Object-Oriented Specification & Design - 5CCS2OSD

Investment Analysis System - Team Project Report

Team Members

Andrei Niculescu-Duvaz - 1533089
Jeong Ho Suh - 1533264
Aakash Ranade - 1533916
Jakub Jeziorski - 1534686
Euron Beqiri - 1535107

Introduction

The task at hand was to develop and implement a system that would give an 'investor' the opportunity to purchase a variety of different bonds. The bonds may differ in the term length, as well as the percentage return in the form of 'coupons'. The system should be able to calculate returns throughout the term length as well as being able to calculate the final payment to an investor.

The system would be designed and implemented gradually, in three stages:

1. Requirements analysis + initial specification
    - Outline ambiguous requirements, assumptions, use case diagram and class diagram
2. Design
    - Architecture diagram and detailed explanation of operations
3. Implementation and testing
    - Testing the system and adjusting it if output does not match expected output

Organisation and Plan of Action

Jakub first contacted us via email and gathered all of our phone numbers, and then proceeded to add us to a WhatsApp group chat. Here we arranged our initial meeting, in which we all discussed our individual strengths and weaknesses and what we felt we could contribute to the project, and then

tasks were delegated between team members. Each member was assigned a task and we made it clear that if anyone was to struggle with their designated task, that they should ask someone on the team for help in moving forward towards the completion of the task.

During the initial meeting we decided that Jeongho Suh would be the group leader, and Aakash created a GitHub repository for our work. We also created an online word document which made it possible for any of the team members to access and control the content of the report as we went along, making it extremely convenient for us to adjust the document if need be. We looked closely at the requirements and at the assumptions that we were going to have to make due to any ambiguous or incomplete requirements, and these were then listed and posted in the WhatsApp group chat. We then discussed possible ideas of how to design our program and came up with a very basic class diagram just to put our ideas on paper and give us an indication of what we were looking to achieve.

Our meetings were as frequent as it was possible (at least once a week), so that everyone was kept up to date with the latest development of our program and report. During our project we communicated with each other through different online mediums such as WhatsApp, Word Online, a Facebook group and Skype video calls. We used the Issues and Project tabs on GitHub to create our to-do list, however we tried to fulfil the potential of Word Online, as it allowed us to comment on each other's work.

Primary member roles (as well as assisting other members with their respective tasks):

- Jeongho: *Team leader, Main coder, Test cases*
- Aakash: *Pseudo-code, Creating Github repository*
- Jakub: *Class diagram, Use case diagram, Organised meetings*
- Andrei: *Architecture diagram, Assumptions*
- Euron: *Use case diagram, Formulation of final report*

Requirements analysis and Assumptions

According to Karl Wieger's Software Requirements, a stakeholder is '*a person, group, or organization that is actively involved in a project, is affected by its outcome, or can influence its outcome.*' In the case of our system, the bank can

be considered a stakeholder, as well as the investor, and even we ourselves as the developers can be considered stakeholders as we are actively involved in the project. The main stakeholder in our system is the *investor*, interacting with a variety of different functions to allow them to purchase bonds and establish a portfolio, while calculating returns based on differing interest rates.

In order to create a specification, we had to make a number of different assumptions, due to there being ambiguous requirements that left a lot of room for misconception. These assumptions were made in order for the stakeholders to all understand what the requirements were, and to make it possible to come up with a concise specification for the program. These are the assumptions that were made:

- Bonds are all purchased from the same centralised system.
- Since availability is not stated, we have worked under the assumption that an unlimited. number of investors may purchase the same bond type.
- There is no upper limit to the amount of money that can be invested for a given bond.
- The rate of inflation is separate for each bond type.
- The inflation is subject to change, in this case, the inflation value changes annually.
- Use cases to add and remove Bond types from the system (and from the investors which have them).
- A non-functional requirement which has not been stated is whether a bond's term is fixed or whether it can be terminated during the course of the agreement. We have made the assumption that the contract is fixed and therefore it cannot be terminated by the investor (or the system) until the maturity date stated in the agreement.
- The InvestmentAnalysisSystem cannot exist without any investors, and so it has a dependency on investors.
- A single bond will always have a price of £100.
- The investor knows the inflation rate.
- The frequency per bond annually is 1.

