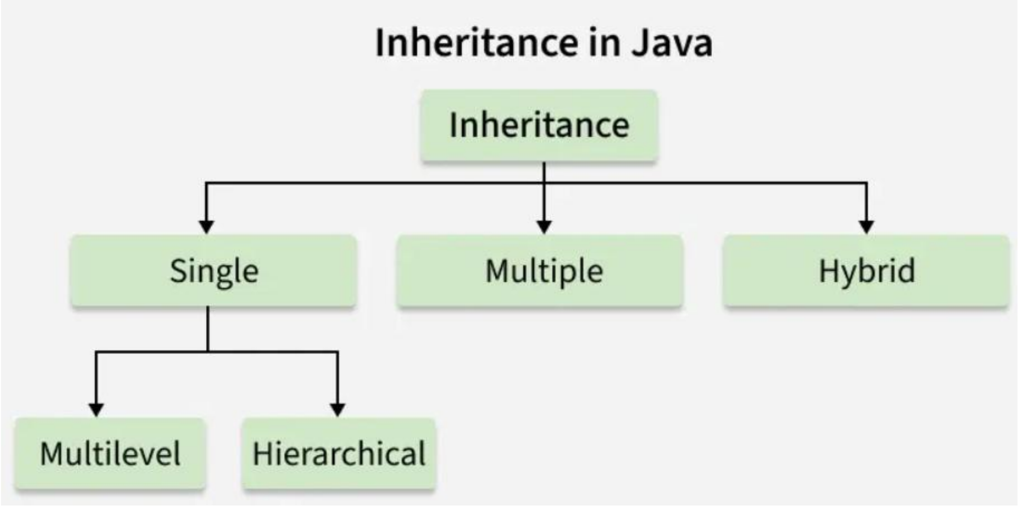


# OOP principles

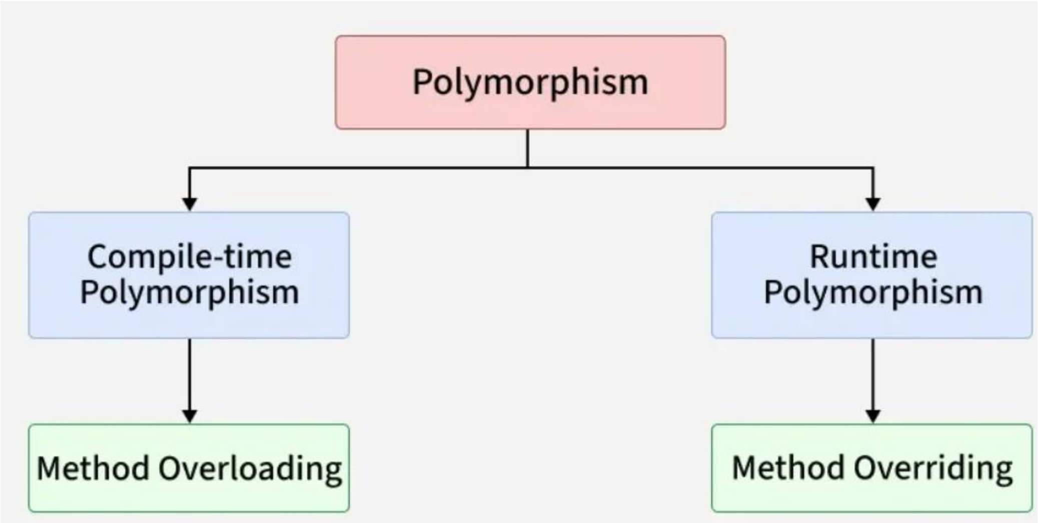
## Inheritance

- **Single Inheritance**: one subclass inherits from one superclass
- **Multilevel Inheritance**: a class is derived from another derived class, forming a chain
- **Hierarchical Inheritance**: multiple subclasses inherit from a single superclass
- **Multiple Inheritance (through Interface)**: a class inherits from multiple interfaces since Java does not support multiple inheritance using classes
- **Hybrid Inheritance (through Interface)**: a combination of two or more types of inheritance, achievable using interfaces



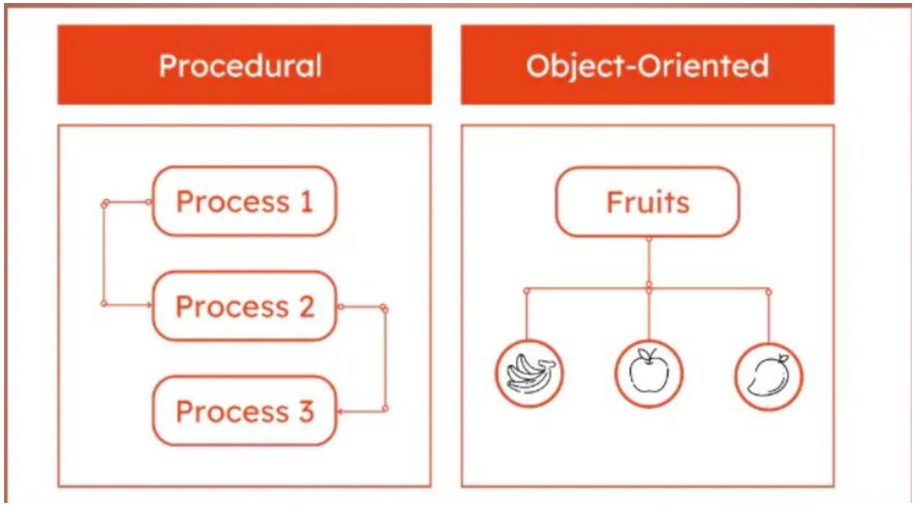
# OOP principles

- **Polymorphism**
  - it allows the **same method or object to behave differently** based on the context, specially on the project's actual runtime class
    - **1. Compile-time Polymorphism (Method Overloading)**: achieved when multiple methods have the same name but different parameters. The method call is resolved at compile time.
    - **2. Runtime Polymorphism (Method Overriding )**: achieved when a subclass provides a specific implementation of a method already defined in its superclass. The method call is resolved at runtime based on the object



