UNIVERSITY OF MISKOLC
FACULTY OF MECHANICAL ENGINEERING
AND INFORMATICS

Java programming

GEIAL31A-B2a

Java classes

Tamás Tompa, PhD
assistant professor
Department of Information Technology

Based on materials prepared by Miklós Szűcs

Miskolc, 2026

Class concept

- **A Java program is a set of classes**
- A class defines a new type
- A class definition is also a complete unit of translation (but usually not a complete program)
- **The name of the class and the file containing its class definition must the same**
- Classes can be arranged into packages to form modules

Class concept

- **The class consists of a header and a body**

```
modifiers class class_name {
   member definitions
}
```

  - Header: modifiers class name
    - modifiers are responsible for controlling the visibility (availability) of departments
      - ex: public (rest shortly)
      - public: the class is visible, can be used in any other class
    - class: keyword that defines a class
    - class name: any unique identifier
  - Body: { }, contains definitions of members

Class concept

- **The class consists of a header and a body**

```
modifiers class class_name
{
   member definitions
}
```

- Members of this class can be:
  - member variables (fields)
  - constructor
  - initialization block
  - method
  - tag class (interface)

Class concept

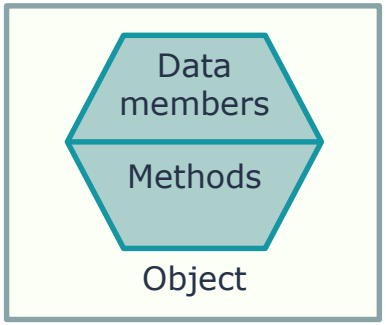
- **A Circle class can be defined as:**

```
public class Circle {
   double kp_x;
   double kp_y;
   double radius;

   void move(double dx, double dy) {
      kp_x += dx;
      kp_y += dy;
   }
   void resize(double newradius) {
      radius = newradius;
   }
}
```

Properties: fields / member variables

Behavior: Methods



Encapsulatoin:
A class treats its data and methods as a single unit and isolates them from the outside world

Class concept

```
public class CircleRun {
   public static void main(String[ ] p){
      Circle a, b;
      a = new Circle();
      a.setX(0);
      a.setY(0);
      a.setRadius(10);

      b = new Circle();
      b.setX(5);
      b.setY(8);
      b.setRadius(3);
      b.shit(5, 5);
   }
}
```

- main method
- declare
- initialize by setter methods
- reference to data member
- reference to method

Class concept

- Implementation of the principle of encapsulatoin: **each class has its own competence / function**
- **Objects communicate with each other by sending messages**
- The set of methods through which an object can be invoked is called the object's interface
- For objects, **can be controlled which member variables are visible** and which methods and constructors can be invoked by other objects

Construtor

- A constructor in Java is a **special method** that is used to initialize objects
- The constructor is **called when an object of a class is created**
- It can be used to set initial values for object attributes
- A constructor has the **same name as the class**
- It does **not have a return type**, not even void
- It can accept **parameters to initialize object properties**

```
modifier classname (parameters) {
      instructions
}
```

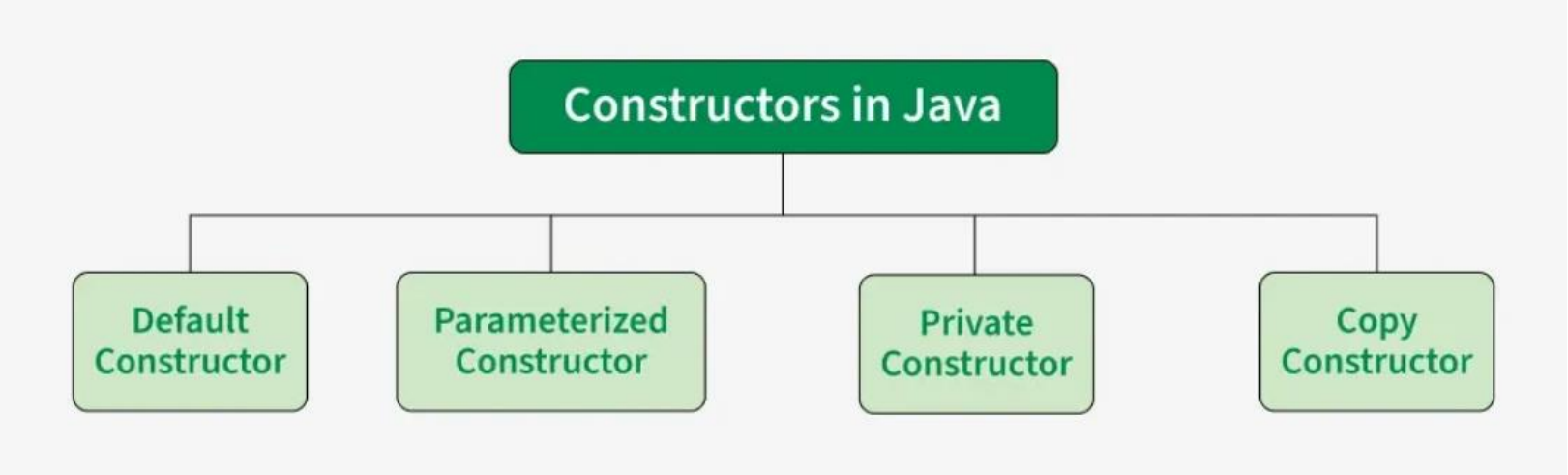
Construtor

```
public class Circle {
    private double x;
    private double y;
    private double radius;

    public void Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.radius = r;
    }
}
```