Use Cases

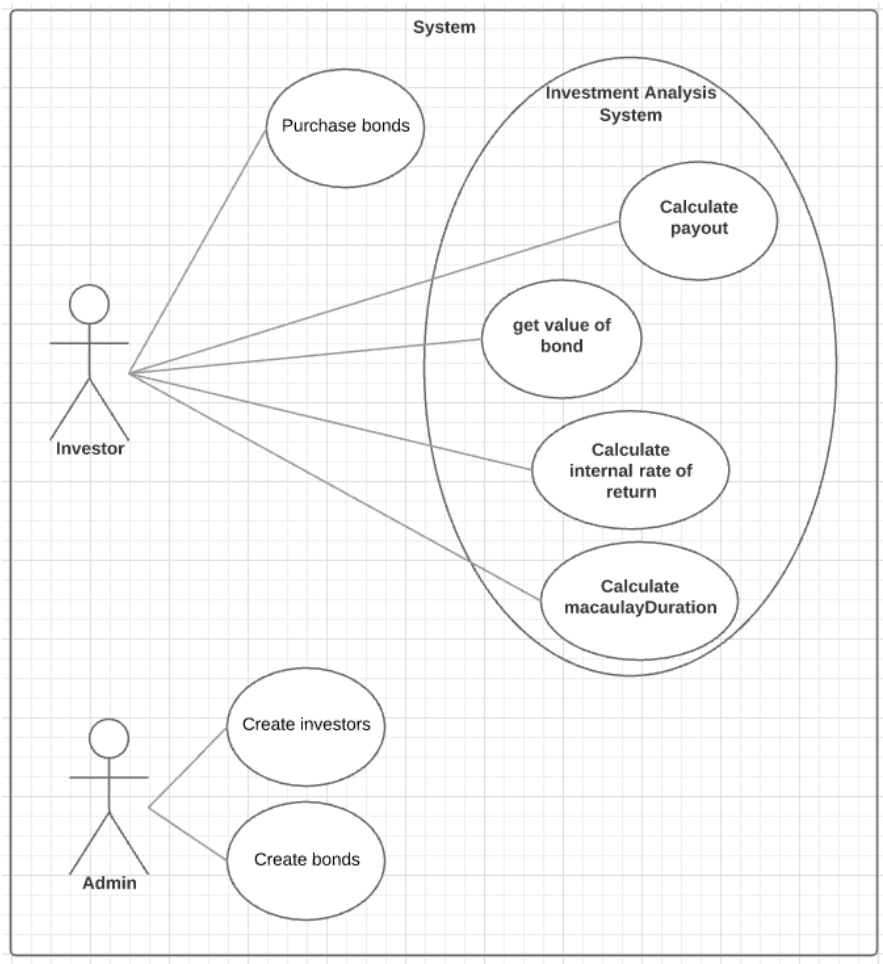


Figure 1 – Use Case Diagram

Figure 1 shows a possible CMS use case diagram model. Here we assume investor is interacting with the investment analysis system. The creation of bonds is decided by investor purchasing operation, which will be then calculated and stored in investor's portfolio. There is an existence of association in readPortoflio operation to several calculation operations provided by dotted lines. Investor is able to view the existing bonds that he or she purchased from previous purchased operation, this will display all the information of calculated values that investor wish to make investments. Turning to details, for the operations of value and Macaulay duration will need to have input parameter of r, which it can be varied by investor's preference of their investing the bonds. In addition, irr is an operation that finds the internal rate of return. In summary, user can view their values by using single readPortofolio operation.