# OOP principles

- **Advantage of OOP over Procedure-Oriented Programming Language**
  - **Code reusability**: classes and objects allow reuse of existing code, reducing duplication and improving efficiency
  - **Better structure and maintainability**: programs are organized into logical units, making code easier to understand, debug, and maintain
  - **Supports DRY principle**: common functionality is written once and reused, leading to cleaner and more maintainable code
  - **Faster development**: modular and reusable components help in quicker and scalable application development



# Inheritance

- **Syntax**
  - core OOP concept that allows **a class to acquire properties and behaviors from another class**
  - inheritance is implemented using the **extends keyword**

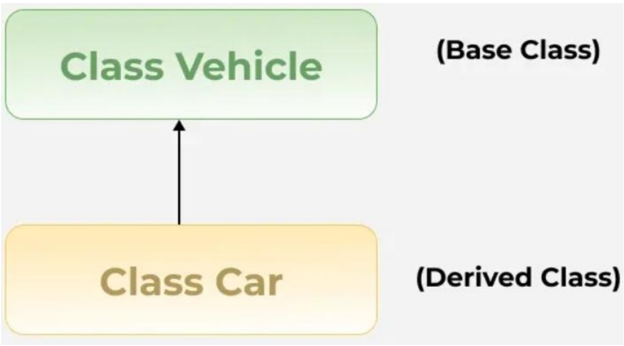
```
class Parent {
   // fields and methods
}

class Child extends Parent {
   // additional fields and methods
}
```

| Term | Description |
|---|---|
| Class | blueprint from which objects are created |
| Superclass | class whose properties are inherited |
| Subclass | class that inherits another class |
| extends | keyword used to inherit a class |

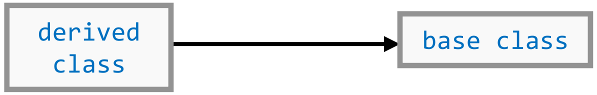
# Inheritance

- **Single inheritance**
  - **a sub-class is derived from only one super class**
  - it inherits the properties and behavior of a single-parent class

```java
//Super class
class Vehicle {
    Vehicle() {
        System.out.println("This is a Vehicle");
    }
}

// Subclass
class Car extends Vehicle {
    Car() {
        System.out.println("This Vehicle is Car");
    }
}
```

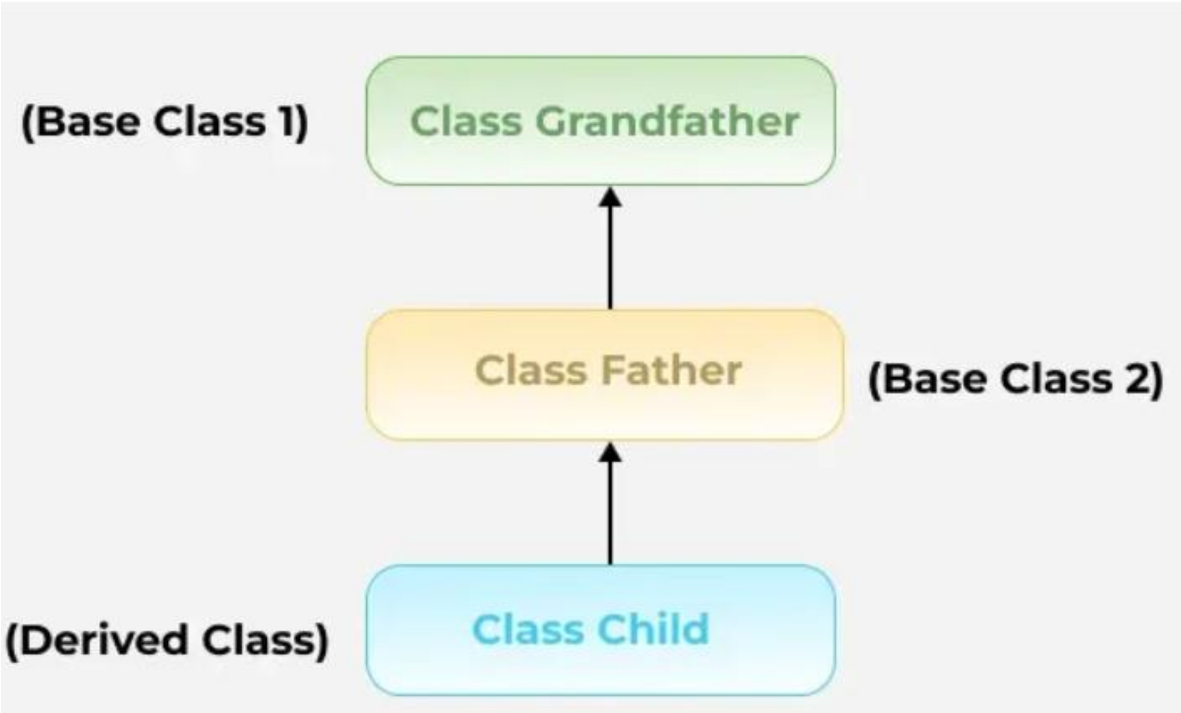



Inheritance denoted by:



# Inheritance

- **Multilevel inheritance**
  - a derived class will be inheriting a base class and as well as **the derived class also acts as the base class for other classes**



# Inheritance

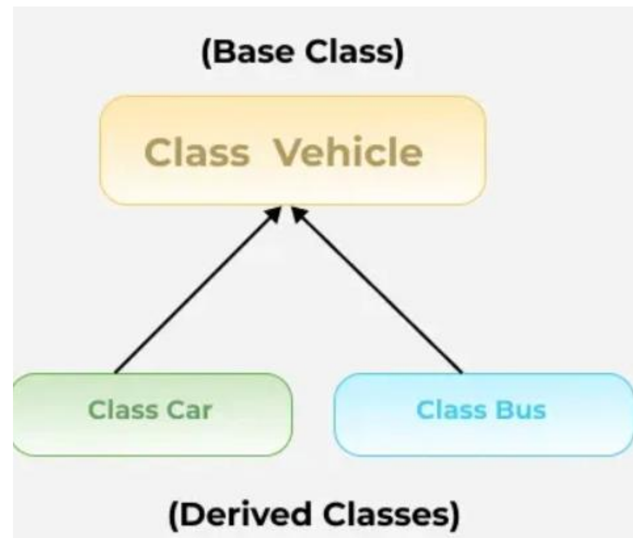
- **Multilevel inheritance**
  - a derived class will be inheriting a base class and as well as **the derived class also acts as the base class for other classes**

```
class Vehicle {
    Vehicle() {
        System.out.println("This is a Vehicle");
    }
}

class FourWheeler extends Vehicle {
    FourWheeler() {
        System.out.println("4 Wheeler Vehicles");
    }
}

class Car extends FourWheeler {
    Car() {
        System.out.println("This 4 Wheeler Vehicle is a Car");
    }
}
```

