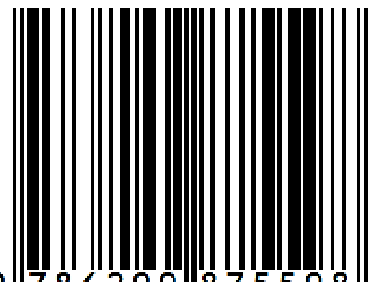
MERGING LANES: WHERE E-LEARNING DIVERSITY MEETS FUTURE TRENDS

VOLUME 11, 2026
e-ISBN : 978-629-98755-9-8



SIG CS@e-Learning
Unit Penerbitan
Jabatan Sains Komputer & Matematik
Kolej Pengajian Pengkomputeran, Informatik & Matematik
Universiti Teknologi MARA Cawangan Pulau Pinang

MERGING LANES: WHERE E-LEARNING DIVERSITY MEETS FUTURE TRENDS

Copyright@2026 by Unit Penerbitan Jabatan Sains Komputer & Matematik (JSKM), Universiti Teknologi MARA Cawangan Pulau Pinang, 13500 Permatang Pauh, Pulau Pinang, Malaysia

All rights reserved. No parts of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying or otherwise, without the prior written permission in writing from Authors of Department of Computer & Mathematical Sciences, Academic Affairs Section, Universiti Teknologi MARA Cawangan Pulau Pinang, 13500 Permatang Pauh, Pulau Pinang, Malaysia.

Advisor
Dr. Nor Hanim Abd Rahman,
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

Chief Editor
Ts. Jamal Othman,
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

Editors
Ts. Dr. Syarifah Adilah Mohamed Yusoff,
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

Dr Arifah Fasha Rosmani,
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

Mohd Saifulnizam Abu Bakar,
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

Published by:
Unit Penerbitan Jabatan Sains Komputer & Matematik (JSKM)
Bahagian Hal Ehwal Akademik (BHEA)
Universiti Teknologi MARA
Cawangan Pulau Pinang
13500 Permatang Pauh
Pulau Pinang
Malaysia

e ISBN : 978-629-98755-9-8

e-ISBN : 978-629-98755-9-8

AGGREGATION IN OBJECT-ORIENTED PROGRAMMING: A SPECIALIZED FORM OF ASSOCIATION

Syarifah Adilah Mohamed Yusoff[1], *Jamal Othman[2], Mohd Saifulnizam Abu Bakar[3] and Arifah Fasha Rosmani[4]

*syarifah.adilah@uitm.edu.my[1], *jamalothman@uitm.edu.my[2], mohdsaiful071@uitm.edu.my[3], arifah840@usm.edu.my[4]*

[1,2,3,4]Jabatan Sains Komputer & Matematik (JSKM),
Universiti Teknologi MARA Cawangan Pulau Pinang, Malaysia

**Corresponding author*

ABSTRACT

Object-Oriented Programming (OOP) is one of the fundamental paradigms in modern software development. It offers a range of powerful features that make it highly suitable for addressing real-world problems. Among these features are code reusability, modularity, data encapsulation, enhanced security, and improved code organization, all of which contribute to the development of efficient, maintainable, and scalable software systems. OOP is structured around several core principles, including the use of classes and objects, inheritance, polymorphism, abstraction, and encapsulation (also referred to as information hiding). These principles collectively enable developers to model complex systems in a way that mirrors real-life entities and their interactions. A significant aspect of object-oriented design is the concept of association, which describes relationships between classes. Within this context, aggregation is recognized as a specialized form of association, often contrasted with composition. Both aggregation and composition are referred to as "has-a" relationships, indicating that one class contains references to another. However, aggregation represents a weaker relationship compared to composition. In aggregation, the contained (or referenced) object can exist independently of the containing object. That is, if the container object is destroyed, the contained object is not necessarily affected and may continue to exist on its own. This article aims to provide a brief yet informative explanation of aggregation within the scope of object-oriented programming. It will explore the concept in greater detail and illustrate its implementation through practical examples. In particular, sample code written in the Java programming language will be presented to demonstrate how aggregation can be applied.

