While folks are joining

Get you laptops ready and login to your **replit** accounts.

We will be coding away in the session!

Crio Sprint : JAVA-2

Session 5

Today’s Session Agenda

- Abstraction
- Abstract Classes
- Interfaces
- Keywords
    - **abstract**
    - **interface**
    - **implements**

Abstraction - Scenario #1 SmartPhone

- We’ve all made phone calls
- What does a normal user do when (s)he receives a call ?
  - Pick up the call
  - Reject the call
- Does user need to know how the call is created?
  - No, there’s a lot of code that runs in the background.
  - Not relevant to the user.
  - That’s the beauty of abstraction - you can still operate a phone without knowing how it internally functions.

Abstraction - Scenario #2 ATM

- What operations can be done at an ATM?
  - Cash withdrawal
  - Mini statement
  - Change password, etc.
- List down the steps for cash withdrawal
  - Enter the Card
  - Input the Pin
  - Input Amount for cash withdrawal
  - Validate Amount (whether account has balance)
  - Deduct Amount from Account
  - Dispense Cash

Activity 1 - ATM Machine

```
class ATMMachine {
 public void enterCard(){
   System.out.println ("Card Verification");
 }
 public void enterPin(){
   System.out.println ("Pin Verification");
 }
 public void cashWithdrawal (){
   System.out.println ("To withdraw cash from ATM");
 }
 public void validateWithdrawAmount(){
     System.out.println ("Validate the Amount to be withdrawn");
 }
 public void updateAmount(){
     System.out.println ("Update the Amount after withdrawal");
 }
 public void cashDispense(){
   System.out.println ("Dispense the cash from ATM");
 }
 public void miniStatement () {
   System.out.println ("Get the mini statement");
 }
}
```

Compile and Run the below program.

```
public class Main{
 // Mimic user behavior
  public static void main (String[]args){
     ATMMachine am = new ATMMachine();
     am.enterCard();
     am.enterPin();
     am.cashWithdrawal();
     am.validateWithdrawAmount();
     am.updateAmount();
     am.cashDispense();
  }
}
```

- Look at the output. Does it make sense?
- Remove the below method and run again.
  - validateWithdrawAmount()
  - updateAmount()
- Does it make sense?
- Where should these removed methods be invoked?

Activity 1.1 - ATM Machine

```
class ATMMachine{
  public void enterCard (){
    System.out.println ("Card Verification");
  }
  public void enterPin (){
    System.out.println ("Pin Verification");
  }
  public void cashWithdrawal(){
    System.out.println ("To withdraw cash from ATM");
    validateWithdrawAmount();
    updateAmount();
    cashDispense();
  }
  private void validateWithdrawAmount(){
    System.out.println ("Validate the Amount to be withdrawn");
  }
  private void updateAmount(){
    System.out.println ("Update the Amount after withdrawal");
  }
  private void cashDispense (){
    System.out.println ("Dispense the cash from ATM");
  }
  public void miniStatement (){
    System.out.println ("Get the mini statement");
  }
}
```

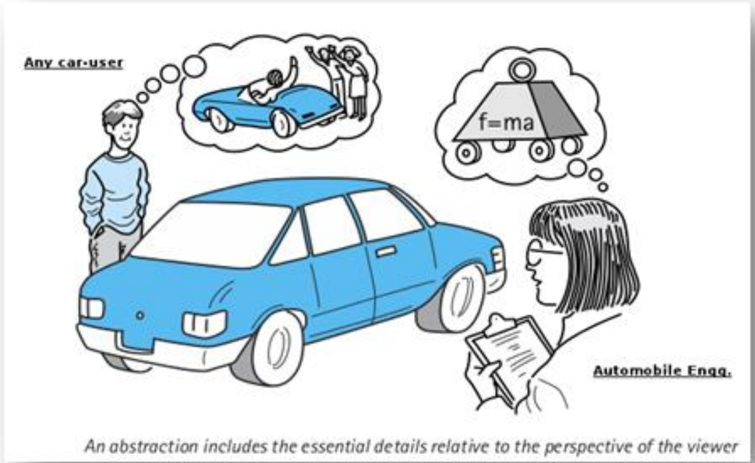
Run the program again.

```
public class Main{
  public static void main (String[]args){
    ATMMachine am = new ATMMachine();
    am.enterCard();
    am.enterPin();
    am.cashWithdrawal();
  }
}
```

- Look at the output. Does it make sense?
- What did we accomplish?