# Inheritance

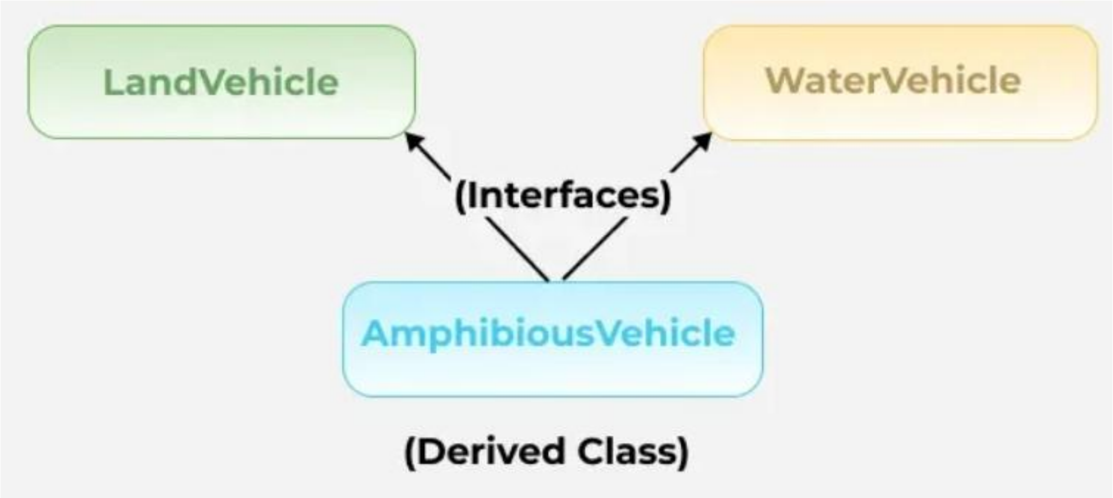
- **Hierarchical inheritance**
  - **more than one subclass is inherited from a single base class.** i.e. more than one derived class is created from a single base class

```
class Vehicle {
    Vehicle() {
        System.out.println("This is a Vehicle");
    }
}

class Car extends Vehicle {
    Car() {
        System.out.println("This Vehicle is Car");
    }
}

class Bus extends Vehicle {
    Bus() {
        System.out.println("This Vehicle is Bus");
    }
}
```



# Inheritance

- **Multiple inheritance**
  - **one class can have more than one superclass** and inherit features from all parent classes
  - **Java does not support multiple inheritances with classes**
  - In Java, we can achieve multiple inheritances **only through Interfaces**



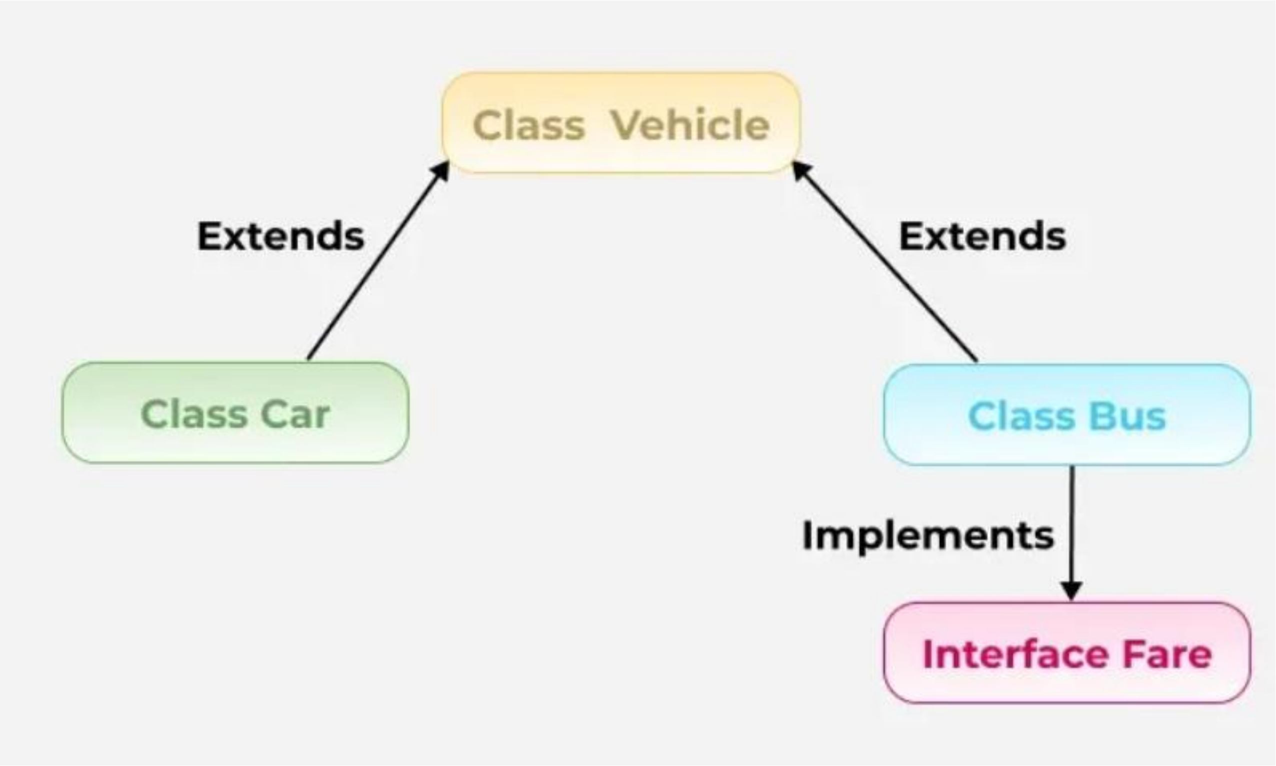
# Inheritance

- **Multiple inheritance**
  - **one class can have more than one superclass** and inherit features from all parent classes
  - **Java does not support multiple inheritances with classes**. In Java, we can achieve multiple inheritances only through Interfaces

```
interface LandVehicle {
    default void landInfo() {
        System.out.println("This is a LandVehicle");
    }
}

interface WaterVehicle {
    default void waterInfo() {
        System.out.println("This is a WaterVehicle");
    }
}

class AmphibiousVehicle implements LandVehicle, WaterVehicle {
    AmphibiousVehicle() {
        System.out.println("This is an AmphibiousVehicle");
    }
}
```