Construtor

- **Type of constructors:**
  - Default Constructor
  - Parameterized Constructor
  - Private Constructor
  - Copy Constructor



Construtor

- **Default Constructor**
  - a default constructor has no parameters
  - it is used to assign default values to an object

```
class Geeks{
    Geeks() {
        System.out.println("Default constructor");
    }
}
```

Construtor

- **Parameterized Constructor**
  - a constructor that has parameters is known as parameterized constructor
  - if we want to initialize fields of the class with our own values

```
class Geeks {
    private String name;
    private int id;

    Geeks(String name, int id) {
        this.name = name;
        this.id = id;
    }
}
```

Construtor

- **Private Constructor**
  - a private constructor cannot be accessed from outside the class
  - it is commonly used in:
    - Singleton Pattern: to ensure only one instance of a class is created
    - Utility/Helper Classes: to prevent instantiation of a class containing only static methods

```
class GFG {
     private GFG() {
          System.out.println("Private constructor called");
    }
}
```

Construtor

- **Copy Constructor**
  - unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object

```
class Geeks {
    private String name;
    private int id;

    Geeks(String name, int id) {
        this.name = name;
        this.id = id;
    }

    Geeks(Geeks obj2) {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
```

Instantiation of objects

```
className name = new className([parameters])
```

- The parentheses () are mandatory, even if the constructor has no parameters.

- The process
  - the **memory** required for the object is **allocated**
  - the **object** is **initialized** (the programmer may influence the initialization)
  - the **constructor** is **executed**
  - a **reference** to the allocated memory area is **stored in the variable**

Initialization of objects

- The **member variables / fields initialize automatically** when an object created:
  - **numeric** types with **0**,
  - **char** types with **0 code** characters,
  - **boolean** types with **false**,
  - **references** with referencia **null** values

- In 99% of cases, a different value would be required, so we have to set the values ourselves
- This can be solved with a method that will be invoked during instantiation
  - **constructor**

Objects lifetime

- **An object is created when a class is instantiated**
- It is destroyed ? (we do not know exactly when)

- **Automatic garbage collector**
  - Java uses an automatic garbage collector, which keeps track of how many references point to each object
  - An object that has no references pointing to it becomes eligible for garbage collection and may be removed (the exact time when this happens is not deterministic)
  - A reference to an object is lost when:
    - the variable itself ceases to exist
    - the value of the variable is changed (it is assigned to another object)
    - the variable is explicitly assigned the value null

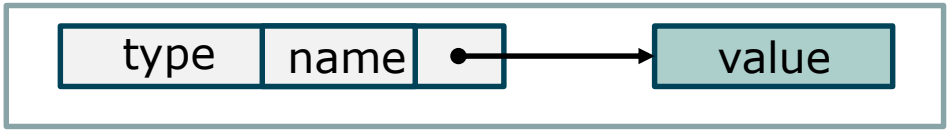
Instantiation example

- **The „Circle" is a type** `Public class Circle…`
- The variables with this type can be declared `Circle a, b;`
- **Instances** can be created `new Circle();`
- The **result is a reference** of the appropriate type (in this case: Circle) that points to the created object
- This reference can be stored in reference variables:

```
Circle a,b;
a = new Circle();
b = new Circle();
```

