



School of Computer Science and Information Technologies

Course: Programming Languages

2021/2022

# Inheritance and Abstract Classes

## Lecture 4

# INHERITANCE

- *Inheritance* is an important concept in object- oriented languages
- It enables code reusability, by creating new data types using existing data types
  - Inherited classes extend the classes they inherit from
    - They provide additional features

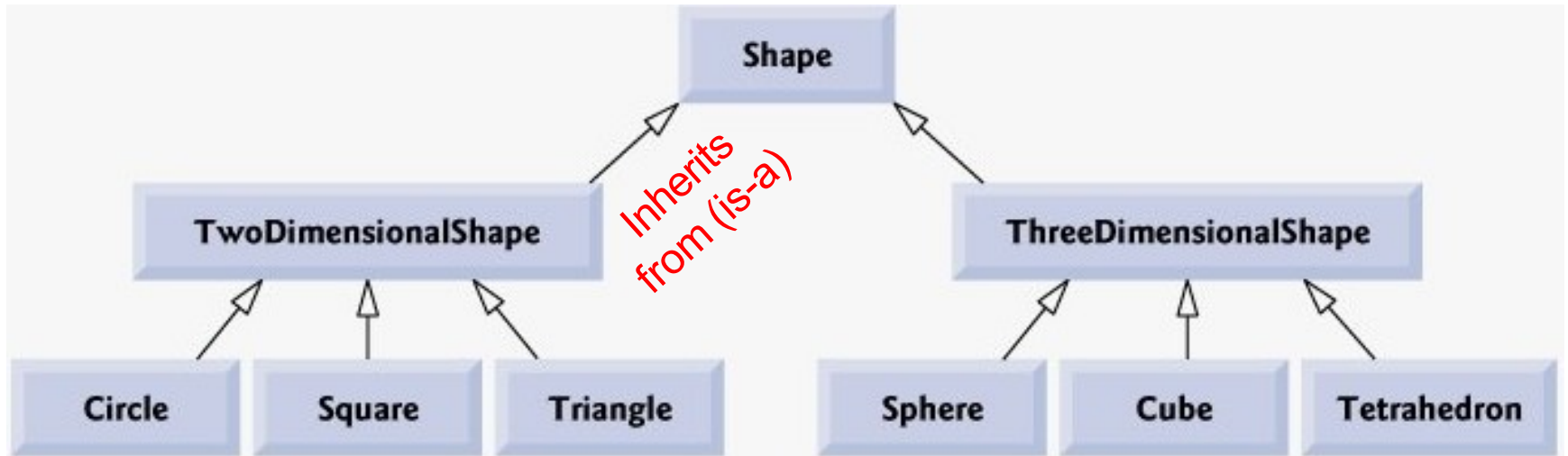
# ELEMENTS OF INHERITANCE

- **Base class / parent class / superclass**
  - The class it is inherited from, i.e. provider of existing data and functional members
- **Derived class / child class / subclass**
  - The class that inherits
  - This class may have its own unique features, besides the ones inherited from the base class
    - This is the most common case

# EXAMPLE

- Let class **Shape** be defined
- A **Shape** can be a **TwoDimensionalShape**
- and a **ThreeDimensionalShape**
  - These classes are derived from **Shape**
- A **TwoDimensionalShape** can be a **Circle**, **Square** and **Triangle**
  - These classes are derived from
  - **TwoDimensionalShape**
- A **ThreeDimensionalShape** can be a
- **Sphere**, **Cube** and **Tethraedron**
  - These classes are derived from **ThreeDimensionalShape**

# EXAMPLE



# EXAMPLE

---

In this example, a **TwoDimensionalShape** would have methods **circumference()** and **area()**

- Every circle, square and triangle would have a circumference and area – those methods would be inherited by their appropriate superclass, i.e. **TwoDimensionalShape**

Also, a **ThreeDimensionalShape** would have methods **area()** and **volume()**

- Every sphere, cube and tetrahedron would have an area and a volume – those methods would be inherited by their appropriate superclass, i.e. **ThreeDimensionalShape**

# SINGLE (“IS-A”) INHERITANCE

- In the previous example, a **TwoDimensionalShape** “is-a” **Shape**, a **Cube** “is-a” **ThreeDimensionalShape** etc
- This is called **single inheritance**, or “**is-a**” **inheritance**, whereas a base class can have more derived classes, but a derived class has just one base class
- The term “is-a” is borrowed from database theory
- This is the most common type of inheritance

