Java programming

GEIAL31A-B2a

Java language features I.
Final, Generics, Abstract Classes, Interfaces

Tamás Tompa, PhD
assistant professor
Department of Information Technology

Based on materials prepared by Miklós Szűcs

Miskolc, 2026

final keyword

- The final keyword is a non-access modifier used to **restrict modification**
- It applies to **variables (value cannot change) methods (cannot be overridden)** and **classes (cannot be extended)**
- It helps create **constants, control inheritance** and enforce **fixed behavior**
- The following are different contexts where the final is used:

Final Variable	→ To Create constant variable
Final Methods	→ Prevent Method Overriding
Final Classes	→ Prevent Inheritance

final keyword

- **Final variables hold a value that cannot be reassigned** after initialization
- **Final methods cannot be overridden** by subclasses
- **Final classes cannot be extended**
- Initialization rules require that a **final variable must be assigned exactly once** either at declaration or inside constructors or initializer blocks
- **Reference final variables cannot change which object they point** to though the internal state of the object can change
- **Static final variables represent constants** shared across all objects
- **Blank final variables** are declared without initialization and must be **assigned once before use**
- **Local final variables** inside methods **must be initialized** within their block

final variables

- **Final variables hold a value that cannot be reassigned** after initialization

```
public class Geeks{
    public static void main(String[] args) {
        final double PI = 3.14159;
        System.out.println("Value of PI: " + PI);
    }
}
```

```
final int THRESHOLD = 5;
```

static final

- Static final fields are **constants** that belong to the class rather than to any instance
- They are **initialized once and shared across all objects**
  - making them ideal for defining fixed values such as configuration parameters or mathematical constants
- Because they cannot be reassigned, they ensure **consistent and immutable behavior** throughout the program
- Any change in that static variable reflects on the other objects operations
- Initialization is mandatory in case of static final variables

static final variables

- Declaring variables **only as static** can lead to changes in **their values by one or more instances of a class** in which it is declared
- Declaring them as **static final** will help you to create a **constant**
- Only **one copy of the variable exists**, which can't be reinitialized
- In short:
  - **final: it cannot be reassigned once initialized**
  - **static: a single shared copy exists at the class level**

```
public class Person {
    String name;
    static final int minimumSalary = 185000;
}
```

static final variables

- **Cannot be initialize a final static variable inside a non-static method or block**

```
class Test {
    final static int x;

    public static void m() {
    }

    public static void main(String[] args) {
        System.out.println("Compile-time error if x is assigned in method.");
    }
}
```

error: variable x not initialized in the default constructor

final methods

- **Final methods cannot be overridden** by subclasses
  - a final method cannot be overridden
  - its purpose is to **prevent modifications to certain behaviors** when such changes could compromise correct functionality

```
class A {
    final void m1() {
        System.out.println("Final method");
    }
}

class B extends A {
    void m1() { }     // compile-time error
}
```

final classes

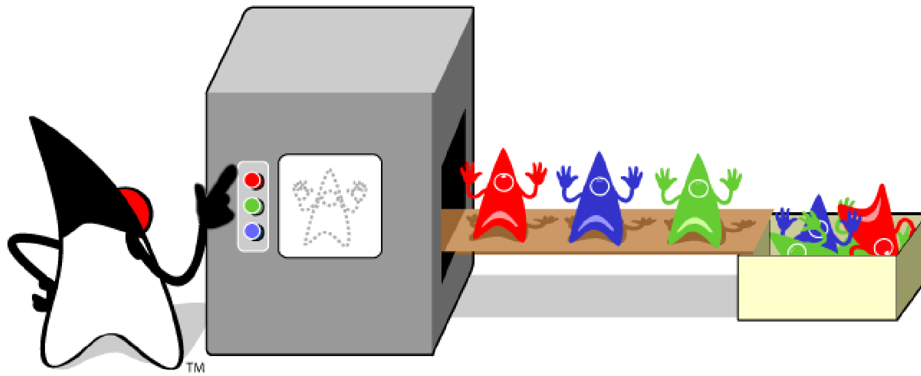
- **Final classes cannot be extended**
  - no other class is allowed to inherit from a final class
  - this prevents subclasses from overriding or changing its behavior
  - **Purpose**
    - to **protect the implementation** of a class
    - to ensure that the class **behavior remains unchanged**
    - to **increase reliability and security** when modification could break correct functionality

```
final class A {
    // fields and methods
}

// Illegal
class B extends A { }
```