Class Diagram

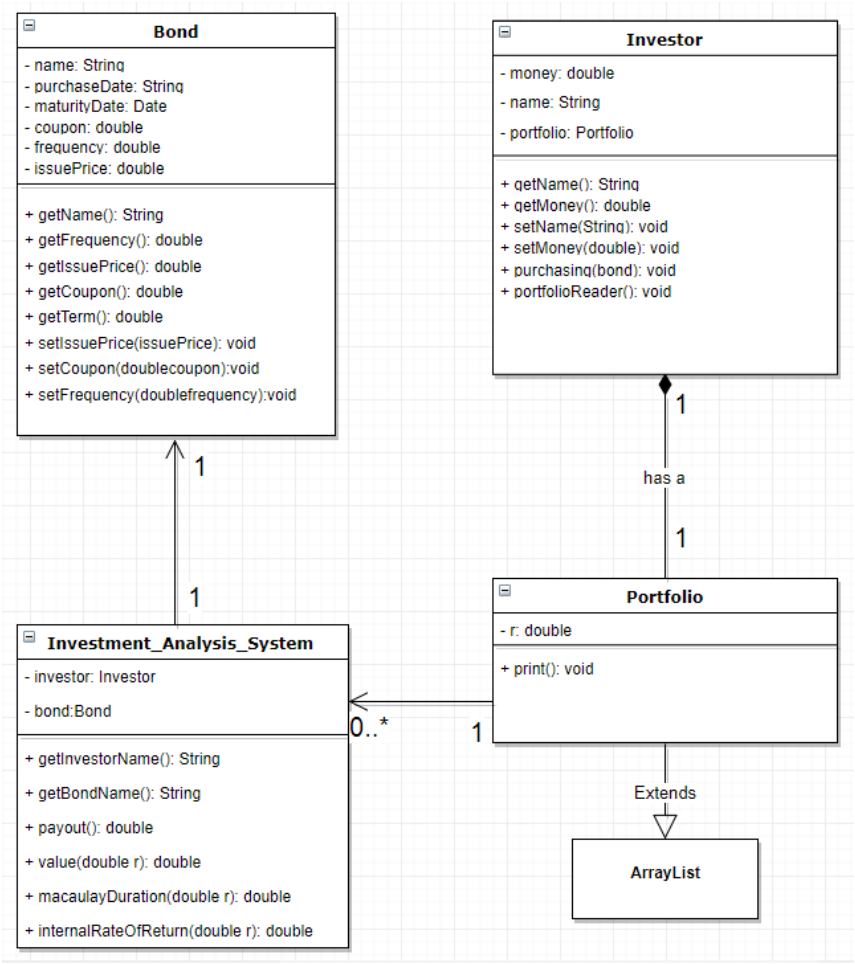


Figure 2 – Class Diagram

We created our class diagram with the use of draw.io, an online tool. As can be seen in figure 2, we have four classes, where the **Portfolio** class extends **ArrayList**. The investor class has a *portfolio* as a private variable, and due to the

assumption that the **Portfolio** class cannot exist on its own, they have a one to one relation (composition).

In class **Portfolio** we can store zero or more **Investment_Analysis_System** classes, hence the use of one to many relations.

Each **Investment_Analysis_System** holds a reference to a purchased **Bond**, and again this is an example of a one to one relation. In our implementation due to the assumption that we made that bonds can exists on their own, same as **Investor**, but bond is added to an *investor's portfolio* only once it is purchased.

