Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

OO Programming Techniques in Java

Abstract Classes and Interfaces

T. Cioara, V. Chifu, C. Pop
2025

Abstract Classes

Definition

- Class declared with the ***abstract*** keyword which may include **abstract methods**
- **Used to generalize behavior**

Contents

- Instances and class (static) variables, constructors, abstract methods, non-abstract methods

Rules

- Abstract classes can not be instantiated into objects
- References to abstract classes could be passed as parameters in methods
- All abstract methods are inherited
- Abstract methods must be defined in the subclasses

Abstract Classes

- **Consider Shape2D as an abstract class**

```
public abstract class Shape2D
{
    protected Point org;
    public Shape2D(Point org) {
        this.org =org;
        System.out.println("Origin Point initialized in Shape2D");
    }
    public String whoAreYou() {return "SHAPE2D"; }
    public abstract double perimeter();
    public abstract double area();
}
```

- **Shape2D cannot be instantiated**

```
Shape2D myShape2D = new Shape2D(new Point( x: 3,  y: 5));
```

'Shape2D' is abstract; cannot be instantiated

Abstract Classes

- **Example – constructor usage**

```
public class Circle extends Shape2D {
    private double radius;
    public Circle(Point pc, double radius) {
        super(pc);
        this.radius = radius;
        System.out.println("Initialized the radius of the circle");
    }
    public String whoAreYou() { return "CIRCLE"; }
    public double perimeter() { return 2.0 * Math.PI * radius; }
    public double area() { return  Math.PI * radius * radius; }
}
```

  - **Usage**

```
Circle myCircle = new Circle(new Point( x: 3,  y: 5),  radius: 10);
```

  - **Output**

```
Origin Point initialized in Shape2D
Initialized the radius of the circle
```

Interfaces

Definition

- Reference type **used to standardize behaviour**
- Types: top level interface, nested interface, annotation type interface, functional interface, marker interface

Contents

- Constants, method signatures (i.e., abstract methods without implementation), default methods (since Java 8), static methods (since Java 8), private methods (since Java 9)

Rules

- Interfaces can not be instantiated
- Reference variables can have as type an interface, but any object assigned to them must be instance of a class that implements the interface
- A class that implements an interface, but does not implement all the methods of the interface must be declared abstract

Interfaces

- **Definition**

```
[modifiers] interface <interface-name>{
      <constant-declaration>
      <method-declaration>
      <nested-type-declaration>
}
```

- **Examples**

```
package …
interface Trackable {
   // … interface members
}
```

or equivalent:

```
abstract interface Trackable {
 // … interface members
}
```

```
public interface Trackable {
 // … interface members
}
```

or equivalent:

```
public abstract interface Trackable {
 // … interface members
}
```

- **Notes**
  - Java lower than 8 - all members are public
  - Java 9 - allows private methods

Interfaces

- **Constant fields**

```
public interface Status {
  int ON = 1;
  int OFF = 2;
}

public class StatusTest {
 public static void main(String[] args) {
    System.out.println(“Status.ON = " + Status.ON);
    System.out.println("Choices.OFF = " + Status.OFF);
 }}
```

Implicitly public, static and final!

- **Good programming practice!**
  - Do not declare an interface to only have constant fields.
  - To group constants in one construct, use a class, not an interface.
  - Consider **enum** to declare your constants (provides type safety and compile-time checks for your constants)

Interfaces

- **Abstract methods**

Implicitly abstract and public!

```
public interface ATM {
    boolean login(int account) throws AccountNotFoundException;
    boolean deposit(double amount);
    boolean withdraw(double amount) throws InsuffAmountException;
    double getBalance();
}
```

- Abstract methods of an interface are inherited by classes that implement the interface
- Classes should override them to provide an implementation.

Interfaces

- **Static methods**

```
public interface <name> {
    …
    // static convenience method(s)
    public static <returnType> <methodName> (<parameters>) {
        // … method body
    }
}
```

  - Invocation: **<interface-name>.<static-method>**
  - Java versions < 8 include many utility classes with static methods associated with interfaces
    - Collection/Collections, Path/Paths, Executor/Executors, etc.