# Inheritance

- **Hybrid inheritance**
  - it is a mix of two or more of the above types of inheritance
  - In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance



# Inheritance

- **Hybrid inheritance**

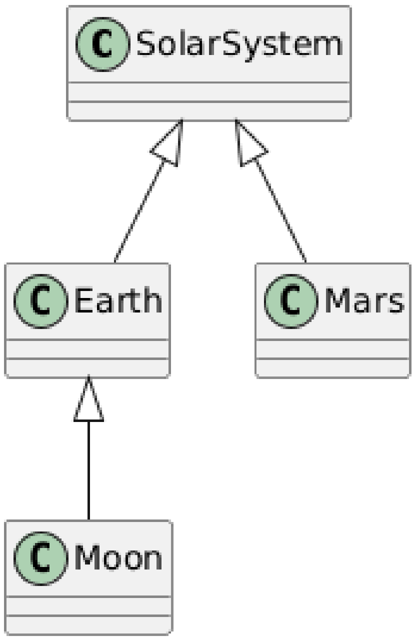
```
class Vehicle {
    void vehicleType() {
        System.out.println("This is a Vehicle");
    }
}

interface Fare {
    default void fareInfo() {
        System.out.println("Fare information");
    }
}
```

```
class Car extends Vehicle {
    void carType() {
        System.out.println("This is a Car");
    }
}

class Bus extends Vehicle implements Fare {
    void busType() {
        System.out.println("This is a Bus");
    }
}
```

# Inheritance

- **IS-A type of Relationship**
  - IS-A represents an inheritance relationship in Java, meaning this object is a type of that object

```
public class SolarSystem {
}

public class Earth extends SolarSystem {
}

public class Mars extends SolarSystem {
}

public class Moon extends Earth {
}
```



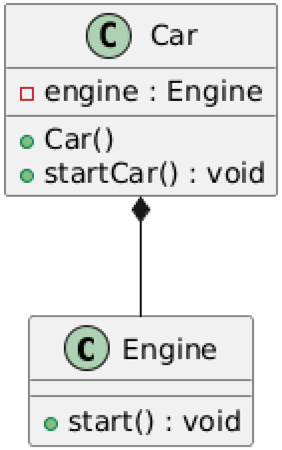

  - SolarSystem is the superclass of Earth class
  - SolarSystem is the superclass of Mars class
  - Earth and Mars are subclasses of SolarSystem class
  - Moon is the subclass of both Earth and SolarSystem classes

# Inheritance

- **HAS-A type of Relationship**
  - a has-a relationship means that **one class contains or uses another class**
  - it represents **composition or aggregation, not inheritance**
  - the relationship is implemented by having an object of one class as a field of another class
  - **one object has another object as a part of it**

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine; // has-a relationship

    public Car() {
        engine = new Engine();
    }

    void startCar() {
        engine.start();
    }
}
```

- Car has an Engine
- Car does not extend Engine
- Car uses Engine to perform its behavior



# HAS-A type of relationship

- Two types
  - **Composition**
    - **strong** "has-a" **relationship**
    - the **contained object's lifecycle depends on the container** - if the container is destroyed, the contained object is destroyed as well
    - parts are created and owned by the whole; lifecycle depends on the whole
    - in UML, composition is shown with a filled diamond (◆)
    - example: a house has rooms; without the house, the rooms do not exist
  - **Aggregation**
    - **weak** "has-a" **relationship**
    - the **contained object can exist independently of the container object** and may be shared among multiple objects
    - parts are injected, can exist independently and may be shared
    - in UML, aggregation is shown with a hollow diamond (◊)
    - example: a Computer has a CPU, but the CPU can exist independently or be replaced

# HAS-A type of relationship

- **composition code snippet**

```
// Composition: Computer *owns* its parts (created inside, no external sharing required)

public class Computer {
    private final CPU cpu;
    private final Memory memory;

    // Whole creates its parts (strong lifecycle dependency)
    public Computer(int cpuClock, int memoryCapacity) {
        this.cpu = new CPU(cpuClock);
        this.memory = new Memory(memoryCapacity);
    }
    …
}
```