Architecture Diagram

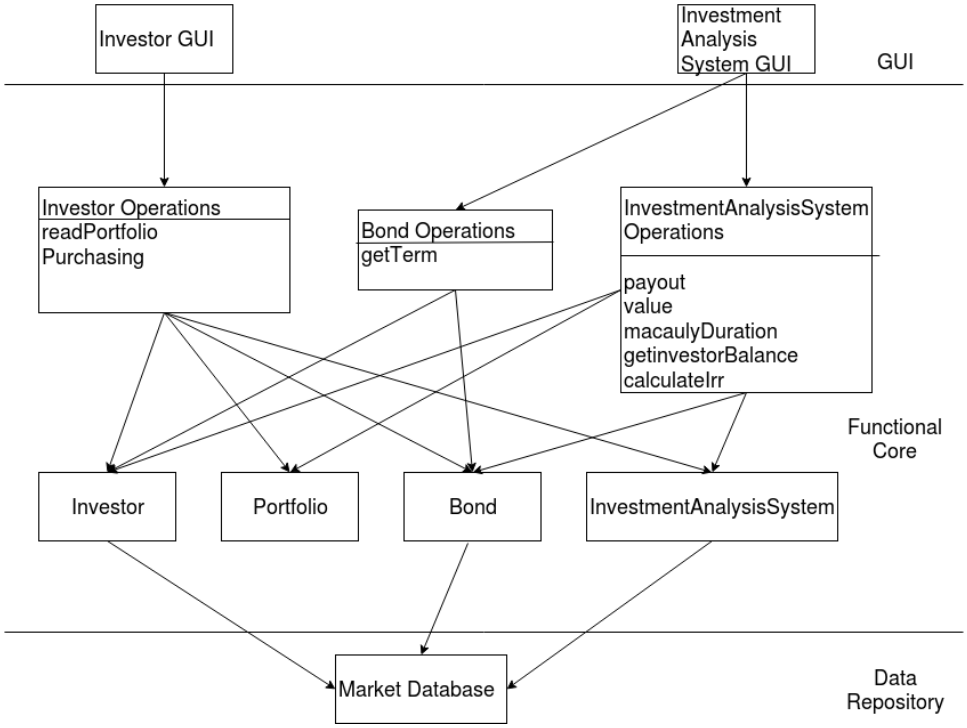


Figure 3 – Architecture Diagram

Figure 3 lays out the basic architecture of what our program would look like if we fully implemented this as a real-world program.

The highest level is the GUI level with which the users actually interact with to carry out the use cases (i.e. the public methods). This implements functionality from components in the functional core by accessing their public methods. In this case, the investor GUI invokes the operations from investor, and the Investment Analysis System GUI draws its functionality by accessing the public methods in the InvestmentAnalysisSystem and Bond classes. In our case these GUI classes were implemented in the command line.

For the Investor Operations, this functionality is invoked from the Investor, Portfolio and Bond classes. In the case of the investmentAnalysisSystem Operations, they are invoked using the Bond and Investor classes. The Bond operations draws its functionality from invoking the methods in the Bond and Investor classes. To represent this, there are arrows pointing from the operations to the constituent classes from which it draws functionality. The Investor and Bond classes may be required to access data from the Database in order to implement their operations properly. Arrows have again been used to represent that these classes may refer to this data.

Pseudocode

Pseudocode is used as a detailed step in the process of developing a program. It allowed us (as developers) to express the design of the program in detail and also allowed us to create a template as a stepping stone into actually writing the code in Java. Once this template was created, it allowed us to analyse it and ensure that once the final program is compiled, that it would match the design specifications.

Bond

Initiate public class ***Bond***

Declare private String *name*
Declare private String *purchaseDate*
Declare private String *maturityDate*
Declare private double *coupon*
Declare private double *frequency*
Declare private double *issuePrice*

Initiate public ***Bond*** constructor; parameters: String *name*, *purchaseDate, maturityDate;* double *coupon, frequency, issuePrice*
Assign the received parameters to the field variables appropriately

Create ***get*** methods for *name, frequency, term* and *issuePrice*

Create ***set*** methods for *coupon, frequency* and *issuePrice*

Portfolio

Initiate public class ***Portfolio*** extends **ArrayList** of **InvestmentAnalysisSystem**

Declare private double *r*

Initiate public constructor ***Portfolio;*** parameters: double *r*
Assign the received *r* to the field *r*

Initiate public method ***print;*** void return
New *for* loop; *(int i = 0; i < this.size(); i++)*
Print all the values from IAS

Investor

Initiate public class ***Investor***

Declare private String *name*
Declare private double *balance*
Declare private Portfolio *portfolio*
Declare private Integer *numberOfBonds*

Initiate public constructor ***Investor;*** parameters: String *name,* integer *balance,* double *r*

Assign the received parameters to the field variables appropriately and initialise *portfolion* by passing *r*

Initiate public method ***purchasing*;** void return; parameters: Bond… *bonds*

New *for-each* loop; *(Bond bond: bonds)*

portfolio.add(new InvestmentAnalysisSystem(this, bond))

Initiate public method ***portfolioReader*;** void return

portfolio.print

Create ***get*** methods for *name* and *money*

Create ***set*** methods for *name* and *money*

InvestmentAnalysisSystem

Initiate public class ***InvestmentAnalysisSystem***

Declare private **Bond** *bond*

Declare private **Investor** *investor*

Declare private **Portfolio** *portfolio*

Initiate public constructor **InvestmentAnalysisSystem;** parameters: Investor *investor,* Bond *bond*