Static methods in an interface are not inherited by the implementing classes or sub interfaces (as different from static methods in a class)

Interfaces

- **Default methods**

```
public interface <name> {
   …
   public default <returnType> <methodName> (<parameters>) {
      // … method body
   }
}
```

  - Allow for evolving the existing interfaces without breaking the existing code

All classes implementing the interface will inherit the default implementation => the classes will not break

Interfaces

- **Default methods**

Default method (DM) in interface VS a concrete method (CM) of a class that implements the interface		
Differences	**CM**	Can access the instance variables of the class
	DM	• Does not have access to the instance of variables of the class that implements the interface. • Has access to the members of the interface.
Similarities	**CM and DM**	• Provide an implementation. • Have access to the keyword **this** in the same way. • Can use their parameters. • Can have a throws clause

Interfaces

- **Default methods - example**

```
public interface Translatable {
  void setX(double x);
  void setY(double y);
  double getX();
  double getY();
}
```

Extended with the relativeTranslate() default method

```
default void relativeTranslate (double dX, double dY){
   double newX = getX() + dX;
   double newY = getY() + dY;
   setX(newX);
   setY(newY);
 }
```

Will not affect the classes already implementing Translatable (e.g., Table)

```
public class Table implements Translatable {
  private double x;
  private double y;

  public Table() {
    // By default at (0.0, 0.0)
  }

  public Table(double x, double y) {
    this.x = x;
    this.y = y;
  }
 @Override
 public void setX(double x){this.x = x;}

 @Override
 public void setY(double y){this.y = y;}

 @Override
 public double getX() { return x; }

 @Override
 public double getY() { return y; }

 @Override
 public String toString() {
  return "Table (" + x + ", " + y + ")";}
}
```

Interfaces

- **Default methods as optional methods**

```
public interface Named {
  void setName(String name);

  default String getName() {
     return "John Doe";
  }

  default void setNickname(String nickname) {
                      throw new UnsupportedOperationException("setNickname");
  }

  default String getNickname() {
                       throw new UnsupportedOperationException("getNickname");
  }
}
```

- Override the setNickname() and getNickname() methods to provide implementation, if the class supports a nickname.
- Otherwise, they throw a runtime exception to indicate that they are not supported.

Interfaces

- **Private methods**

Context and motivation

```
public interface AnInterface {
    default int m1(… parameters …) {
        // … m1 specific code
        // … common code m1 and m2
    }

   default int m2(… parameters …) {
        // … m2 specific code
        // … common code m1 and m2
    }
}
```

Solution

```
public interface AnInterface {
    default int m1(… parameters …) {
        // … m1 specific code
       // call to pm1m2
    }

   default int m2(… parameters …) {
        // … m2 specific code
        // call to pm1m2
    }

    private int pm1m2( … parameters …) {
        …
       return val;
    }
}
```

Interfaces

- **Supported Modifiers in Method Declarations**

Modifiers	Supported?	Description
`public static`	Yes	Supported since JDK 8.
`public abstract`	Yes	Supported since JDK 1.
`public default`	Yes	Supported since JDK 8.
`private static`	Yes	Supported since JDK 9.
`private`	Yes	Supported since JDK 9. This is a non-abstract instance method.
`private abstract`	No	This combination does not make sense. A private method is not inherited, so it cannot be overridden, whereas an abstract method must be overridden to be useful.
`private default`	No	This combination does not make sense. A private method is not inherited, so it cannot be overridden, whereas a default method is meant to be overridden, if needed.

Interfaces

- **Nested interfaces**
  - Declared in the body of another class or interface
  - Static by default

```
interface Showable{
  void show();
  interface Message{
   void msg(); }
}

class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();
  message.msg();
 }
}
```

- **Group related interfaces so that they can be easy to maintain**

Source

Interfaces

- **Functional interfaces**
  - Interface with only one abstract method annotated with the @FunctionalInterface annotation
    - The compiler will verify the annotated interface if it really contains only one abstract method; otherwise, the interface declaration will not compile
  - The static and default methods are not counted to designate an interface a functional interface

```
@FunctionalInterface
public interface Adder {
   public int add(int a, int b);
}
```

Interfaces

- **Annotation Type interfaces – motivating scenario**

```
public class Employee {
   public void setSalary(double salary) {
      System.out.println("Employee.setSalary():" + salary);}
}

public class Manager extends Employee {
   // Override setSalary() in the Employee class
   public void setSalary(int salary) {
      System.out.println("Manager.setSalary():" + salary);}
}
…
Employee ken = new Manager();
int salary = 200;
ken.setSalary(salary);
// Output: Employee.setSalary():200.0 -> not the expected output
```