Curious Cats

- How is Abstraction different from Encapsulation?
  - Abstraction refers to
    - **Hide the implementation details** or **complexity.**
    - **Expose only relevant functionality** to the user.
    - End user focus is on ***what the*** object does instead of ***how it*** does it.
    - By using access specifiers, but there are other ways as well, as we'll see next



An abstraction includes the essential details relative to the perspective of the viewer

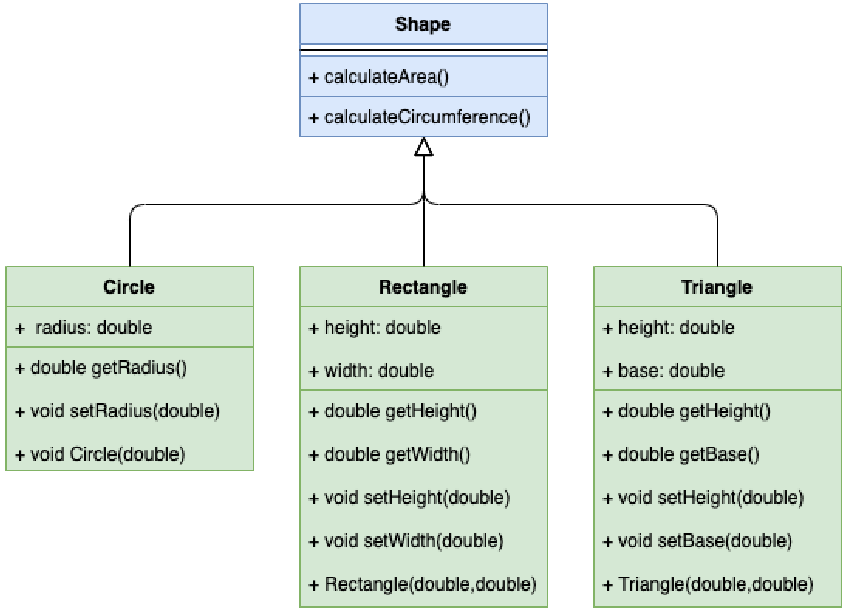
  - Encapsulation refers to
    - **Grouping the behavior and data in a single logical unit** and enable **data hiding** to protect the access of data from the outside world.
    - Use of access specifiers

Curious Cats

- Other than making methods ***private,*** are there other ways to achieve abstraction?
  - **abstract** class (0% - 100% Abstraction)
  - **interface** (100% Abstraction)

Activity 2 - Shape of Shapes

- Your manager has asked you to implement Triangle, Rectangle and Circle Classes and provided a UML diagram as below:



- You provided this solution
  - Shape of Shapes
- Your manager didn't like what you did with the *Shape* class. Can you identify the issue?
- How can we resolve the issue?
  - Using *abstract* methods and *abstract* classes.

What is an Abstract class?

- Cannot be instantiated, but they can be subclassed
- May contain regular fields
- May contain **abstract methods**

Syntax: `public abstract class Animal{}`

- Declared in **abstract classes.**
- **Body must be empty (no curly braces)**
- It doesn’t provided **any actual implementation**

Syntax: `public abstract void build();`

Activity 2.1 - Shape of Shapes

- Improve the solution using **abstract** keyword.
    - Shape of Shapes
- Improved Solution
    - Shape of Shapes Improved

Curious Cats

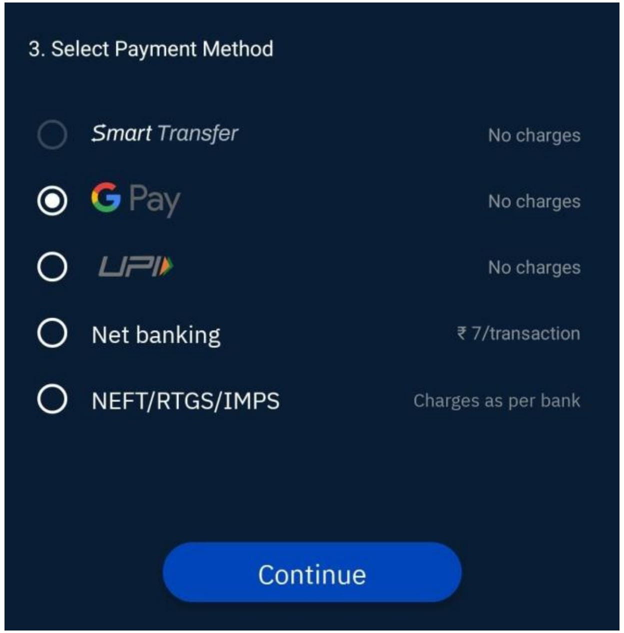
- Why does an abstract class have a constructor even though we cannot create an object?
  - Why do abstract classes in Java have constructors? - StackOverflow
- Can a non-abstract class have an abstract method?
  - No. A non-abstract class cannot have an abstract method whether it is inherited or declared in Java.
- Should an abstract class have at least one abstract method?
  - No.
- Can abstract methods be made private?
  - No. Private methods(cannot be inherited) and can't be overridden, so they can't be abstract.
- Can we declare an abstract method final or static in java?
  - Can we declare an abstract method final or static in java? (tutorialspoint.com)