Generic types

- Generics allowing to write classes, interfaces, and methods that **work with different data types, without having to specify the exact type** in advance (from Java 5)
- This makes the code more flexible, reusable, and type-safe
- Advantages
  - **code reusability**: write one class or method that works with different data types
  - **type safety**: catch type errors at compile time instead of runtime
  - **cleaner code**: no need for casting when retrieving objects



Generic types

- **Defining classes and methods that are generic without using specific types**
  - old solution: using Object referecne

```
public class Storage {
    private Object object;

    public void set(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}
```

Generic types

- Defining classes and methods that are generic without using specific types
  - old solution: using Object referecne

```
public class StoragePrg {
   public static void main(String args[]) {
      Storage t = new Storage();

      t.set(3);
      int i = (Integer)t.get();

      t.set("Apple");
      int k = (Integer)t.get();
   }
}
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer at StoragePrg.main(StoragePrg.java:7)
```

Generic types

- Defining classes and methods that are generic without using specific types
  - good solution: using generic type

```
class ClassName<T> {
     // Class body where T is used as a placeholder for the data type
}
```

```
public class GenericStorage<T> {
   private T t;

   public void set(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }
}
```

Convention:
Notation of types used as parameters:

- **E** - element
- **K** - key
- **N** - number
- **T** - type
- **V** - value

Generic types

- Defining classes and methods that are generic without using specific types
  - good solution: using generic type
    - the data can be accessed directly; **no cast is required**
    - if a String is accidentally added, **the error is detected at compile time**

```
public class GTPrg {
   public static void main(String args[]) {
      GenericStorage<Integer> gt = new GenericStorage<Integer>();
      gt.set(3);
      int i = gt.get();
      gt.set(„Apple");
   }
}
```

```
GTPrg.java:5: set(java.lang.Integer) in GenericTarolo<java.lang.Integer> cannot be
applied to (java.lang.String)
    gt.set(„Apple");
      ^
```

Generic types

- **Can be created a class that works with different data types using generics**
  - T is a generic type parameter (a placeholder for any data type):
    - when `Box<String>`, `T` becomes String
    - when `Box<Integer>`, `T` becomes Integer

```
public class Box<T> {
  T value;

  void set(T value) {
    this.value = value;
  }

  T get() {
    return value;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    // Create a Box to hold a String
    Box<String> stringBox = new Box<>();
    stringBox.set("Hello");
    System.out.println("Value: " + stringBox.get());

    // Create a Box to hold an Integer
    Box<Integer> intBox = new Box<>();
    intBox.set(50);
    System.out.println("Value: " + intBox.get());
  }
}
```

Abstract keyword

- Abstract is **non-access modifier** in java **applicable for classes, and methods** but **not variables**
- It is used to **achieve abstraction** which is one of the pillars of Object Oriented Programming
  - during the design process, it often occurs that **at the class level we know that a certain method** will be required in subclasses, **but its implementation cannot yet be specified**
- Properties:
  - Abstract classes cannot be instantiated
  - Abstract classes can have both abstract and concrete methods
  - Abstract classes can have constructors
  - Abstract classes can contain instance variables
  - Abstract classes can implement interfaces
  - Abstract methods do not have a body