Source: K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams, Apress, 2014 – Chapter 1

Interfaces

- **Annotation Type interfaces – motivating scenario**

Use the override annotation instead of a comment to save debugging time!

```
public class Manager extends Employee {
   @Override
   public void setSalary(int salary) {
      System.out.println("Manager.setSalary():" + salary);}
}
```

  - Indicates the programmer's intention to override the method in the superclass
  - At source code level, it serves the purpose of documentation.
  - When the compiler comes across the @Override annotation, it makes sure that the method really overrides the method in the superclass.
    - Annotations document the source code BUT they also have compiler support
    - Annotations instruct the compiler to enforce some rules

Source: K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams, Apress, 2014 – Chapter 1

Interfaces

- **Annotation Type interfaces - definition**

```
<modifiers> @ interface <annotation-type-name> {
        // Annotation type body goes here
}
```

The same as for an interface declaration (public/package level)

Static and default methods are not allowed here

- Associates (or annotates) metadata (or notes) to the program elements
  - package, class, interface, field of a class, local variable, method, etc.) in a Java program
- Acts like a decoration or a note for the program element that it annotates

```
@annotationType(name1=value1, name2=value2, names3=values3...)
```

Source: K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams, Apress, 2014 – Chapter 1

Interfaces

- **Annotation Type interfaces – Restrictions**
  - **R1**: An annotation type cannot inherit from another annotation type
  - **R2**: Method declarations in an annotation type cannot specify any parameters
  - **R3**: Method declarations in an annotation type cannot have a throws clause
  - **R4**: The return type of a method declared in an annotation type must be one of the following types: any primitive type, java.lang.String, java.lang.Class, an enum type, an annotation type, an array of any of the previous mentioned type
  - **R5**: An annotation type cannot declare a method, which would be equivalent to overriding a method in the Object class or the Annotation interface
  - **R6**: An annotation type cannot be generic

Source: K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams, Apress, 2014 – Chapter 1

Interfaces

- **Annotation Type interfaces - example**

major and minor are annotation elements having the int data type

```
package annotationexample;
public @ interface Version {
   int major();
   int minor();
}

@Version(major = 1, minor = 0)
public class VersionTest {
   // Annotation for instance variable xyz

   @Version(major = 1, minor = 1)
   private int xyz = 110;

   // Annotation for constructor VersionTest()
   @Version(major = 1, minor = 0)
   public VersionTest() {
   }

   // Annotation for the printData() method
   @Version(major = 1, minor = 0)
   public void printData() {}
}
```

Source: K. Sharan, Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams, Apress, 2014 – Chapter 1

Interfaces

- **Marker interfaces (i.e., tagging interface)**
  - Interface with no methods declared
  - Provides run-time type information about objects, so the compiler and JVM have additional information about the object
  - Examples in Java: Serializable, Cloneable, Remote
  - Considered as pointing to code smells => newer developments favor annotations instead of marker interfaces

Interfaces

- **Marker interfaces – Object serialization in Java**
  - Persist an object state even after the program is not running
  - Object whose class implements the **Serializable** interface
    - into a sequence of bytes that can be written to disk and later restored to recreate the original object
  - Mechanism for implementing a **lightweight persistence**
    - The user must explicitly serialize and deserialize the objects in a program
  - A serialized object can be transmitted over the network
  - The transient keyword can be used to turn off serialization for a field
  - Static fields are not serializable

Interfaces

- **Marker interfaces – Object serialization in Java**

```
public class SerializationOperations {
    public static void main(String[] args) throws ParseException,
                IOException, ClassNotFoundException {

        FileOutputStream fileOutputStream = new
                        FileOutputStream("john_doe.txt");
        ObjectOutputStream objectOutputStream = new
                         ObjectOutputStream(fileOutputStream);
        objectOutputStream.writeObject(user);
        objectOutputStream.flush();
        objectOutputStream.close();

        FileInputStream fileInputStream = new
                                FileInputStream("john_doe.txt");
        ObjectInputStream objectInputStream = new
                              ObjectInputStream(fileInputStream);
        User restoredUser = (User) objectInputStream.readObject();
        objectInputStream.close();
        System.out.println(restoredUser.toString());
    }
}
```