Summary - Abstract classes and Methods

- **abstract** keyword is a **non-access modifier**.
- Can be used with **methods** and **classes**.
- Abstract keyword **cannot be used with** following keywords
  - final
  - static
  - private
- Abstract classes **cannot be instantiated**.
- Subclass of an abstract class must
  - **Implement all abstract methods** in the super-class
  - Or **be declared abstract itself.**
- A class that contains at **least one abstract method** must be declared abstract.
- Abstract methods have no body.

Activity 3 - Payment Option

- Have you ever paid your credit card bill?
- What are the different payment options available?



3. Select Payment Method

○ Smart Transfer	No charges
[✓ G Pay]	No charges
○ UPI	No charges
○ Net banking	₹ 7/transaction
○ NEFT/RTGS/IMPS	Charges as per bank

- Compile and run the below program.
  - Payment Option
- Can you support 100 more payment methods?
- What is the drawback of the current approach?
- How can we improve it?
  - Using Interfaces

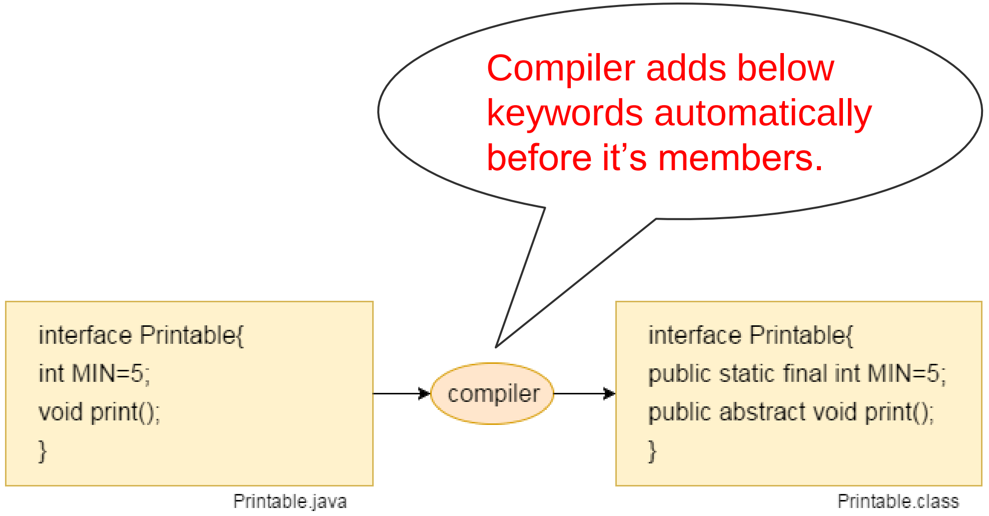
Java Interface

- What is an **interface** and why do we need it?
  - A contract which specifies what a class must do and not how.
  - Anybody can implement this contract and we can easily switch between these implementations.
- How to declare an interface?

```
interface <interface_name>{
    // declare constant fields
    // declare methods

}
```

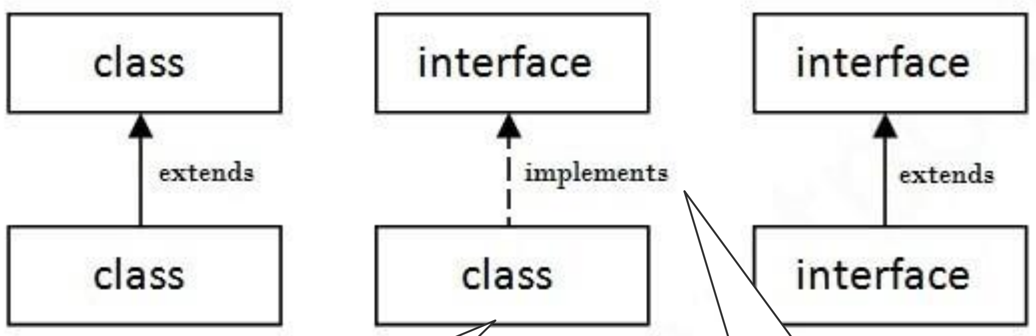
- fields are public, static and final by default
- methods are public and abstract.