Assign the received *investor* and *bond* to the field *investor* and *bond*

Initiate public method ***payout;*** double return

double *payout = term * coupon + investormoney *frequency*

Return *payout*

Initiate public method ***value*;** double return

Declare and initiate a local double *val* at *0*

Declare and initiate a local double *rep = 100 / Math.pow((1+r) , term);*

New *for* loop; *(int i = 0; i < term; i++)*

val += coupon /Math.pow((1+r), i);

val += rep;

Return *val*

Initiate public method ***macaulayDuration;*** double return

Declare and initiate a local double *val* at *0*

Declare and initiate a local double *MacD* at *0*

New *for* loop; *(int i = 1; i < term; i++)*

if *(i != term())*

val += coupon / Math.pow((1 + r), i);

*MacD += coupon / Math.pow((1 + r), i) * i;*

Else

val += (coupon + issuePrice) / Math.pow((1 + r), i);

*MacD += (coupon+ issuePrice) / Math.pow((1 + r), i) * i;*

MacD /= val;

Return *MacD*

Initiate public method ***calculateIrr;*** double return
Declare and initiate a local double *r* at *0*

New *for* loop; *(int i = 1; i < term; i++)*
if *(value(1 + r) > bond.getIssuePrice())*

r += 1;

else if (value(0.1 + r) > bond.getIssuePrice())

r += 0.1;

else if (value(0.01 + r) > bond.getIssuePrice())

r += 0.01;

else if (value(0.001 + r) > bond.getIssuePrice())

r += 0.001;

else if (value(0.0001 + r) > bond.getIssuePrice())

r += 0.0001;

else break

Create ***get*** methods for *name* and *money*

Figure 4 - Pseudocode

Pseudocode operation

```
query payout() : double
    pre: true
    post:   result = payments->collect(amount)->sum()

query value(r : double)
    pre: true
    post:   result = payments->collect(amount)->sum()

query macaulayDuration(r : double)
    pre: true
    activity: for p : payments do result:= result +p.tamount(r) ; return result/value(r)

query calculateIrr()
    pre: true
    post: v= value(r) & ((v(r : 0.1 +r) =>result =
```

Test Cases

The first major difficulty that we encountered was the problems that kept arising while we were implementing the math formulas.

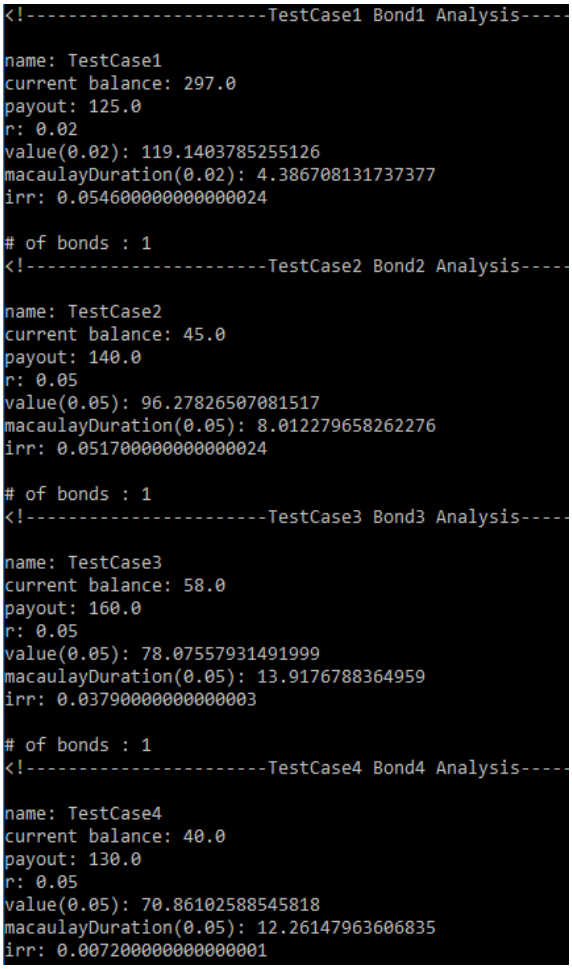
After the initial implementation of the Macaulay Duration and the Internal Rate of Return formulas, we used the test cases provided in figure 5 in order to make sure the output was as expected. However, the same problem kept occurring, and the true output was not what we were expecting the output to be. A lot of our time was focused on getting the correct implementation of these formulas, which was done by formulating the algorithm using pen and paper, and once we had successfully passed the test cases that we were provided with for both the 2% and 5% interest rate (As seen below in figure 5), we decided to create our own TestCaseGenerator. This allowed us to input test values into the program other than the test cases we were initially provided with, and to see if the output was as expected. Figure 5 shows an example generated by our TestCaseGenerator.