Serialization

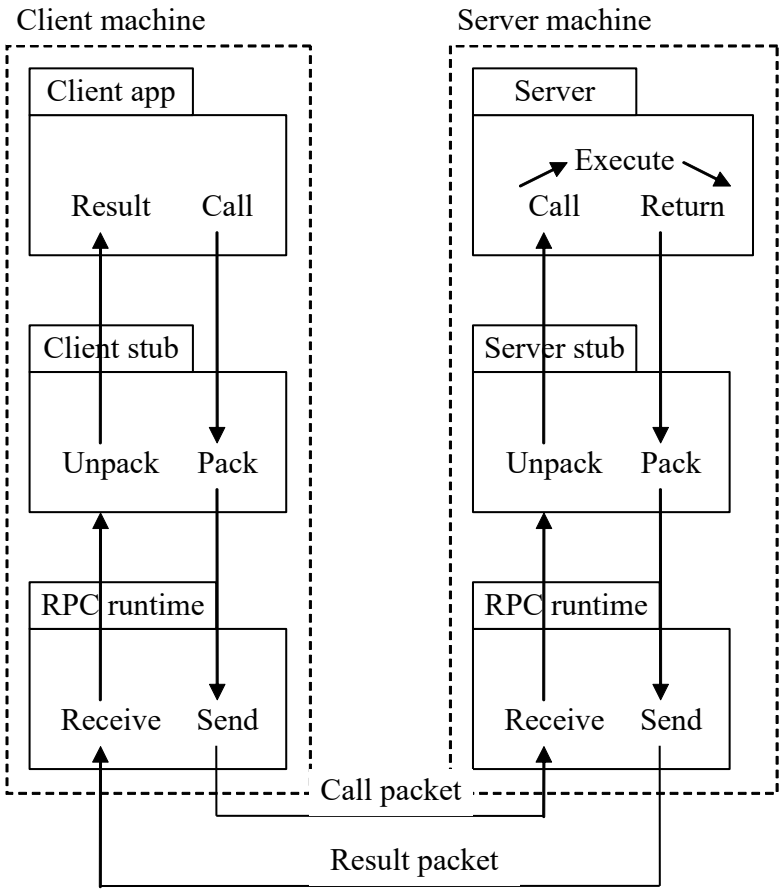
Deserialization

Used to remember versions of the Serializable class to verify that a loaded class and the serialized object are compatible.

```
public class User implements
Serializable {
    private static final long
            serialVersionUID = 1L;
    private String username;
    private transient
                String password;
    private String name;
    private LocalDate birthdate;
    // get and set methods…
```