Reference variable

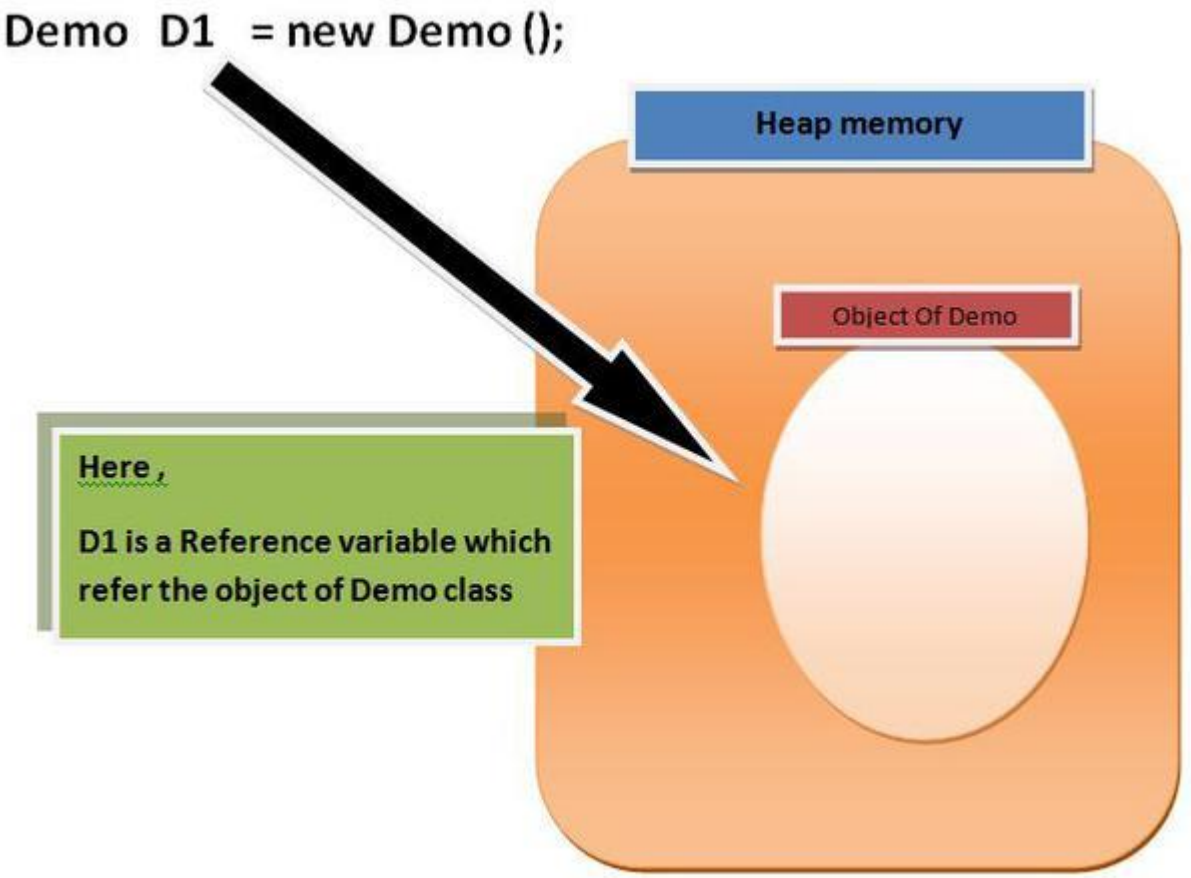
- When an object (instance) created then space is reserved in heap memory
- The space in the heap Memory is created but the question is **how to access that space?**
- Pointing element or simply called **Reference variable** can be created which simply points out the Object (the created space in a Heap Memory)



Reference variable

- Reference variable is **used to point object/values**
- Classes, interfaces, arrays, enumerations, and, annotations are reference types in Java
  - reference variables hold the objects/values of reference types in Java
- Reference variable **can also store null value**
  - by default, **if no object is passed to a reference variable** then it will store a null value

Reference variable



Reference variable

- What can be do with reference variables?
  - Declare:
```
Circle a, b;
```
  - Give values:
```
a = new Circle();
```
  - Reference transfer:
```
b = a;
```
    - `b` points the same object as `a`
  - Examine equality between two references (==, !=):
```
boolean isSame = (a == b);
```
  - Can be referenced to its members:
```
a.x = 2;
a.shift(3, 2);
```