Keywords: Object-oriented programing (OOP), association, aggregation, Unified Modeling Language (UML), Java

Introduction

Computer programming often presents significant challenges, particularly as the code grows in size and complexity due to the inclusion of multiple functions or subprograms. The selection of appropriate tools and techniques is essential to guide software developers in completing projects within the specified timeframe and allocated budget (Otu et al., 2023). Object-Oriented Programming (OOP), which is structured around objects defined by standard classes, allows for dynamic modification and interaction among objects through message passing. OOP facilitates faster application development, simplifies code modification, promotes code reusability, and enhances the overall comprehensibility of the program structure (Asagba & Ogheneovo, 2010). In addition to its development advantages, OOP also

provides built-in mechanisms to strengthen software security. Through encapsulation, OOP conceals the internal implementation details of objects by restricting direct access to their attributes and methods. Only designated interfaces are exposed to the outside world, reducing the risk of unintended interference or misuse by other parts of the program (Vincent & Afolorunso, 2020).

Modularity is a fundamental principle in software engineering that involves decomposing a large and complex program into smaller, manageable, and self-contained units or modules. These smaller units are designed to function independently but can be seamlessly integrated to achieve the objectives of the overall system. This approach not only enhances the structural organization of the software but also improves the readability, maintainability, and scalability of the code. Modular programming contributes significantly to the accuracy and clarity of program logic and plays a vital role in simplifying future modifications, debugging, and updates (Hutabarat et al., 2009).

Within the paradigm of Object-Oriented Programming (OOP), one common design technique that reflects modular principles is aggregation, a form of association that establishes a "has-a" relationship between two classes. Both aggregation and composition fall under the broader category of association relationships, which describe how one class incorporates or is composed of instances of another class. However, these two associations differ in terms of the strength of their coupling and the lifecycle dependencies between the related objects. Aggregation is characterized by a loose coupling, where the associated "part" objects can exist independently of the "whole" object. In practical terms, this means that the destruction of the whole object does not necessarily lead to the destruction of its constituent parts. For example, if a "Library" object is deleted, the "Book" objects associated with it can still exist independently. This contrasts with composition, which signifies a strong coupling between classes, where the lifecycle of the part objects is strictly bound to the lifecycle of the whole. In composition, if the whole is destroyed, the parts are destroyed as well (Larman, 2004). This nuanced understanding of class relationships enables developers to model real-world scenarios more accurately and design systems that are both flexible and robust.

This paper is organized to provide a comprehensive examination of the implementation of the aggregation association, specifically focusing on its characteristic of loose coupling, using the Java programming language. Aggregation, as a form of association in object-oriented programming, enables one class to reference another through a "has-a" relationship without establishing strict dependency between their lifecycles. To contextualize this concept, a real-world problem scenario involving two distinct classes will be introduced. This scenario will demonstrate how aggregation is used to model relationships between objects that are logically connected but maintain their independence. For clearer visual representation between classes, the structure of aggregation association will be illustrated through a Unified Modeling Language (UML) class diagram. Following this, the paper will present detailed code implementations for each class, alongside a sample application program. These code examples

will serve to clarify the practical aspects of aggregation and highlight its advantages in promoting modularity, reusability, and maintainability within object-oriented software development.

Methodology

To illustrate the class relationship represented by an aggregation association, a Unified Modeling Language (UML) class diagram will be utilized. Figure 1 presents the UML class diagram that depicts the relationship between the Customer and Item classes. In this context, the Customer class represents the "whole," while the Item class functions as the "part" within the aggregation relationship. The diagram reflects a scenario where an Item—identified by a specific item code—can be associated with multiple Customers, or it may not be associated with any customer at all. This demonstrates that the Item objects can exist independently of the Customer objects, reinforcing the concept of loose coupling that characterizes aggregation in object-oriented programming.