1) term = 5, coupon = 5, price = 103
payout = 125
value(0.05) = 99.999
value(0.02) = 114.14
macaulayDuration(0.05) = 4.54
macaulayDuration(0.02) = 4.578
irr = 0.0432

2) term = 10, coupon = 4, price = 95
payout = 140
value(0.05) = 92.278
value(0.02) = 117.965
macaulayDuration(0.05) = 8.359
macaulayDuration(0.02) = 8.579
irr = 0.0463

3) term = 20, coupon = 3, price = 92
payout = 160
value(0.05) = 75.075
value(0.02) = 116.35
macaulayDuration(0.05) = 14.47
macaulayDuration(0.02) = 15.71
irr = 0.0356

4) term = 15, coupon = 2, price = 120
payout = 130
value(0.05) = 68.86
value(0.02) = 99.99
macaulayDuration(0.05) = 12.61
macaulayDuration(0.02) = 13.1
irr = 0.006



```
<!------ result: test1 ------>

term = 5.0, coupon = 5.0, price = 100.0
payout: 125.0
value(0.05): 99.99999999999999
value(0.02): 114.1403785255126
macaulayDuration(0.05): 4.545950504162359
macaulayDuration(0.02): 4.5788709836748644
irr(): 0.049900000000000035

<!------ result: test2 ------>

term = 10.0, coupon = 4.0, price = 95.0
payout: 140.0
value(0.05): 92.27826507081517
value(0.02): 117.96517001248445
macaulayDuration(0.05): 8.359589164010508
macaulayDuration(0.02): 8.579642968268928
irr(): 0.046300000000000015

<!------ result: test3 ------>

term = 20.0, coupon = 3.0, price = 92.0
payout: 160.0
value(0.05): 75.07557931491999
value(0.02): 116.35143334459708
macaulayDuration(0.05): 14.473825547456943
macaulayDuration(0.02): 15.717880872158787
irr(): 0.03560000000000002

<!------ result: test4 ------>

term = 15.0, coupon = 2.0, price = 120.0
payout: 130.0
value(0.05): 68.86102588545819
value(0.02): 99.99999999999997
macaulayDuration(0.05): 12.617602115465148
macaulayDuration(0.02): 13.106248770585527
irr(): 0.006
```

Figure 5 – Test Cases

```
<!----------------------Jeongho Bond1 Analysis---------------------->

name: Jeongho
current balance: 297.0
payout: 125.0
r: 0.05
value(0.05): 99.99999999999999
macaulayDuration(0.05): 4.545950504162359
irr: 0.04320000000000001

# of bonds : 1
<!----------------------Jeongho Bond2 Analysis---------------------->

name: Jeongho
current balance: 202.0
payout: 140.0
r: 0.05
value(0.05): 92.27826507081517
macaulayDuration(0.05): 8.359589164010508
irr: 0.046300000000000015

# of bonds : 2
<!----------------------Jeongho Bond3 Analysis---------------------->

name: Jeongho
current balance: 110.0
payout: 160.0
r: 0.05
value(0.05): 75.07557931491999
macaulayDuration(0.05): 14.473825547456943
irr: 0.03560000000000002

# of bonds : 3
+-------------------------------------------------------------------------------+
| Bond4 cannot be purchased of your account balance is too low                  |
+-------------------------------------------------------------------------------+
<!----------------------Andrew Bond1 Analysis---------------------->

name: Andrew
current balance: 37.0
payout: 125.0
r: 0.05
value(0.05): 99.99999999999999
macaulayDuration(0.05): 4.545950504162359
irr: 0.04320000000000001

# of bonds : 1
<!----------------------Christian Bond1 Analysis---------------------->

name: Christian
current balance: 47.0
payout: 125.0
r: 0.05
value(0.05): 99.99999999999999
macaulayDuration(0.05): 4.545950504162359
irr: 0.04320000000000001
```