Reference variable

```
public class AutoRun {
    public static void main(String[ ] args) {
        Auto a;
        System.out.println(a);
    }
}
```

```
C:\Java_5>javac -d . AutoP.java
AutoP.java:5: variable a might not have been initialized
       System.out.println(a);
                          ^
1 error
```

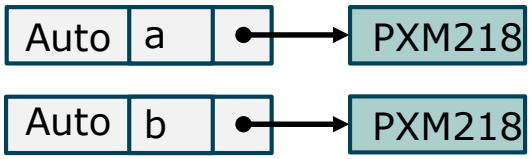
- No operation can be performed on a declared but uninitialized reference, the compiler will report an error!

Reference variable

```
public class AutoRun {
    public static void main(String[ ] args) {
        Auto a = new Auto("PXM218");
        Auto b = new Auto("PXM218");
        System.out.println(a==b);
    }
}
```

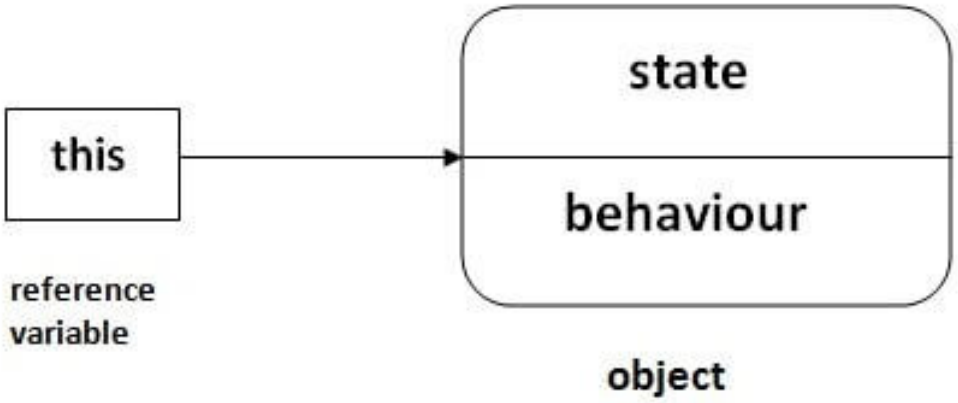
```
C:\Java_5>java AutoP
false
```

- Two references are not the same even if all their data members are the same!



Reference variable

- **this**
  - reference variable that **refers to the current object**



Access modifiers

- A tool for implementing the information hiding principle

Modifier	Description
`private`	The code is only accessible within the declared class
`-`	The code is only accessible in the same package. This is used when you don't specify a modifier
`protected`	The code is accessible in the same package and subclasses
`public`	The code is accessible for all classes

Access modifiers

Modifier	Class	Package	Subclasses	All
private	X	NO	NO	NO
-	X	X	NO	NO
protected	X	X	X	NO
public	X	X	X	X

- **Within the same class**, all objects (members) are accessible, regardless of their access modifier
- **Within the same package**, private members are not accessible
- **From a subclass**, only protected and public members are accessible
- **From all other (external) classes**, only public members are accessible

Access modifiers

- Its application is up to the programmer!
- The data of an object must be inaccessible to the outside world (other objects)
  - from this it follows that **all data members (fields) must have the private access modifier**
- An object may communicate with the outside world only through its interface
  - from this it follows that only **those methods should be public that are used to communicate with the outside world**

Access modifiers

- A **class** should be public or package-private (default) only
- **Data members / fields** and methods may belong to any access category
- In order to follow the principle of information hiding, **every element should be assigned the most restrictive access level possible**

Access modifiers

- **Common conventions**
  - A **class** should be declared **public** only if it is intended for general use
  - **Data members / fields** should be declared **private** (or possibly protected)
    - if access is required, it should be provided through (getter/setter) methods
    - public data members (fields) are considered an error!
  - Among the **methods**, only those that are required by the **outside world** (i.e. those that form the class interface) should be declared **public**

Inner class

- **Nested classes (a class within a class)**
  - purpose of nested classes is to **group classes that belong together**
  - makes the code more readable and maintainable
  - to access the inner class, create an object of the outer class, and then create an object of the inner class
  - **Areas of application**
    - we want **to hide a helper class from the outside world** (private nested class)
    - when implementing a class, we need a helper class that must have access to the private members of the enclosing class
    - we want **to express that a class or an interface is logically subordinate to another one**