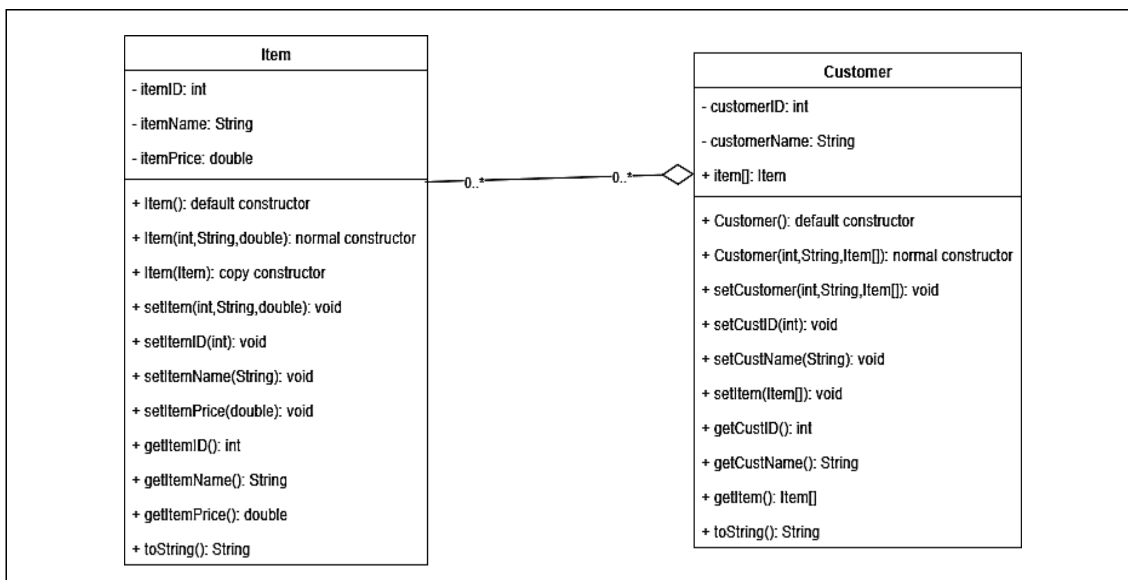


Figure 1: UML Class Diagram – Aggregation Association between Item and Customer classes

Implementation

The Java programming language was utilized to develop the Item and Customer classes. Each class includes standard methods, such as default and parameterized constructors, mutator methods, accessor methods, and printing functions. The following code presents the complete definition of the Item class.

```
public class Item {
    //attributes
    private int itemID;
    private String itemName;
    private double itemPrice;

    //default constructor
```

```
        public Item()
        {       itemID = 0;
                itemName = "";
                itemPrice = 0;      }

        //normal constructor
        public Item(int iID, String iName, double iPrice)
        {       itemID = iID;
                itemName = iName;
                itemPrice = iPrice;       }

        //copy constructor
        public Item(Item i)
        {       itemID = i.itemID;
                itemName = i.itemName;
                itemPrice = i.itemPrice; }

        //group setter
        public void setItem(int iID, String iName, double iPrice)
        {       itemID = iID;
                itemName = iName;
                itemPrice = iPrice;       }

        //individual setter
        public void setItemID(int itemID){this.itemID = itemID;}
        public void setItemName(String itemName){this.itemName = itemName; }
        public void setItemPrice(double price){itemPrice = price;   }

        //getter
        public int getItemID()        { return itemID;   }
        public String getItemName()   { return itemName; }
        public double getItemPrice()  { return itemPrice;   }

        //printer
        public String toString(){
                return "\n Item ID     : "+itemID+
                       "\n Item Name   : "+itemName+
                       "\n Item Price  : RM "+itemPrice;     }

        public void display()
        { System.out.printf("%-10d %-15s %-6.2f%n",
                                         itemID, itemName, itemPrice);     }
}//close class
```