Bond Class

```
import java.text.DecimalFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Bond {

    //<!-- variables-->

    private String name;

    private String purchaseDate; //date

    private String maturityDate;

    private double coupon;

    private double frequency;

    private double issuePrice;

    //<!-- variables-->




    //<!---constructor--->


    public Bond(String name, String purchaseDate, String maturityDate, double coupon, double issuePrice, double
frequency) {

        this.name = name;
        this.purchaseDate = purchaseDate;
        this.maturityDate = maturityDate;
        this.coupon = coupon;
        this.issuePrice = issuePrice;
        this.frequency = frequency;

    }

    public Bond() {
    }// default constructor

    //<!---constructor--->


    //<!---getter--->

    public String getName() { return name; }

    public double getFrequency() {
```

```
    return frequency;
}

public double getIssuePrice() {
    return issuePrice;
}

public double getCoupon() {
    return coupon;
} // bond must be stored in decimal (e.g. 50%  -> 0.5)

public double getTerm() {

    /*logic*/

    // maturityDate-purchaseDate

    try {
        Date start;
        Date end;


        SimpleDateFormat dates = new SimpleDateFormat("yyyy/MM/dd");


        start = dates.parse(this.purchaseDate);
        end = dates.parse(this.maturityDate);

        double difference = Math.abs(start.getTime() - end.getTime());
        double differenceByYear = (difference / (24 * 60 * 60 * 1000)) / 365;

        DecimalFormat decf = new DecimalFormat("##.0");

        return Double.parseDouble(decf.format(differenceByYear));

    } catch (ParseException e) {

        return (double) e.getErrorOffset();
    }


}

//<!---getter--->

//<!---setter--->

public void setIssuePrice(double issuePrice) { this.issuePrice = issuePrice; }

public void setCoupon(double coupon) { this.coupon = coupon; }

public void setFrequency(double frequency) { this.frequency = frequency; }

//<!---setter--->
```

}

InvestmentAnalysisSystem Class

```
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Scanner;

public class InvestmentAnalysisSystem {

/*
   Deadline: November 30th, 4pm.
*/
    //<!-- variables--->

    private Bond bond;

    private Investor investor;

    private Portfolio portfolio;

    //<!-- variables--->



    public InvestmentAnalysisSystem(Investor investor, Bond bond) {/*default constructor*/

        this.investor = investor;
        this.bond = bond;


    }


    public String getInvestorName() {

        return investor.getName();

    }

    public String getBondName() {

        return bond.getName();
    }

    public int getBondPurchasedNumber() {

        return investor.getNumberOfBonds();
    }

    public double getBondIssuePrice(){

        return bond.getIssuePrice();
    }


    //<!---functions--->
```

```
public double getInvestorBalance(){

    return investor.getBalance();

}

public double payout() {

    double payout = bond.getTerm() * bond.getCoupon() + (100 * bond.getFrequency());
    return payout;
}

public double value(double r) {

    double val = 0;

    for (int i = 1; i <= bond.getTerm(); i++) {

        val += bond.getCoupon() / Math.pow((1 + r), i);
    }
    double actualVal = 100 / Math.pow((1 + r), bond.getTerm());
    val += actualVal;

    return val;

}
public double macaulayDuration(double r) {

    double val1 = 0;
    double val2 = 0;
    for (int i = 1; i <= bond.getTerm(); i++) {
        val1 += i * bond.getCoupon() / Math.pow((1 + r), i);
```

```
    }

    val2 = bond.getTerm() * 100 / Math.pow((1 + r), bond.getTerm());

    double val3 = val1 + val2;

    double MacD = val3 / value(r);

    return MacD;

 /*   double val = 0;
    double MacD = 0;

    for (int i = 1; i <= bond.getTerm(); i++) {
        if (i != bond.getTerm()) {

            val += bond.getCoupon() / Math.pow((1 + r), i);
            MacD += bond.getCoupon() / Math.pow((1 + r), i) * i;

        } else {

            val += (bond.getCoupon() + investor.getMoney()) / Math.pow((1 + r), i);
            MacD += (bond.getCoupon() + investor.getMoney()) / Math.pow((1 + r), i) * i;

        }
    }

    MacD /= val;

    return MacD;
*/

}

public double calculateIrr() {

    double r = 0;

    for( ;bond.getIssuePrice() != value(r); )
    {

       if (value(1 + r) > bond.getIssuePrice()) r += 1;
```

```
        else if (value(0.1 + r) > bond.getIssuePrice()) r += 0.1;
        else if (value(0.01 + r) > bond.getIssuePrice()) r += 0.01;
        else if (value(0.001 + r) > bond.getIssuePrice()) r += 0.001;
        else if (value(0.0001 + r) > bond.getIssuePrice()) r += 0.0001;
        else break;


      }


      return r;

   }


}


//<!---functions--->
```