```
Computer pc = new Computer(3600, 16384);
```

- CPU and Memory are created inside Computer
- They are final and not externally replaceable → the whole owns the parts
- Conceptually, when Computer is gone, its parts are gone too

# HAS-A type of relationship

- **aggregation code snippet**

compulsory dependecy (constructor)

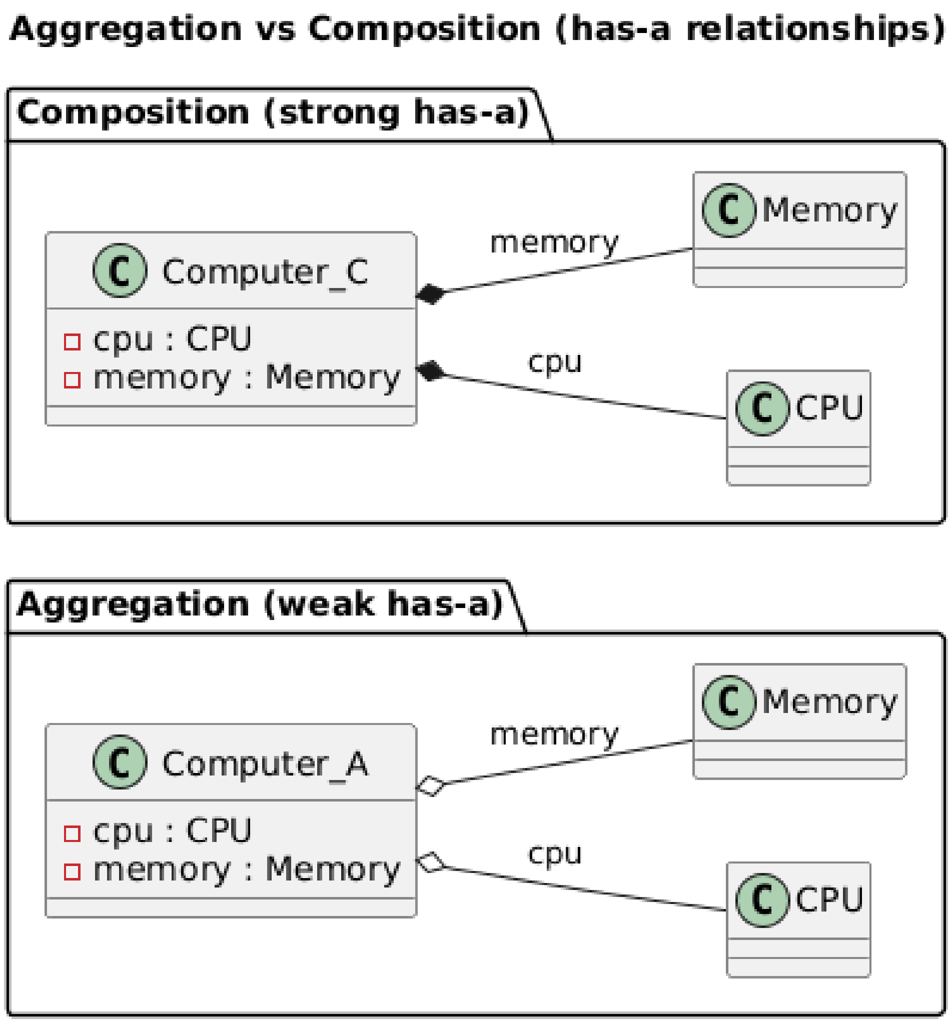
optional dependecy (setter)

```
// Aggregation: Computer *uses* CPU and Memory, but does not own their lifecycle

public class Computer {
    private CPU cpu;
    private Memory memory;

    // Parts are injected by the constructor (compulsory)
    public Computer(CPU cpu, Memory memory) {
        this.cpu = cpu;
        this.memory = memory;
    }

    // Parts are injected by the setter (optional)
    public void setCPUandMemory(CPU cpu, Memory memory) {
        this.cpu = cpu;
        this.memory = memory;
    }
}
```

```
CPU cpu = new CPU(3200);
Memory mem = new Memory(16384);
Computer pc1 = new Computer(cpu, mem);