Inner class

- **Nested classes (a class within a class)**
  - Syntax

```
class OuterClass
{
...
    class NestedClass
    {
        ...
    }
}
```

Inner class

- **Nested classes (a class within a class)**

```
class OuterClass {
  int x = 10;

  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs: 15 (5 + 10)
```

Inner class

- **Nested classes (a class within a class)**

```
class Outer {
   private int num = 25;
   private class InnerClass {
      public void print() {
         System.out.println("Num="+num);
      }
   }
   void displayInner() {
      InnerClass inner = new InnerClass();
      inner.print();
   }
}
```

```
c:\Java8>java OuterP
Num=25
```

```
public class OuterP {
   public static void main(String args[]) {
      Outer outer = new Outer();
      outer.displayInner();
   }
}
```

Anonymous class

- **A class without name**
  - an anonymous class is **a class without a name that is defined and instantiated at the same time**
  - it is typically **used when a class is needed only once**, usually to override a method or implement an interface in a concise way

  - Key characteristics
    - it has no explicit name
    - it is declared and instantiated in a single expression
    - it can override methods or implement interface methods
    - it cannot have constructors, only instance initializers

Anonymous class

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running in an anonymous class.");
    }
};


r.run();
```

- In this example
  - no separate class is defined
  - the Runnable interface is implemented anonymously
  - the run() method is overridden

Anonymous class