# MULTIPLE INHERITANCE

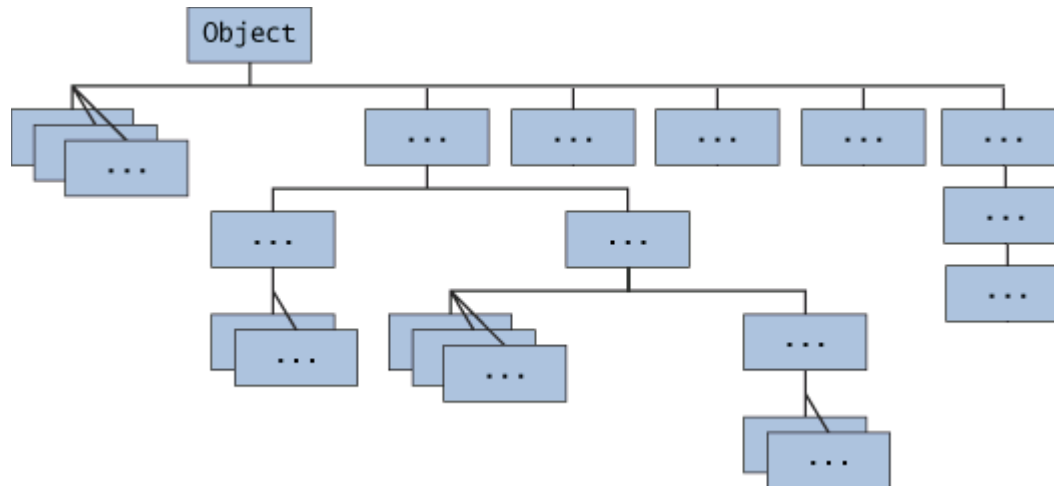
- In *multiple inheritance*, a derived class can have more than one base class
- **Java does not support multiple inheritance**
  - To simulate multiple inheritance, *interfaces* are used
    - They will be introduced subsequently

# EXTENDS

- The keyword `extends` states that the class is a subclass of another class
- Example:
  - `class Vehicle`
  - `{...}`
  - `class Car extends Vehicle`
  - `{...}`
- All data fields and methods from `Vehicle` are now available for objects of type `Car`

# THE JAVA CLASS HIERARCHY

- The `Object` class is at the top of the hierarchy



- It contains the most generic code for all other classes in Java
  - Including user defined classes
- The classes lower in the hierarchy represent a less generic behavior

# THE JAVA CLASS HIERARCHY

- Whenever a user defined class is created, it is implicitly assumed that it inherits from the Object class
  - It is not necessary to explicitly state that

Example:

- `class Vehicle`
- `{...}`
- is the same as
- `class Vehicle extends Object`
- `{...}`
- The Java compiler understands by default that the class inherits from the Objectclass, unless the `extends` keyword specifies that it inherits from a different class

# CASTING

- The object is of the data type of the class from which it has been instantiated. For example
- `public Car myCar = new Car();`
  - `myCar` is of type `Car`
  - `Car` is a subclass of `Vehicle`, which itself is a subclass of `Object`
- `Car` inherits from `Vehicle` and `Object`
- Therefore, `Car` is a `Vehicle` and also is an `Object`, so it can be used whenever objects of the type `Vehicle` or `Object` are needed
- The reverse need not be true: a `Vehicle` may be a `Car`, but not necessarily
  - Similarly, an `Object` may be a `Vehicle` or a `Car`, but not necessarily

# CASTING

---

*Casting* represents using an object of one type instead of an object of a different type, which can be obtained according to the rules of inheritance

For example

```
Object obj = new Car();
```

- Now obj is both an Object and a Car (until the moment when obj is assigned a different object, which is *not* a Car, such as a Vehicle object)
- This is called *implicit casting*

# CASTING

- On the other hand, if the following happens,
  - `Car myCar = obj;`
  - an error upon compilation would be reported
    - `obj` is not known to the compiler as an object of type `Car`
  - Still, the compiler can be *instructed* that the type `Car` *will be* assigned to `obj`
  - using *explicit casting*:
    - `Car myCar = (Car)obj;`
  - This casting forces a run-time check that `obj` is assigned a value of type `Car`, so that the compiler may freely assume that `obj` is a `Car`
    - If `obj` is not a `Car` during run time, an exception will be thrown
      - Exceptions will be presented subsequently

# INSTANCEOF

- The instanceof operator returns true if a certain object is an instance of a specified class, and false otherwise
  - It is a Boolean operator
- Example:

```
if (obj instanceof Car)
{
    Car myCar = (Car)obj;
}
```
- This is useful to determine if a certain object is an instance of a subclass or a superclass
  - This is to certify that there will be no errors due to inappropriate casting
  - Very useful for polymorphism
    - It will be discussed subsequently

# SUPER

- When a member of a subclass has the same identifier as a member of its superclass, the superidentifier may be used to access the member of the superclass
- This identifier can be used to access data fields, methods, as well as the constructor of the superclass
- The elements which are accessed using the super identifier must be accessible to the subclass
  - – i.e. they must be declared as public or
    - protected in the superclass

# SUPER

- Example:

```
class Vehicle
{
    public String type;

    Vehicle() // constructor
    {
        ...
    }

    void ignite() // method
    {
        ...
    }
    ...
}
```