Abstract keyword

- **Advantages of Abstract Keywords**
  - Provides a way to define a **common interface**: abstract classes can define a common interface that can be used by all subclasses
  - **Enables polymorphism**: allows for greater flexibility and extensibility in the code, can be added new subclasses without changing the code that uses them
  - **Encourages code reuse**: defines common methods and properties that can be reused by all subclasses
  - **Provides a way to enforce implementation**: abstract methods must be implemented by any concrete subclass of the abstract class
  - **Enables late binding**: which subclass to use at runtime

Abstract methods

- An abstract method is a method that is **declared but not implemented**
- It specifies **what must be done, but not how it is done**
- Abstract methods are used to define **required behavior for subclasses**
- Abstract methods **can appear only in abstract classes**
- An abstract class:
  - cannot be instantiated
  - may contain abstract and non-abstract methods

```
public abstract void calculate();
```

```
public abstract class Shape {
    public abstract double area();
}
```

Abstract methods and classes

- **Rules for abstract methods and classes**
  - **a method is abstract if it has no body**, it receives an implementation only when overridden
  - an abstract **method cannot be declared private, final, or static**, because such methods cannot be overridden
  - **a class is abstract if it has at least one abstract method**
  - an **abstract class cannot be instantiated**
  - an **abstract class** is intended to **serve as a superclass for other classes**
    - it is the responsibility of subclasses to override and implement abstract methods
  - a subclass of an abstract class may itself be abstract if it does not implement all abstract methods
  - an abstract class can be used as a reference type (static type of a reference)

Abstract methods and classes

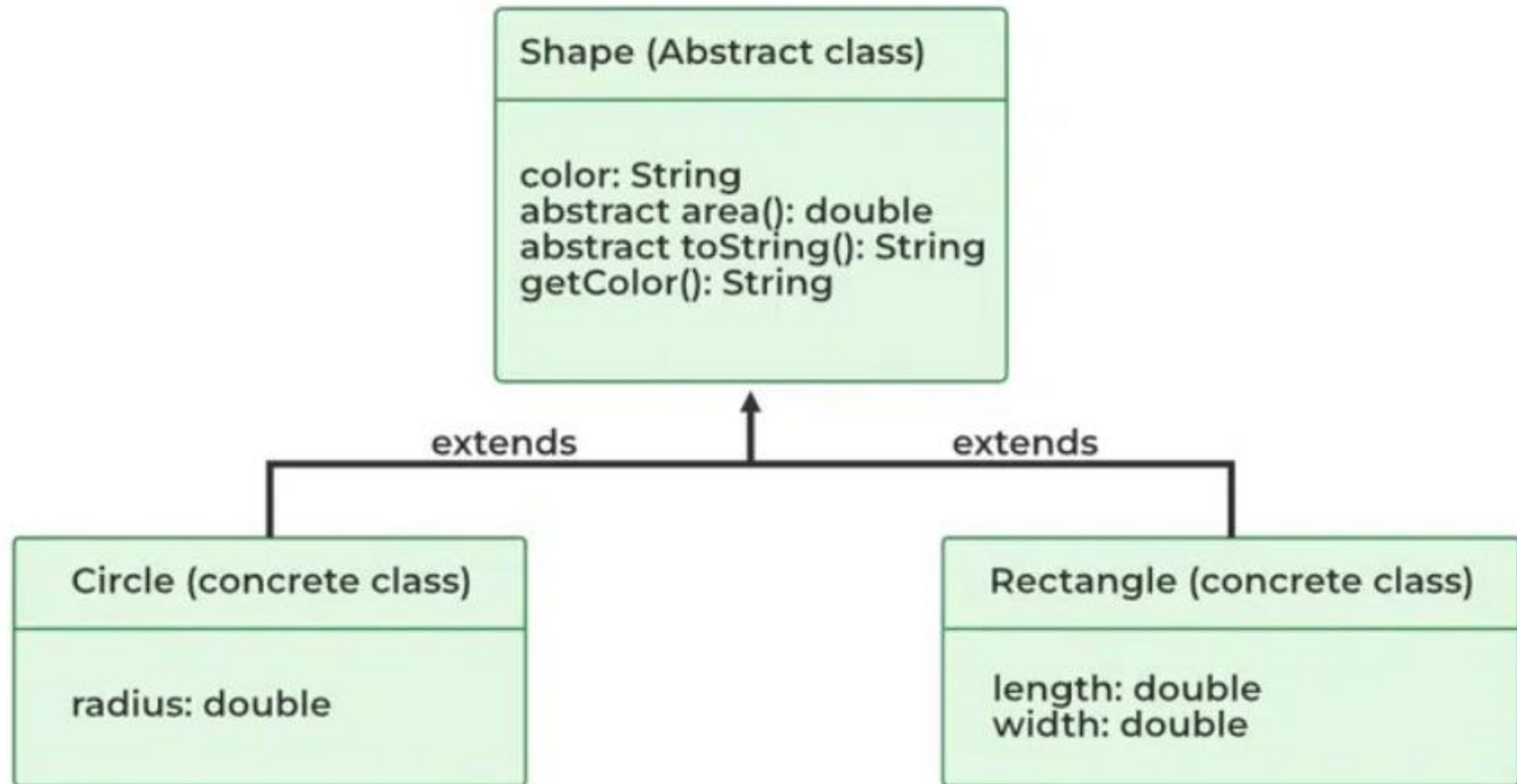
- **Example**

```
public abstract class Shape {
    public abstract double area();
}
```

```
public class Circle extends Shape {
    @Override
    public double area() {
        return Math.PI * r * r;
    }
}
```