Computer pc2 = new Computer();
pc2.setCPUandMemory(cpu, memory);
```

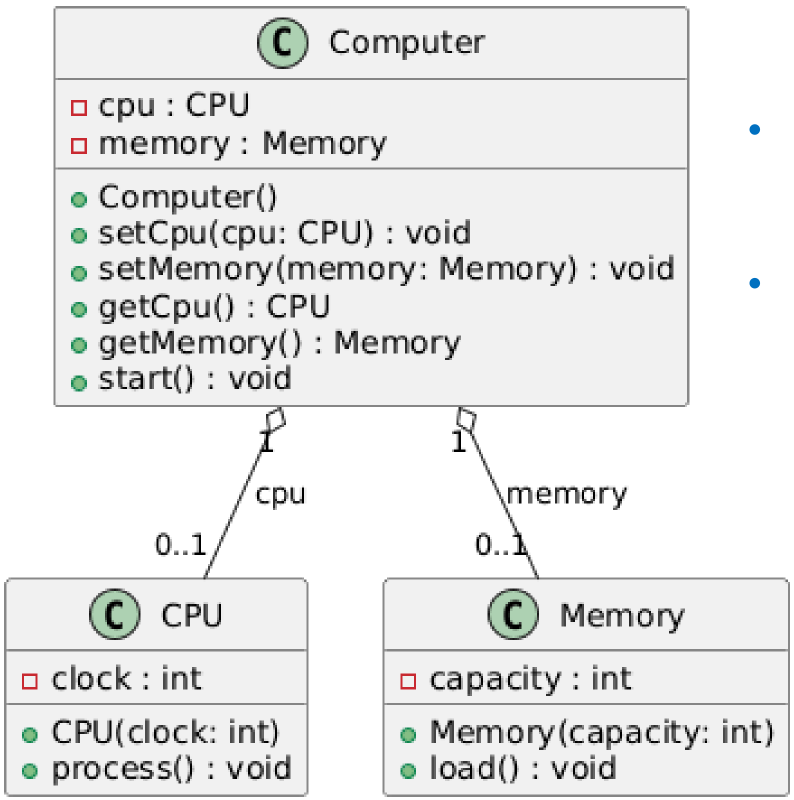
- CPU and Memory are created outside the computer and passed in
- They can be shared or replaced without destroying the Computer

# HAS-A type of relationship



# HAS-A type of relationship

**Aggregation (setter injection): Computer has-a CPU and Memory**

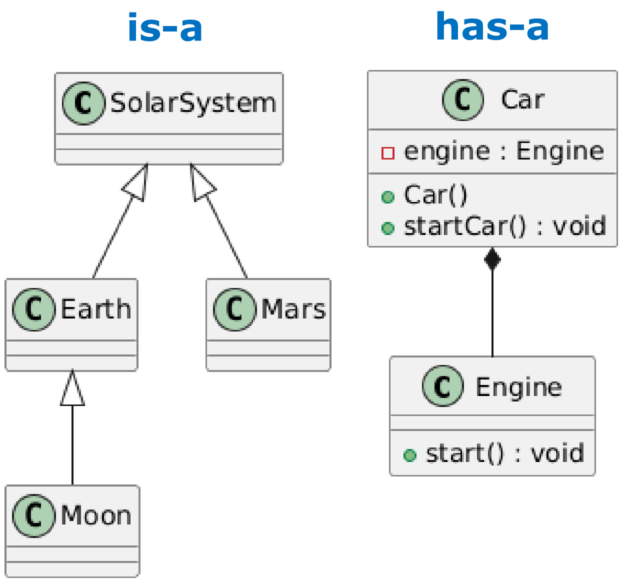



- Aggregation: the CPU and Memory can exist independently and can be replaced (setter injection).
- 0..1 multiplicity: since the dependencies are optional, the Computer can be created without a CPU or Memory, and they can be set later

# When to use is-a vs. has-a in Java

- **is-a relationship (Inheritance)**
  - one class is a specialized type of another
  - implemented using extends
  - the subclass is a kind of the superclass

- **has-a relationship (Composition)**
  - a class contains or uses another class
  - implemented using object fields
  - one object has another object as part of it



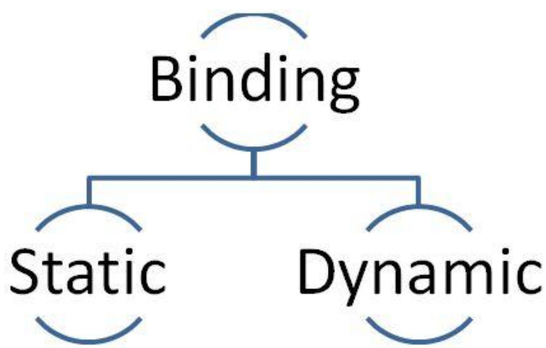

- Use **is-a** for inheritance when **one class is a specialized version of another**, and **has-a** for composition when **a class contains or uses another class**

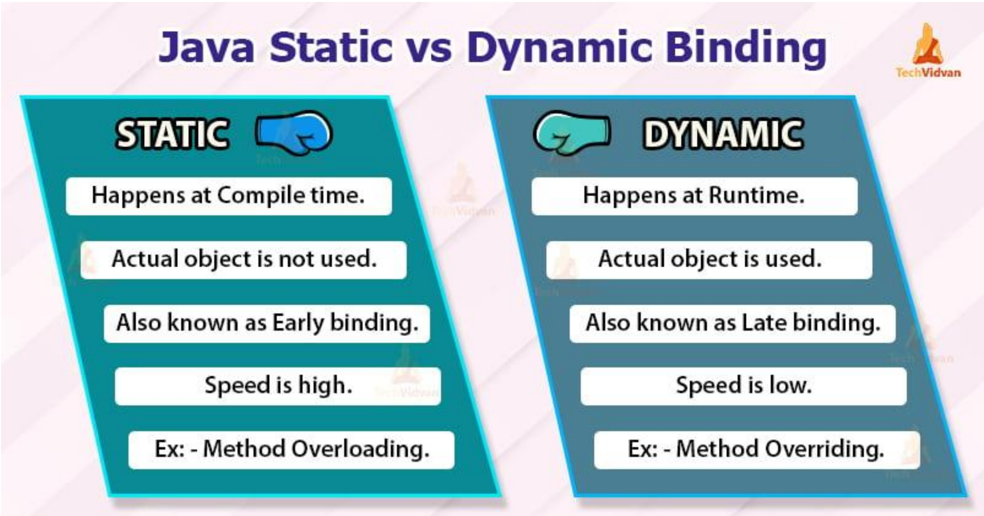
# Inheritance

- **Advantages of Inheritance in Java**
  - **Code Reuse**: subclasses can reuse fields and methods of the superclass, reducing duplicated code
  - **Abstraction**: common behavior can be defined in a superclass, making the code easier to extend and maintain
  - **Class Hierarchy**: inheritance helps model real-world relationships between classes
  - **Polymorphism**: subclasses can override superclass methods to provide different behavior
  - **Polymorphism: i**nheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways
- **Disadvantages of Inheritance in Java**
  - **Increased Complexity**: deep inheritance hierarchies can make the code harder to understand
  - **Tight Coupling**: changes in the superclass may affect all subclasses

# Binding

- Binding is a mechanism **creating link between method call and method actual implementation**
- As per the **polymorphism** concept in Java, object can have many different forms
- Object forms can be resolved at compile time and run time
  - if it’s mapped at **compile time**, it’s a static or **early binding**
  - if it’s resolved at **runtime**, it’s known as dynamic or **late binding**



# Binding

- **The static and the dynamic type of variables**

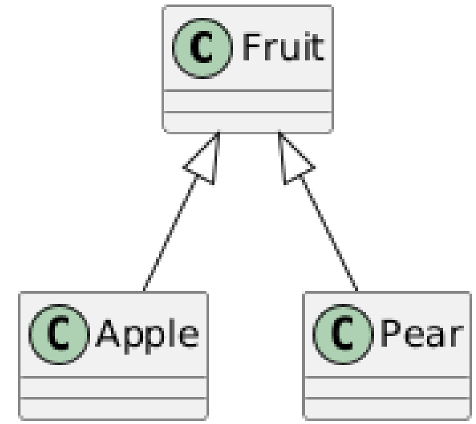
```
Fruit x = new Apple(„Gala");
```

static type

- never be change
- properties:
  - data members
  - methods

dynamic type

- can be change
- it must have all the properties of the static type, but it may also have additional properties
  - therefore, it can be of the **static type or** of any of its **subclasses**

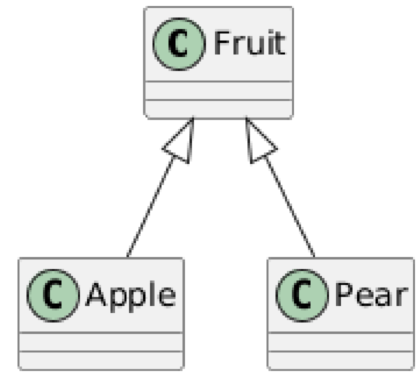



- A Fruit instance can be of type Fruit, Apple, or Pear
- An Apple instance can only be of type Apple
- A Pear instance can only be of type Pear

# Binding

- **The static and the dynamic type of variables**

```
Fruit f = new Pear(„Bosc");
Apple a = new Apple("Jonatan");
System.out.println(f instanceof Apple); false
a = (Apple)f;
```