Interfaces

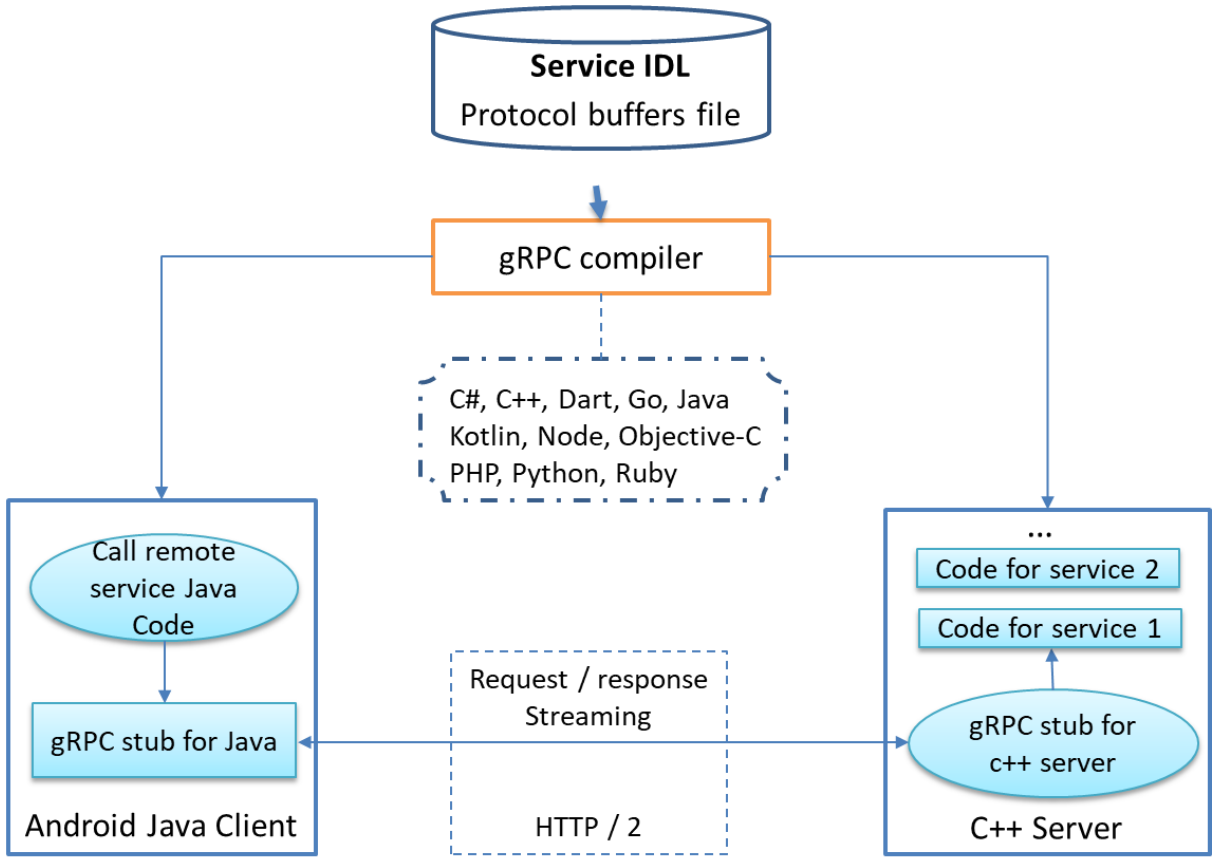
- **Marker interfaces – Object serialization in Java**
  - Java object serialization is limited – only Java programs can deserialize these objects
    - Java uses reflection to get the data from an object's field that need to be serialized
    - The deserialization does not use the constructor of the class
      - Creates an empty object and uses reflection to write data to the fields
  - A more interoperable solution is to convert data to other formats
    - XML, JSON, etc.



Remote Procedure Call

Interfaces

- **Cross technology interoperability**
  - Use of Interface Definition Language



Interfaces versus Abstract Classes

- **Features**
  - **Common**: Both define a contract that must be implemented by a class
  - Interface inheritance – allows for multiple interface extensions
  - A class can implement multiple interfaces
  - A class can only extend one other class

```
public interface IF {
   public void m();
}
```

```
public abstract class AC {
   public abstract void m();
}
```

Interfaces versus Abstract Classes

Able or *can do* vs. *is-a*

- Interface
  - Is not describing class main role
  - Describes the peripheral class abilities
  - Example
    - Bicycle may implement Recyclable
    - Many other (unrelated) classes may implement Recyclable
- Abstract class
  - defines the core identity of its descendants
  - Example – class Dog
- Implemented interfaces
  - Specifies what a class can do
  - Doesn't specify what a class is

Interfaces versus Abstract Classes

Plug-in

- Interface
  - New implementations – no common code with previous implementations
  - Start from scratch
  - Freedom to implement a totally new internal design

- Abstract class
  - AC should be used as it is (good or bad)
  - Imposes a certain structure to the new implementer

Third party functionality

- Interface
  - Interface implementation may be added to any existing third-party class
- Abstract class
  - Third-party class must be rewritten to extend from the abstract class

Interfaces versus Abstract Classes

Homogeneity

- Interfaces
  - All the various implementations share is the method signatures
- Abstract class
  - All various implementations are all of a kind and share a common status and behavior

In terms of subclasses

- Abstract class' subclasses are homogeneous
- Interface subclasses are heterogeneous, use interface