Abstract methods and classes

- **Example**



Interface

- **Another way to achieve abstraction**, is with interfaces
- An interface is a **completely "abstract class"** that is used to group related **methods with empty bodies**

- Technical perspective
  - an interface is a **special type of class that can contain only abstract methods and constants**, therefore it cannot be instantiated
  - interfaces are **not used through inheritance** like abstract classes, **but through implementation**
  - **implementation means providing concrete implementations of the methods declared in the interface**
  - **a class can implement any number of interfaces**

- Design perspective
  - an interface **represents a behavioral specification that a class implements**
  - it allows us to **bind code to behavior** (an interface) rather than to a concrete implementation (a class)

Interface

```
interface Animal {
  public void animalSound();
  public void sleep();}
```

- To access the interface methods, the interface must be "implemented" by another class with the **implements keyword** (instead of extends)
- The body of the interface method is provided by the "implement" class

```
class Pig implements Animal {
  public void animalSound() {
      System.out.println("The pig says: wee wee");
  }

  public void sleep() {
      System.out.println("Zzz");
  }
}
```

Interface

- Every interface is implicitly abstract; there is no need to declare it with the abstract modifier
- An interface can extend other interfaces using the extends keyword
- Multiple interface inheritance is supported
- An interface does not contain executable code; the implementation must be provided in the implementing class (implements)
- By convention, interface names start with the letter "I"

```
[modifier] interface Iname extends If1, If2 {
    [declaration of elements]
}
```

Interface

- **Implementation of interfaces:**

```
[modifier] class classname implements If1, If2() { }
```

- if a class implements an interface, it is required to implement all of its methods
- the implemented elements must not be modified
- in the case of methods, the method signature must match exactly
- an element with the same name cannot be inherited and implemented at the same time