Compiler adds below keywords automatically before it's members.
compiler
Printable.java
Printable.class

```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

Relationship between Classes and Interfaces

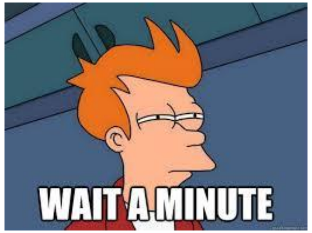


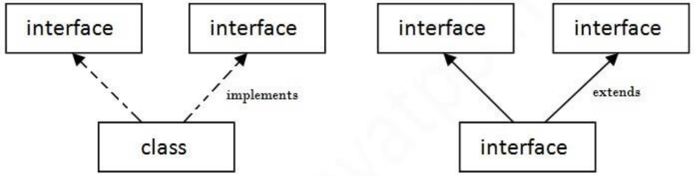
Class which implements interface must provide logic in every method body.

What is implements keyword?

Curious Cats

- Can we achieve multiple inheritance in Java?
  - Not possible with classes.
  - But,



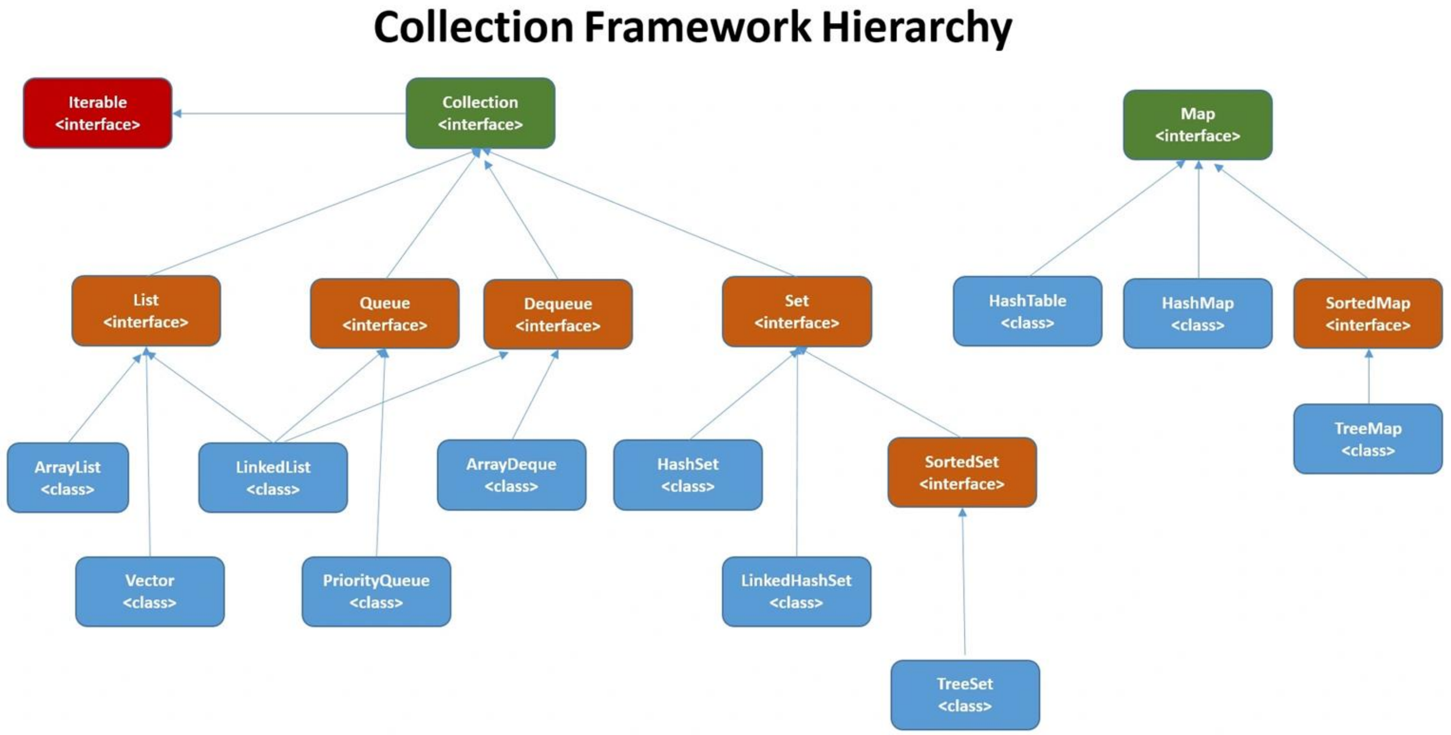


Multiple Inheritance in Java

Activity 3.1 - Payment Option

- Improved Solution - Payment Option using Interfaces
    - Can we accommodate as many payment methods as possible easily?
    - What did we achieve here?
        - Total abstraction of Payment Options from CreditCard class.
        - Loose coupling using interface.

Application of Interfaces in Java Collection



Summary - Interfaces

- An interface defines a contract which specifies what a class **must do** and **not how.**
- An interface declaration contains
  - **Signatures, but no implementations** for methods
  - May also contain constants
- A class that **implements an interface** must implement all the **methods declared in the interface**.
- You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.
- **Multiple inheritance** can be achieved using *interfaces.*

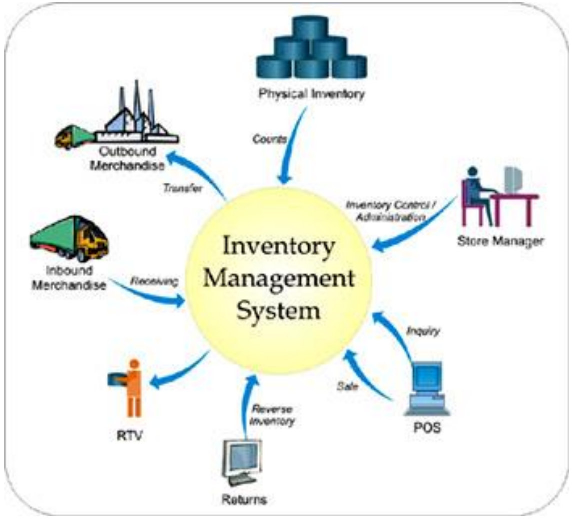
Curious Cats

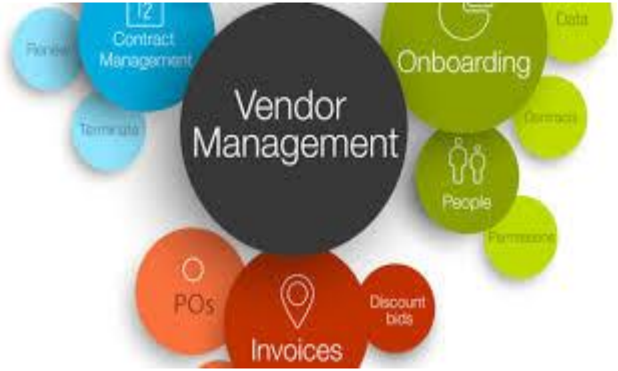
- How is an interface different from an abstract class? When to use which?
  - Detailed Reading:- Difference between Abstract Class and Interface in Java - GeeksforGeeks