# SUPER

```
class Car extends Vehicle
{
    public String type;

    Car()
    {
        // constructor
        super();          // call to the constructor of the superclass
    }

    void ignite()
    {
        super.ignite(); // call to the method of the superclass
    }

    void setType(String type)
    {
        this.type = type; // access to a field of the current class
        super.type = type; // access to a field of the superclass
    }
}
```

# SUPER



- It is possible that the subclass *overload* the methods of the superclass
  - i.e. assign its own code to them
- The method with a given signature will be invoked instead of the method with the same signature in the superclass
  - The rule of the most local access applies
- But the methods of the superclass will remain available to the subclass and can be accessed using the *super* keyword from *inside the subclass*

# SUPER

- Example:

```
class Supertype
{
    public void M()
    {
        System.out.println("Method of the superclass");
    }
}
```

```
class Subtype extends Supertype
{
    public void M()
    {
        System.out.println("Method of the subclass");
    }

    public void superM()
    {
        super.M();
    }
}
```

# SUPER

```
public class Test
{
    public static void main(String[] args)
    {
        Supertype o1 = new Supertype();
        Subtype o2 = new Subtype();

        o1.M();
        o2.M();
        o2.super.M();
//      o2.super.M();           // ERROR! No superclass of the
                                //Test class contains method M()
    }
}
```

# TYPES OF INHERITANCE IN JAVA

- There is no public, protected or private inheritance in Java
  - Java does not support inheritance using an access modifier
- Access modifiers for members of the superclass do not get modified in the subclass
  - i.e. public elements in the superclass remain public in the subclass etc

# ABSTRACT CLASSES

- An *abstract class* is a class that cannot be instantiated
  - – i.e. no objects can be created from it
- Still, these classes can be inherited from
- Therefore, these classes are useful when they contain fields and methods that can be used in more subclasses

A circular inset image on the left side of the slide shows a collection of colorful wooden letters and numbers (red, yellow, green) scattered on a blue surface. The letters include 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z' and numbers '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'.

# ABSTRACT CLASSES

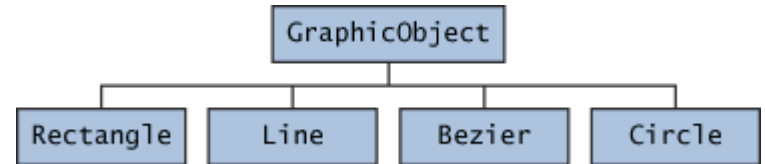
- An abstract class is specified using the keyword `abstract`

# ABSTRACT METHODS

- Abstract methods are methods without code
- Example:
  - `abstract void someMethod();`
- Abstract methods *must be redefined (overridden)*
- in the subclasses
  - – i.e. the subclasses must *offer an implementation* for the abstract methods, so that they would be able to instantiate objects
- If the class contains at least one abstract method, it must be declared as an abstract class

# ABSTRACT METHODS

- Example:



```
abstract class GraphicObject
{
    int x, y; ...
    void moveTo(int newX, int newY)
    { ... }
    abstract void draw();
    abstract void resize();
}
```

# ABSTRACT METHODS

- **Each non-abstract subclass of `GraphicObject` must offer implementations for the `draw()` and `resize()` methods**
  - – Otherwise, the subclasses would also have to be declared as **abstract**

```
class Circle extends GraphicObject
{
    void draw()      { ... }
    void resize()    { ... }
}
```

```
class Rectangle extends GraphicObject
{
    void draw()      { ... }
    void resize()    { ... }
}
```

# ABSTRACT CLASSES AND CONSTRUCTORS

- Abstract classes may contain constructors with appropriate definitions (i.e. code)
- But the attempt to instantiate an abstract class using its (defined) constructor will result in an error
  - Abstract classes, by definition, contain incomplete code and therefore offer *partial implementation*
  - The policy of Java is that only fully implemented classes may instantiate objects
  - Even if all methods of the class contain code, if the class is declared as abstract, the compiler will not allow it objects to be instantiated from it
- The constructors of the abstract class are allowed, so that the constructors of the subclasses would be able to initialize fields inherited from the superclasses using the constructor of the superclass
  - i.e. the constructor of the superclass will be invoked in the constructors of the subclasses

# ABSTRACT CLASSES AND STATIC MEMBERS

- Abstract classes may contain static fields and methods
- These may be invoked as all other static members of non-abstract classes
  - i.e. using the name of the class to qualify the field or method
  - There is only one instance of the static field or method for all subclasses of the abstract class – therefore, all may use it
    - Object code *is* generated for the abstract classes – therefore the static fields and methods are available for use

# ELEMENTS WHICH CANNOT BE ABSTRACT

- **The constructors cannot be declared as abstract**
  - Since they are not inherited by the subclasses, no other class can implement them – hence, they must contain code and be non-abstract
- **Static class members cannot be declared as abstract**
  - There is only one copy of the static member in the class and all of its subclasses, so it cannot be defined (implemented) by another class – it must be defined (i.e. contain a definition) in the current class and be non-abstract

# QUESTIONS