```
jrb1 = new JRadioButton(„Red");
jrb1.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent e ) {
      if (jrb1.isSelected())
         color = „Red";
   }
});
```

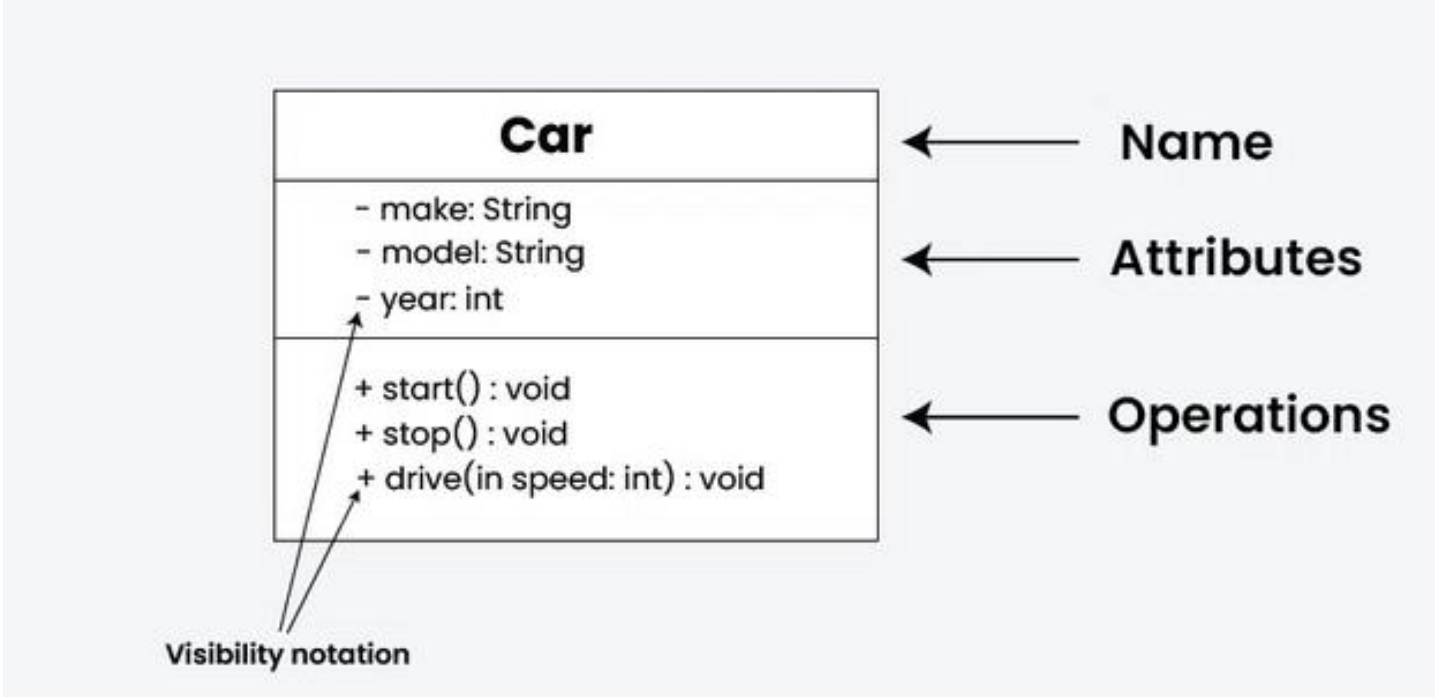
- When the radio button is clicked, the actionPerformed method is automatically called
- Inside this method, the program checks whether the radio button is currently selected
- If it is selected, the variable color is set to „Red”
- An anonymous class is used here to implement the ActionListener interface without defining a separate named class
- This is useful when the listener logic is simple and used only once

UML class diagram

- A UML class diagram **shows the structure of a system by displaying its classes, their attributes, methods, and the relationships between them**
- It helps the team understand **how the system is organized** and how components interact
- Represents **classes, attributes, methods, and relationships**
- Helps communicate and document the software structure