```
Exception in thread "main" java.lang.ClassCastException:
Pear cannot be cast to Apple
```

- f is a reference of type Fruit that refers to a Pear object
- f instanceof Apple returns false, because the actual object is a Pear, not an Apple
- the cast a=(Apple)f is an explicit downcast
- although the cast is allowed at compile time, it fails at runtime because a Pear object cannot be cast to Apple
- the actual object type (Pear) is not compatible with the target type (Apple)

# Binding

- **Static or early binding**
  - the method call is **resolved at compile time**
  - the method to be executed is **determined based on the static type**, not the actual (dynamic) object type
  - this happens when the compiler knows exactly which method or variable to call based on the code

```
<ClassName> <objectName> = new <ClassName>();
```

  - **Characteristics**
    - **Linking**: the link between a method call and its method implementation is resolved at compile time
    - **Resolve mechanism:** early binding uses the static type of the object to resolve the method call
    - **Example**: method overloading is a typical example of early binding

# Binding

- **Static or early binding**

```
class A {
    static void show() {
        System.out.println("Class A");
    }
}

class B extends A {
    static void show() {
        System.out.println("Class B");
    }
}

A obj = new B();
obj.show();   // calls A.show() → early binding
```

```
Output:
Class A
```

- The method show() is declared as static in both classes
- Static methods are not overridden, they are hidden
- Method calls to static methods are resolved at compile time, based on the reference type, not the actual object type
- Although obj refers to an object of class B, the reference (static) type is A
- Therefore, the call obj.show() is bound at compile time to A.show()

# Binding

- **Dynamic or late binding**
  - it refers to the process in which **linking between method call and method implementation is resolved at run time** (or, a process of calling an overridden method at run time)
  - the method to be executed is **determined based on the actual (dynamic) object type**

```
<ClassName> <objectName> = new <DerivedClassName>();
```

  - **Characteristics**
    - **Linking**: linking between method call and method implementation is resolved at run time
    - **Resolve mechanism**: dynamic binding uses the dynamic type of object to resolve binding
    - **Example**: method overriding is the example of dynamic binding

# Binding

## Dynamic or late binding

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
}


public class Tester {
   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object

      // Dynamic Binding
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
   }
}
```

```
Output:
Animals can move
Dogs can walk and run
```

- Animal defines a method move()
- Dog extends Animal and overrides the move() method with its own implementation
- In main, two objects are created:
  - a is an Animal reference pointing to an Animal object
  - b is an Animal reference pointing to a Dog object
- When move() is called:
  - a.move() executes the method defined in Animal
  - b.move() executes the overridden method in Dog, even though the reference type is Animal

# Exception handling

- Different types of errors can occur while running a program - such as coding mistakes, invalid input, or unexpected situations
- Type of errors:

| Error type | Description |
|---|---|
| Compile-Time Error | Detected by the compiler. Prevents code from running |
| Runtime Error | Occurs while the program is running. Often causes crashes |
| Logical Error | Code runs but gives incorrect results. Hardest to find |

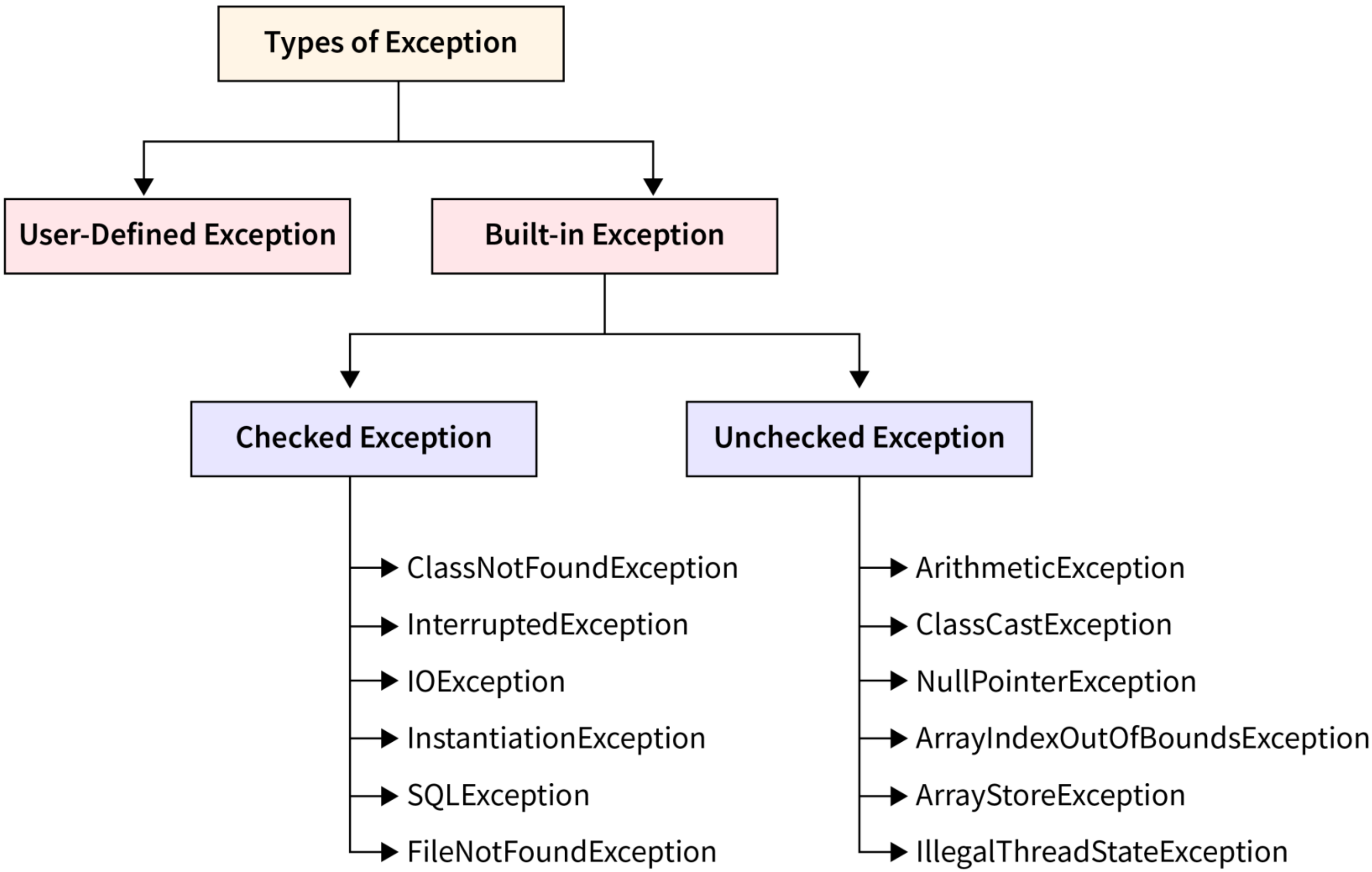
- When an error occurs, Java will normally stop and generate an error message
- The technical term for this is: Java will throw an **exception** (throw an error)