Figure 2: Complete definition for Item class

Next, Figure 3 presents the detailed definition of the Customer class. This class incorporates the Item class as one of its attributes through association. Specifically, the attribute is defined as an array of Item objects, indicating that each customer can purchase multiple items.

```
    public class Customer {
        //attributes
        private int custID;
        private String custName;
        public Item[] item;  //array of object and type is Item(is a class
                             // as created in figure 1): aggregation
```

```
      //default constructor
      public Customer(){
            custID = 0;
            custName = "";
            item = new Item[10]; }//maximum item can be stored is 10 objects
      //normal constructor
      public Customer(int cID, String cName, Item[] itm)
      {     custID = cID;
            custName = cName;
            item = itm;
      }

      //group setter
      public void setCustomer(int cID, String cName, Item[] itm)
      {     custID = cID;
            custName = cName;
            item = itm;
      }

      //individual setter
      public void setCustID(int custID)
      {     this.custID = custID;       }

      public void setCustName(String custName)
      {     this.custName = custName;   }

      public void setItem(Item[] itm) //receives list of items purchased
      {     item = itm;                 }

      //getter
      public int getCustID()        {  return custID;        }
      public String getCustName()   {  return custName;      }
      public Item[] getItem()       {  return item;          }

      //printer
      public void display(int totItemPurchased)
      {
            System.out.print("\n Customer id   :  "+custID);
            System.out.print("\n Customer name : "+custName);
            System.out.print("\n -------------------------------------");
            System.out.print("\nItem ID    Item Name   Item Price (RM)");
            System.out.print("\n");

            double totPurchase = 0;
            for (int x=0;x<totItemPurchased;x++){  //to display all items
                                                    //purchased
                  item[x].display();
                  totPurchase = totPurchase + item[x].getItemPrice();
            }

            System.out.print("\n Total purchase RM "+totPurchase);
            System.out.print("\n");
      }
   }//close class
```

Figure 3: Complete definition for Customer class

The third code segment as shown figure 4, represents the application program that integrates both the Item and Customer classes. This program demonstrates that each customer can purchase a variable number of items by inputting the corresponding item IDs. The array of Customer objects implements aggregation, whereby Item objects are passed as parameters to the Customer constructor during the instantiation process.

```
import java.util.Scanner;

public class custItemApp {
   public static void main(String [] args)
   {
         Scanner scan = new Scanner(System.in);
         String lineSeparator = System.getProperty("line.separator");
         scan.useDelimiter(lineSeparator);

         final int aSIZE = 5;//fixed size to store list of items

         Item[] item = new Item[aSIZE];
         item[0] = new Item (101, "Laptop", 4500);
         item[1] = new Item (102, "Printer", 1200);
         item[2] = new Item (103, "Scanner", 750);
         item[3] = new Item (104, "Pen Drive", 30);
         item[4] = new Item (105, "Head Phone", 120);

         Customer[] cust = new Customer[100];//maximum size is 100
         int cntCust = 0;
         int cont = 1;

         while (cont == 1) {
               System.out.println("\n *************************** ");
               System.out.print("\n Enter the customer ID  : ");
               int custID = scan.nextInt();
               System.out.print("\n Enter customer name : ");
               String custName = scan.next();

               Item[] custBuy = new Item[10];
               System.out.print("\n Enter the item ID to purchase : ");
               int totItemPurchased = scan.nextInt();
               //each customer may purchase any of the item as listed
               System.out.println("   1. Laptop - RM 4500 ");
               System.out.println("   2. Printer - RM 1200 ");
               System.out.println("   3. Scanner - RM 750 ");
               System.out.println("   4. Laptop - RM 30 ");
               System.out.println("   5. Head Phone - RM 120 ");

               for (int y =0;y<totItemPurchased;y++){
                System.out.print("\n  Enter item number("+(y+1)+"):");
                int itemNumber = scan.nextInt();
                custBuy[y] = item[itemNumber-1];
                    //Assume the customer purchase 3 items
                    //eg: custBuy[3] = {item[0], item[4], item[2]};
               }

               cust[cntCust] = new Customer(custID, custName, custBuy);
```

```
                    cntCust++;
                    System.out.print("\n To continue press [1-Yes,0-No]: ");
                    cont = scan.nextInt();

              }

              System.out.println("\n List of customer and item purchased  ");

              for (int x = 0;x<cntCust;x++){
                    Item[] totItem = cust[x].getItem();
                    int cntNotNull = 0;
                    for (int a=0;a<totItem.length;a++)
                    {//to determine how many items purchased that stored
                     //in the array totItem
                          if (totItem[a] != null){
                                cntNotNull++;
                          }
                    }

                    cust[x].display(cntNotNull);
                    System.out.println("\n **************************** ");
              }
        }//close main
} //close class
```

Figure 4: Program segment implementing customer purchases of items

The code segment initializes an array named `item` to store the profiles of five items. An array of `Customer` objects, referred to as `cust`, is used to store customer information, including customer ID, customer name, and an aggregated `Item` object. Each customer may select multiple items for purchase, and the selected items are stored within the `item` array. During instantiation, the customer's ID, name, and associated `Item` objects are assigned to the `cust` array. The application prompts the user to choose whether to continue or terminate the program. Before the program concludes, it displays a list of customers along with detailed information about the items purchased. Additionally, the total purchase amount for each customer is presented.