- Makes it easier for developers and designers to understand the system

UML class diagram

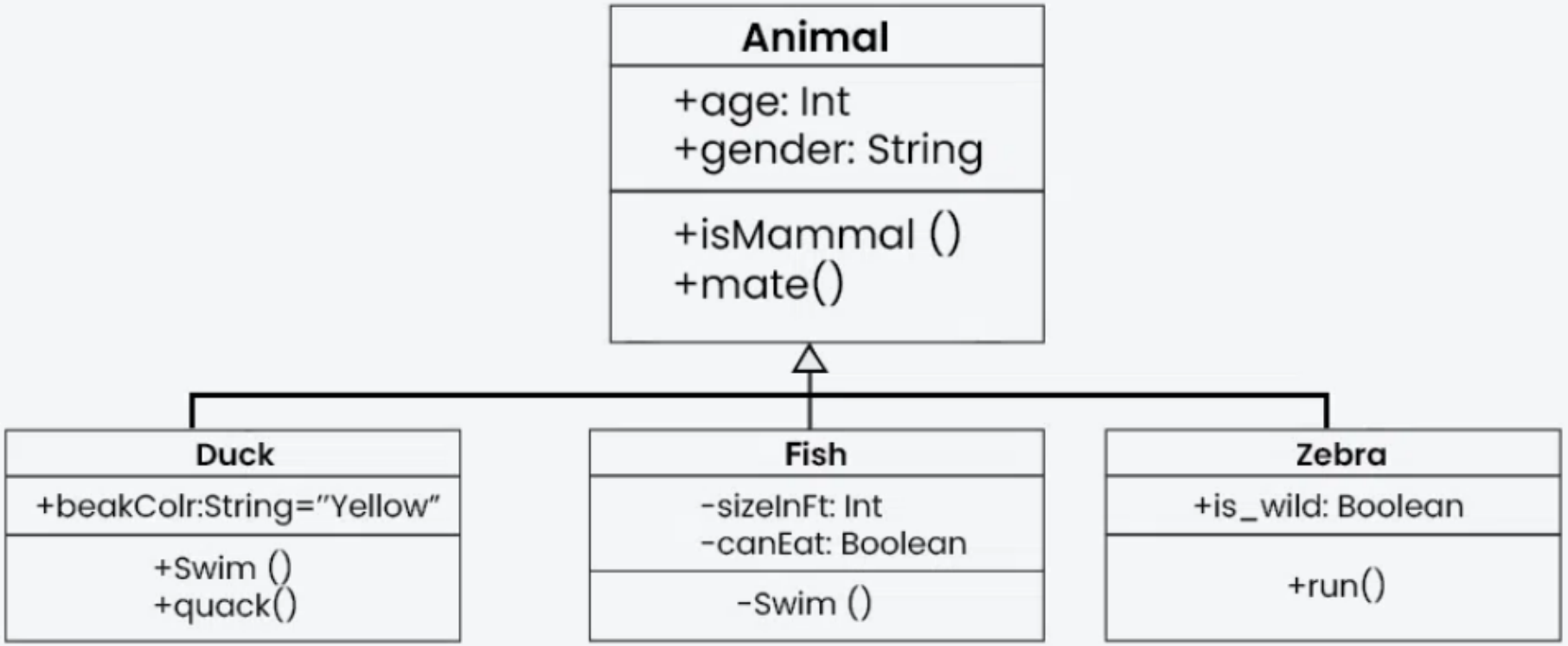
- Classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods



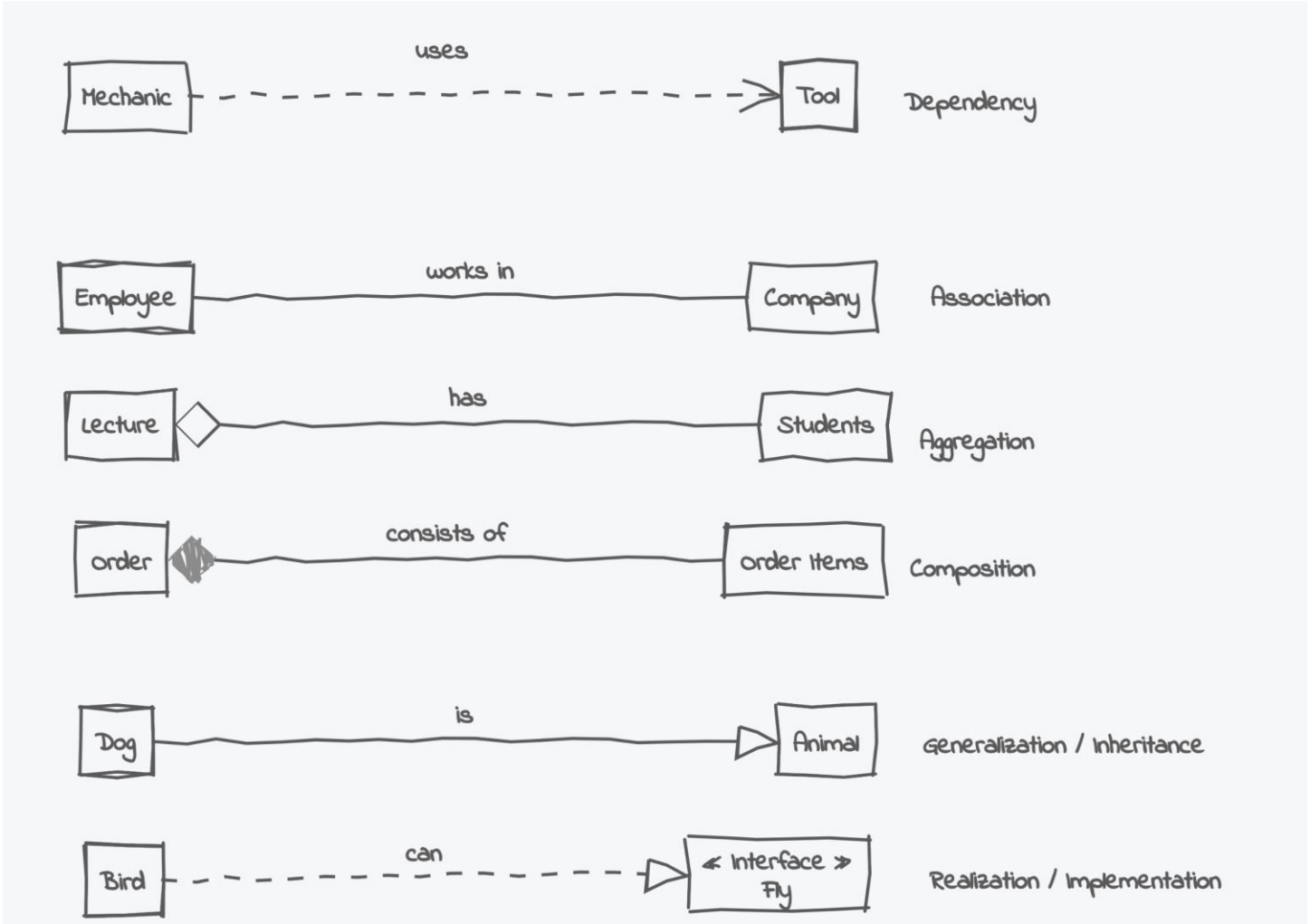
UML class diagram

- **Class Name:** the name of the class is typically written in the top compartment of the class box and is centered and bold
- **Attributes:** Attributes (properties or fields) represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute
- **Methods:** Methods (functions or operations) represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.
- **Visibility Notation:** indicate the access level of attributes and methods:
  - + for public (visible to all classes)
  - - for private (visible only within the class)
  - # for protected (visible to subclasses)
  - ~ for package or default visibility (visible to classes in the same package)

UML class diagram



UML class diagram



Thank you for your attention!