Interfaces versus Abstract Classes

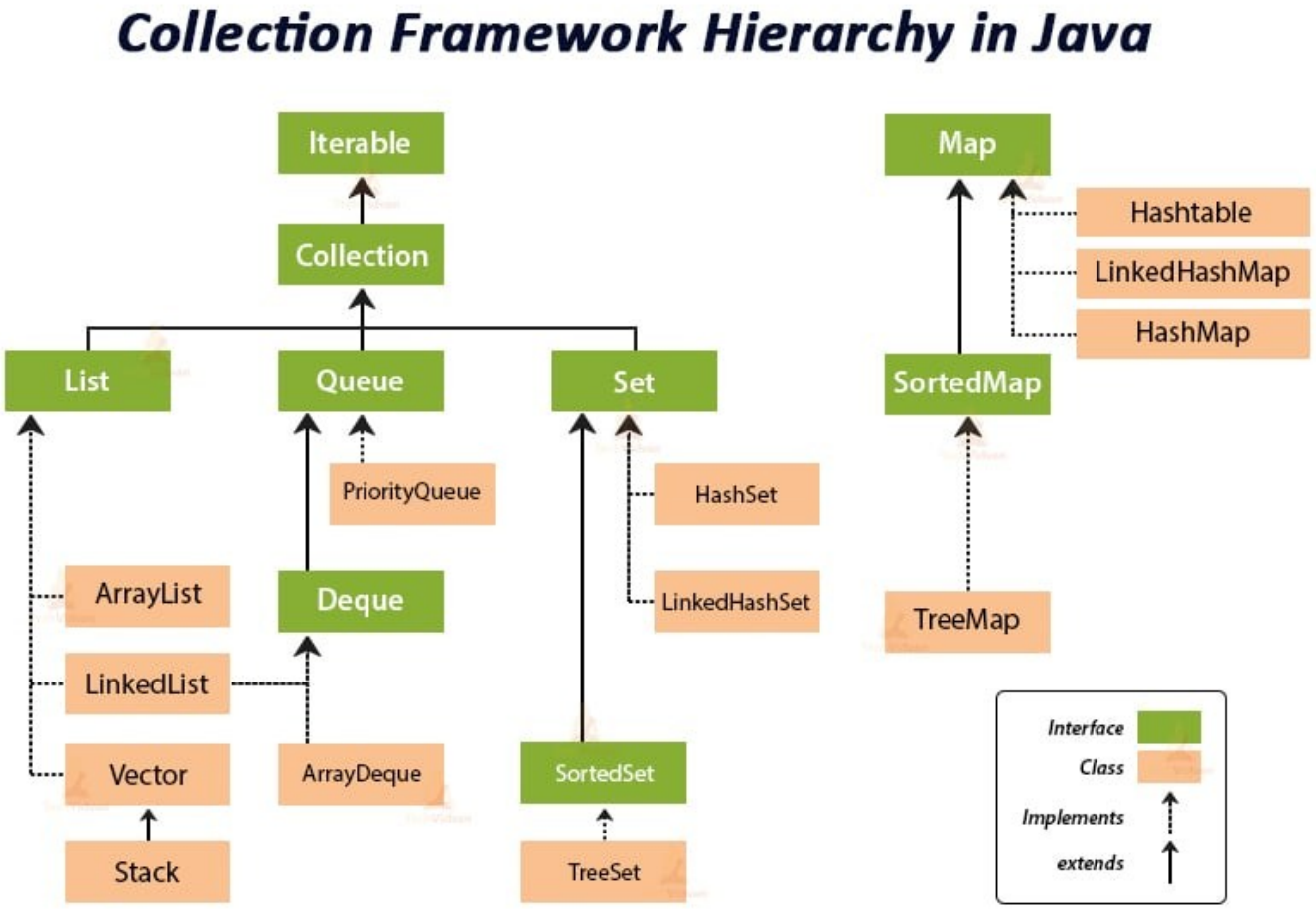
Pro interface

- If you think that the API will not change for a long time
- When you need something similar to multiple inheritance

Pro abstract class

- You plan on using inheritance (hierarchies of classes);
  - Abstract classes provide a common base class implementation to subclasses

JCF Interfaces and Abstract classes



Source

JCF Interfaces and Abstract classes

Level 0 - Top hierarchy

Interfaces such as Collection, List

Describe contracts (behavior specification)

Level 1

Abstract classes such as AbstractList

Provide partial implementations

Level 2

Concrete classes such as ArrayList or Vector

Define all abstract methods that are not already defined

Benefits of programming

Much of the implementation is already done in superclasses

Easy switch between implementations

Develop parallel implementations