Abstraction Byte Overview

Messaging Application

Products/Teams in an eCommerce Company





Inventory management refers to the process of ordering, storing, and using a company's inventory. These include products as well as warehousing and processing such items.

Vendor management system allows us to take appropriate measures for controlling cost, reducing potential risks related to vendors, ensuring excellent service deliverability and deriving value from vendors.

Logistics management includes customer service, scheduling, packaging, and delivery. It functions across these dimensions - strategic, operational and tactical

Configuration needed for Products for flexibility

Vendor credit limit
Vendor minimum rating
Maximum vendors per product

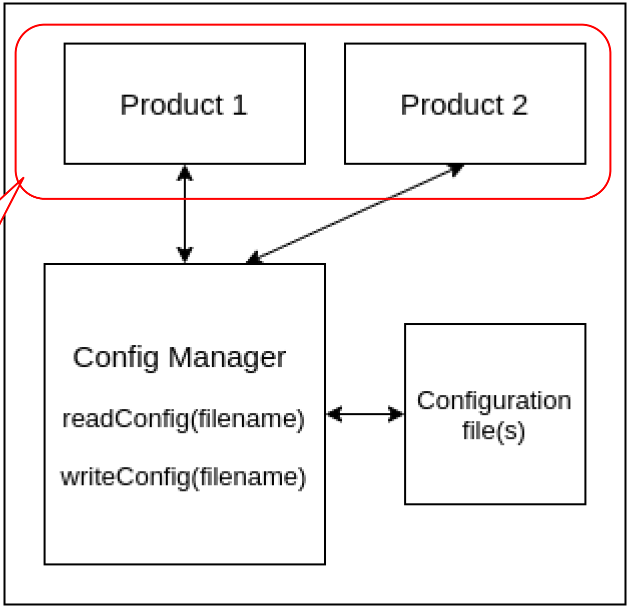
```
{
  "credit_limit": 100000,
  "min_rating": 2,
  "max_vendors_per_product": 10
}
```

Current allowable discount
Delivery vendor list
Warehouse list

Storage capacity
Maximum pending orders
Default re-order timer

Vendor Management
Logistics Management
Inventory Management

Product(s) can read or write their configuration from/to a file using the Config Manager



New Requirements

Config Manager initially only supports JSON file format. There is a new requirement to support XML File Format.

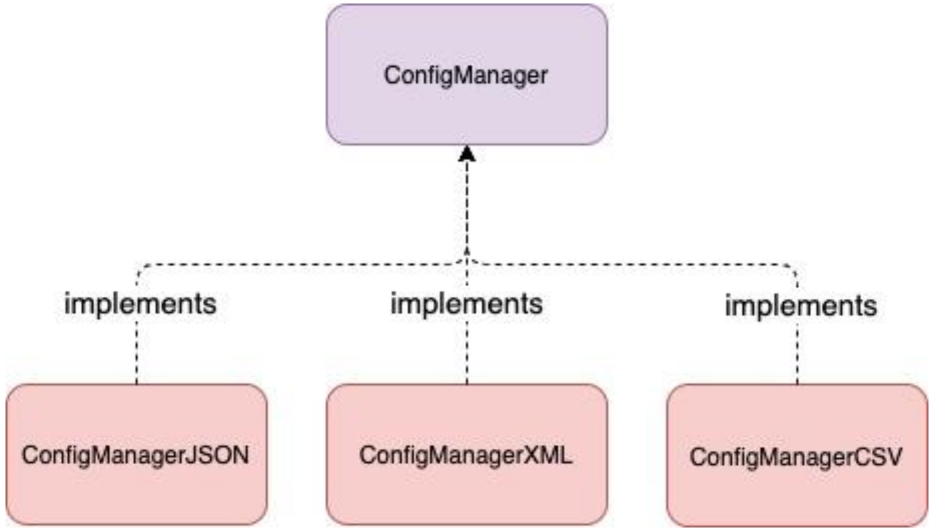
Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <credit_limit>100000</credit_limit>
            <min_rating>100</min_rating>
            <max_vendors_per_product>3306</max_vendors_per_product>
</configuration>
```