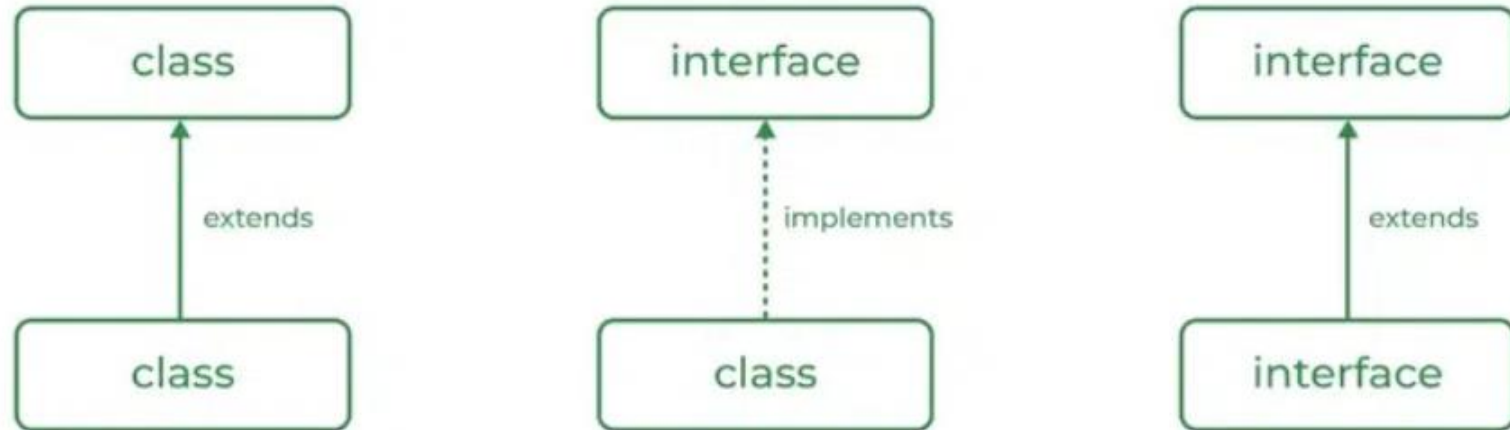
Interface

- **An interface may contain only constants and abstract members/fields**
  - Constants:
    - implicitly public static final
    - the modifiers may be written explicitly, but it is not required
  - Methods:
    - implicitly public abstract, the public modifier may be written explicitly, writing abstract is not recommended
    - no other modifiers are allowed (e.g. static, final, etc.)

- Using an Interface
  - an interface introduces a new reference type, which can be used anywhere a class can be used
  - it can appear in variable declarations
  - a variable of an interface type may refer to an object of any class that implements the interface

Interface

- **Relationship between class and interface**



- Use a class when you need to represent a real-world entity with attributes (fields) and behaviors (methods)
- Use an interface when you need to define a contract for behavior that multiple classes can implement

Interface reference

```
public interface A { … }
public class B implements A { … }
public class C implements A { … }
public class D { … }
```

```
1. A x;
2. x = new B();  //OK!
3. x = new C();  //OK!
4. x = new D();  //NO! Error!
5. x = new A();  //NO! Error!
```

```
A x = new B();
6. System.out.println(x instanceof A);
7. System.out.println(x instanceof C);
```

```
true
false
```

Interface reference

1	Declaration of an interface type variable
2 3	Instantiation of an interface type variable with a class type that implements the interface
4	Attempt to instantiate with a class type that does not implement the given interface
5	Attempt to instantiate with an interface type
6	The dynamic type of the interface type variable named x can be A
7	The dynamic type of the interface type variable named x cannot be C

Interface reference

```
interface A {
    void print();
}
```

```
class B implements A {
    public void print(){
        System.out.println("A");
    }
}
```

```
public static void main(String args[]){
    A x = new B();
    System.out.println(x instanceof A);
    A y = (A)x;
    y.print();
    System.out.println(y);
    System.out.println(y instanceof B);
}
```