The concept of aggregation is effectively implemented in the application program through the relationship established between the Customer and Item classes. In object-oriented programming, aggregation represents a "has-a" relationship where one object contains or references another object without taking ownership of it. In this case, each Customer object is associated with one or more Item objects, indicating that a customer can purchase multiple items. This relationship is implemented by passing an array of Item objects as a parameter to the Customer class constructor during the object instantiation process. As a result, the Customer class holds a reference to the Item objects, allowing the program to manage customer purchases effectively while keeping the Item class independent. This design promotes modularity and separation of concerns, as the Item class can function independently and be reused elsewhere if needed. During program execution, the user is allowed to input the

customer's ID, name, and select the desired items for purchase by entering item IDs. The selected items are stored in the item array, and together with the customer information, they are used to create a new Customer object stored in the cust array. This structure clearly demonstrates aggregation, where the Customer object maintains a reference to multiple Item objects. Before the application ends—either when the user chooses to stop or all entries are completed—the program displays a summary of each customer's details, the list of purchased items, and the total amount spent. This implementation of aggregation not only ensures a clear and logical organization of data but also enhances the overall readability and maintainability of the code.

Conclusion

In conclusion, this article has demonstrated the application of aggregation as a fundamental concept of association in object-oriented programming through the use of the Java programming language. By designing and implementing the Item and Customer classes, we established a clear one-to-many relationship, where each customer can be associated with multiple items without implying ownership. This form of association was effectively realized by passing arrays of Item objects as parameters to the Customer class constructor, illustrating how objects can collaborate and interact within a program. The aggregation approach not only promotes modularity and reusability of code but also reflects real-world relationships in software design, making it easier to manage complex data structures (GeeksforGeeks, 2025). Through the development of a simple yet functional application, we showed how aggregation facilitates structured data handling, allowing the system to track customer purchases, display itemized details, and compute total expenditures (Gupta, 2023). This practical implementation reinforces the importance of understanding and applying object-oriented principles such as aggregation to build efficient, scalable, and maintainable software systems.

References

Asagba, P., Ogheneovo, E. (2010). A Comparative Analysis of Structured and Object-Oriented Programming Methods. Journal of Applied Sciences and Environmental Management (ISSN: 1119-8362) Vol 12 Num 4. 11. 10.4314/jasem.v11i4.55190.

GeeksforGeeks. (2025, July 23). Aggregation in OOAD. Retrieved from https://www.geeksforgeeks.org/aggregation-in-ooad/

Gupta, L. (2023). Association, aggregation and composition in Java [Tutorial]. HowToDoInJava. Retrieved from https://howtodoinjava.com/java/oops/association-aggregation-composition

Hutabarat, B. I., Purnama, K, E., Hariadi M. (2009). The 5th International Conference on Information & Communication Technology and Systems

Larman, C. (2004). Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development (3rd ed.). Prentice Hall.

Otu, G. A., Usman, S. A., Ugbe, R. U., Iheagwara, S. E., Okafor, A. C., Okonkwo, F. I., Shukurah, O. M., & Adedayo, A. U. (2023). Comparative analysis of aggregation and inheritance strategies in incremental program development. FUDMA Journal of Sciences, 43(3), Article 9. https://doi.org/10.33003/fjs-2023-0702-1710

Vincent O. R., Afolorunso A. A. (2020) CIT383 Introduction to Object-Oriented Programming. National Open University of Nigeria

SIG CS@e-Learning
Unit Penerbitan
Jabatan Sains Komputer & Matematik
Universiti Teknologi MARA Cawangan Pulau Pinang

e-ISBN : 978-629-98755-x-x

Design of the cover powered by
https://www.free-powerpoint-templates-design.com/