Possible Solution

- You have currently two methods in ConfigManager.java
  - readConfig
  - writeConfig
- Add two new methods to support XML file format
  - readConfigXML
  - writeConfigXML
- Rename the previous methods supporting JSON file format
  - readConfigJSON
  - writeConfigJSON
- What’s the issue with the above approach?
  - Adding support for another file type will involve creating even more methods.
    - It no longer serves a single purpose.
  - The Clients using the current methods in multiple places will be affected.
    - Not extensible code

Abstraction Based Solution



Why is this solution better?

- Define common interface which is less likely to change
- Client is unaffected by the internal implementations
- Support multiple implementations for same behaviour
- Can switch between implementations easily

Take home exercises for the session

- You will explore **Abstraction** with this real world scenario in the following Byte:
  - Abstraction Byte - Crio.do
- Complete the Quiz Activity (details on the google site)
  - Java 2 Session 5 Quiz.

These details are also available on the site.

Further Reading

- Java Abstract Keyword - Javatpoint
- Oracle docs - Abstract
- Oracle docs - Interface
- Abstract Classes and Abstract Methods in Java - Dot Net Tutorials
- Abstraction in Java | Abstract Class, Example - Scientech Easy
- OOP Case Study I/O
- Lab Exercise: Abstraction and Encapsulation

Feedback

Thank you for joining in today.

We’d love to hear your thoughts and feedback - Feedback for JAVA-2 Session

Thank you