true

A

B@9e87d65

true

Interface reference

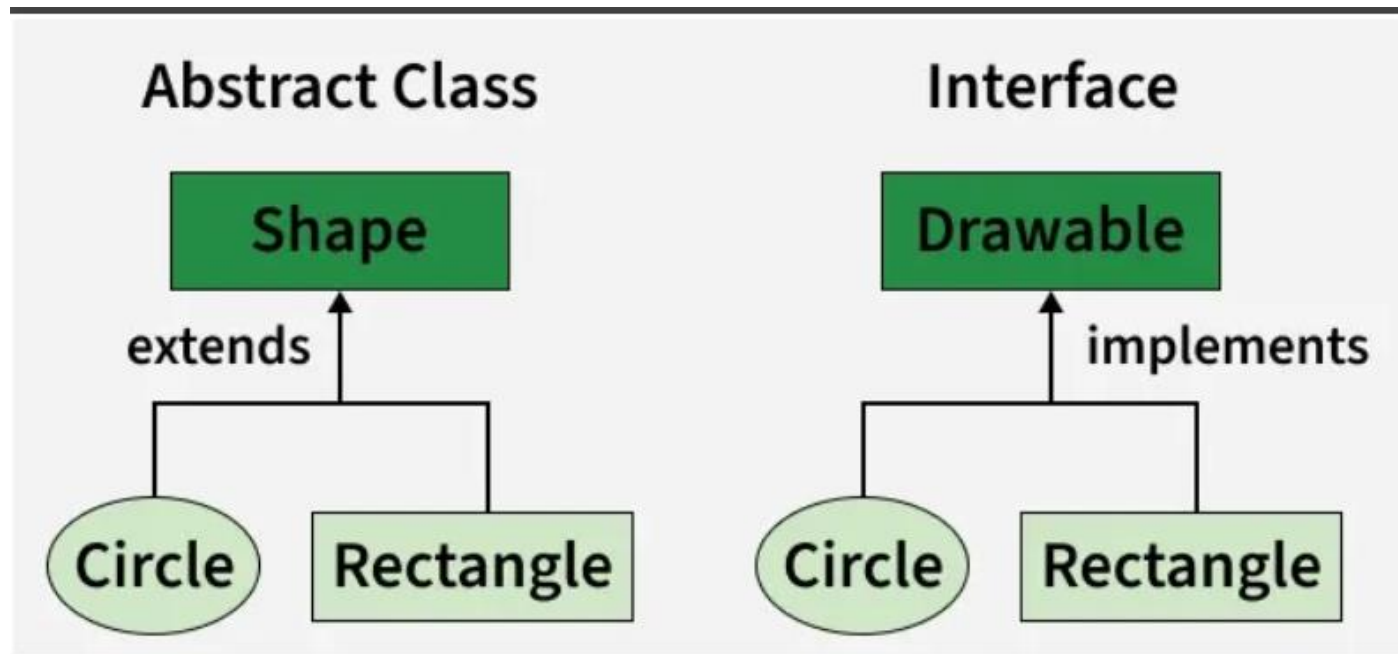
```
interface I1 {…}
interface I2 extends I1 {…}
class A implements I2 {…}
class B implements I1 {…}
```

```
I1 x = new A();   //OK!
I2 y = new A();   //OK!
x = y;            //OK! automatic (implicit) type conversion
y = (I2)x;        //OK! explicit type conversion
A c = (A)x;       //OK! explicit type conversion
x = new B();      //OK!
y = (I2)x;        //NO! Runtime error!
```

x refers to B object
B not implements I2

Abstract class vs Interface

- Abstract classes and interfaces in Java are both **used to achieve abstraction, but they serve different design purposes**
- While they may look similar at first glance, **the way classes interact with them is fundamentally different**



Abstract class vs Interface

Feature	Abstract class	Interface
Instantiation	Cannot be instantiated	Cannot be instantiated
Methods	Can have both abstract and concrete methods	Methods are abstract by default (Java 8+ allows default and static methods)
Variables	Can have variables of any type (static, non-static, final, non-final)	Variables are public static final by default
Inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Constructors	Constructors are allowed	Constructors are not allowed
Access Modifiers	Members can have any access modifier (private, protected, public)	Methods are public by default

Thank you for your attention!