# Exception handling

- **Exceptions**
  - **exceptions are objects in Java**
  - exception classes are subclasses of the Exception class

- Types of exceptions
  - **Checked exceptions**
    - these exceptions must be handled
    - the compiler checks whether they are handled.
    - example: exceptions that occur during I/O operations
  - **Unchecked exceptions**
    - these exceptions do not have to be handled
    - they are subclasses of RuntimeException, which itself is a subclass of Exception
    - examples: null reference errors, array index out of bounds, etc
  - **Custom exception classes**
    - if they extend Exception (directly or indirectly), they are checked exceptions
    - if they extend RuntimeException, they are unchecked exceptions

# Exception handling



# Exception handling

```
try {
          // block of code to try
}
catch(Exception e) {
         // block of code to handle errors
} finally {
         // always executing, regardless of the result
}
```


- The **try** statement defines a block of code to be tested for errors while it is being executed
- The **catch** statement defines a block of code to be executed, if an error occurs in the try block
- The **finally** statement defines a block of code to be always executed, regardless of the result

# Exception handling

- `ArrayIndexOutOfBoundsException` occurs when you try to access an index number that does not exist

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:4)
```

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

```
Something went wrong.
```

# Exception handling

- **Throw keyword**
  - to create a custom error
  - it is used together with an exception type
  - there are many exception types available in Java: `ArithmeticException`, `FileNotFoundException`, `ArrayIndexOutOfBoundsException`, `SecurityException`, etc

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: Access denied -
You must be at least 18 years old.
        at Main.checkAge(Main.java:4)
        at Main.main(Main.java:12)
```

# Exception handling

- **Multiple exceptions**
  - **a try block can be followed by multiple catch blocks**
  - the compiler checks the catch blocks from top to bottom
  - the first catch block that matches the thrown exception type is executed
  - once a matching catch block is found, the remaining catch blocks are skipped
  - more specific exception types must be listed before more general ones

```java
public class Main {
  public static void main(String[] args) {
    try {
      int[] numbers = {1, 2, 3};
      System.out.println(numbers[10]);  // ArrayIndexOutOfBoundsException
      int result = 10 / 0;              // ArithmeticException
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index does not exist.");
    }
    catch (ArithmeticException e) {
      System.out.println("Cannot divide by zero.");
    }
    catch (Exception e) {
      System.out.println("Something else went wrong.");
    }
  }
}
```

```
Output:
Array index does not
exist.
```

# Exception handling

- **Multiple exceptions**

```
try {
    int[] arr = new int[5];
    arr[10] = 3;
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index error");
}
catch (Exception e) {
    System.out.println("General exception");
}
```

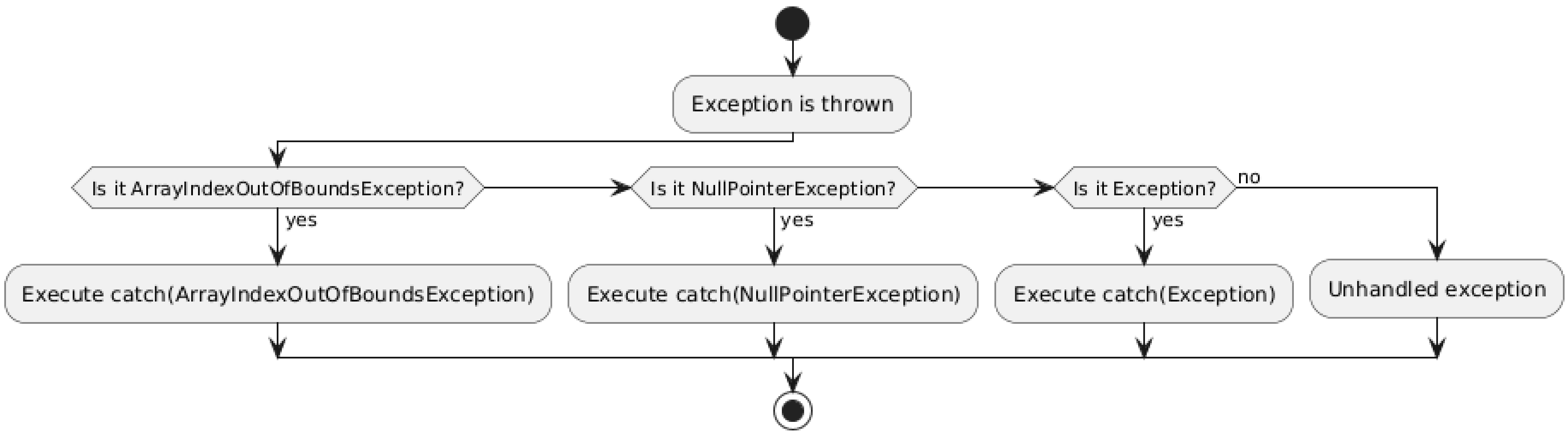
- An ArrayIndexOutOfBoundsException is thrown
- The first catch block matches, so it is executed
- The second catch block is ignored

- **Java 7+ Multi-Catch (Optional)**

```
catch (IOException | SQLException e) {
    System.out.println("I/O or database error");
}
```

# Exception handling

- **Exception matching order**




- Java checks catch blocks from top to bottom
- The first matching catch block is executed
- More specific exceptions must come before more general ones
- If a general exception (e.g. Exception) appears first, the compiler reports an unreachable code error

# Thank you for your attention!