Portfolio Class

```
import java.util.ArrayList;

public class Investor {

    //<!-- variables-->


    private String name;

    private double balance; //balance

    private Portfolio portfolio;

    private int numberOfBonds;


    //<!-- variables-->


    //<!---constructor--->

    public Investor(String name, int balance, double r) {
        this.name = name;
        this.balance = balance;
        portfolio = new Portfolio(r);
    }


    public Investor() {
    } //default constructor

    //<!---constructor--->


    //<!---getter--->


    public int getNumberOfBonds() {
        return numberOfBonds;
    }

    public String getName() {
        return name;
    }


    public double getBalance() {
        return balance;
    }


    //<!---getter--->
```

```
//<!---setter--->


public void setName(String name) {
   this.name = name;
}


public void setBalance(double balance) {
   this.balance = balance;
}


//<!---setter--->


//<!---functions--->

public void purchasing(Bond... bonds) {

   /*logic*/


   for (Bond bond : bonds) {

      numberOfBonds++;

      if(balance > bond.getIssuePrice()) {

         balance -= bond.getIssuePrice();

         portfolio.add(new InvestmentAnalysisSystem(this, bond));

         portfolio.print(numberOfBonds-1);



      }else{



         System.out.println("+-----------------------------------------------------------------------+ ");
         System.out.println("| "+bond.getName()+" cannot be purchased of your account balance is too low          | ");
         System.out.println("+-----------------------------------------------------------------------+ ");

      }


   }
}
```

```
    //new InvestmentAnalysisSystem(this, bond );
    public void analysePortfolio(){


    }

    //<!---setter--->

}
```

Main Class

```
public class Main {

    public static void main(String[] args) {



        Investor investor1 = new Investor("Jeongho", 400, 0.05 );
        Investor investor2 = new Investor("Andrew", 140, 0.05);
        Investor investor3 = new Investor("Christian", 150, 0.05);
        Investor investor4 = new Investor("Kevin", 160, 0.05);



        Bond bond1 =  new Bond("Bond1", "2017/11/11", "2022/11/11", 5, 100,1);
        Bond bond2 =  new Bond("Bond2", "2017/11/11", "2027/11/11", 4, 95,1);
        Bond bond3 =  new Bond("Bond3", "2017/11/11", "2037/11/11", 3, 92,1);
        Bond bond4 =  new Bond("Bond4", "2017/11/11", "2032/11/11", 2, 120,1);



        investor1.purchasing(bond1, bond2, bond3, bond4);

        investor1.analysePortfolio();


        investor2.purchasing(bond1);

        //investor2.readPortfolio();

        investor3.purchasing(bond1, bond2);

        //investor3.readPortfolio();









    }


}
```

```
import java.util.ArrayList;

public class Portfolio extends ArrayList<InvestmentAnalysisSystem> {

    //investor wants to check the actual value of a bond under different inflation rate

    double r;

    public Portfolio(double r) {

        this.r = r;


    }



    public void print(int i) {


            System.out.println("<!----------------------" + this.get(i).getInvestorName() + " " + this.get(i).getBondName() + "
Analysis---------------------->\n");


            System.out.println(
                        "name: " + this.get(i).getInvestorName() + "\n" +
                        "current balance: " + this.get(i).getInvestorBalance() + "\n" +
                        "payout: " + this.get(i).payout() + "\n" +
                        "r: " + r + "\n" +
                        "value(" + r + "): " + this.get(i).value(r) + "\n" +
                        "macaulayDuration(" + r + "): " + this.get(i).macaulayDuration(r) + "\n"+
                        "irr: " + this.get(i).calculateIrr() + "\n"
            );

            System.out.println("# of bonds : "+this.get(i).getBondPurchasedNumber());


    }
}
```