**BMCS2023 Object-Oriented Programming Practical #1**

|  |
| --- |
| **Required Tools**  Before you start to code and execute your Java programs, installations of the following tools are required:  ⮚ ***Java SE Development Kit (***[***https://www.oracle.com/java/technologies/downloads/#jdk21windows)***](https://www.oracle.com/java/technologies/downloads/#jdk21-windows)  ⮚ ***Integrated development environment* (*IDE*), *Apache NetBeans***  [***(https://netbeans.apache.org/download/index.html)***](https://netbeans.apache.org/download/index.html) |

# Question 1

Write a Java program that obtains the following input from the user:

⮚ Name

⮚ Current year of study

⮚ Target GPA for this semester

Then display a welcome message with the details that was input earlier. A sample dialog is shown below:

Enter name: Phua Chu Kang

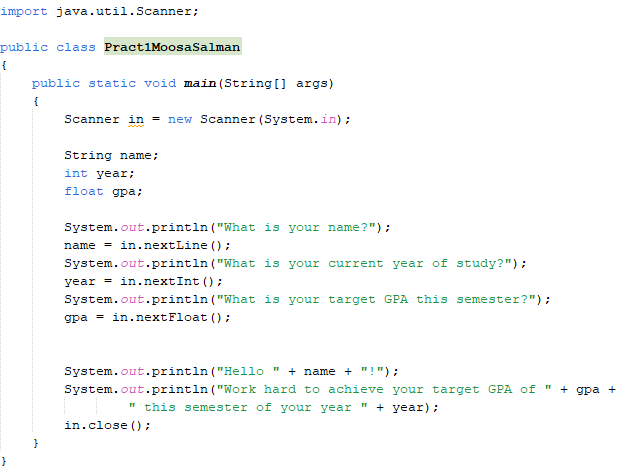
Enter your year of study: 2

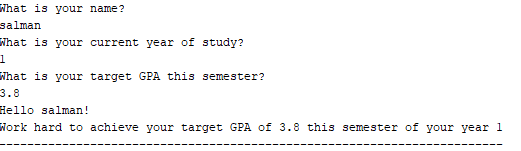
What is your target GPA for this semester? 3.75

Welcome Phua Chu Kang!

Work hard to achieve your target GPA of 3.75 this semester of your Year 2.

Solution:





# Question 2

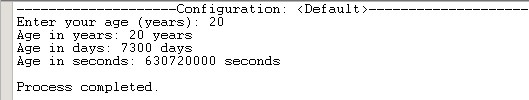
Write a Java program that prompts the user to enter his / her age. The program should then calculate and display the user’s age in terms of days and seconds, as shown below:

**Note**

:

Practice good programming style

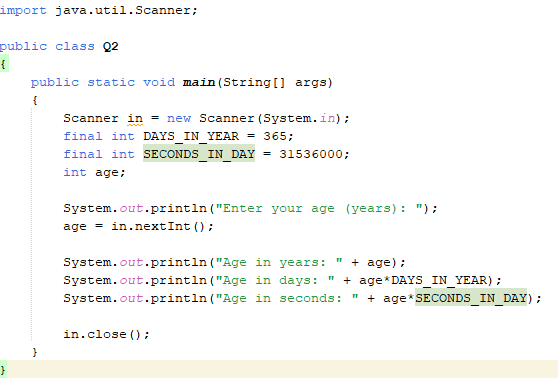
:

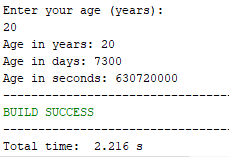


⮚ Follow the standard Java naming convention

⮚ Define and use constants for permanent data and values that never changes

Solution:





# Question 3

Write a Java program that validates a credit card number. The last digit of a credit card number is the check digit, which protects against transcription errors such as an error in a single digit or switching two digits.

The program should prompt the user to enter an 8-digit number, and will print out whether the number is valid or not. If it is not valid, it should print the value of the check digit that would make it valid.

Use the following method for verifying the credit card:

⮚ Starting from the rightmost digit, form the sum of every other digit. For example, if the credit card number is 4358 9795, then you form the sum

5 + 7 + 8 + 3 = 23

⮚ Double each of the digits that were not included in the preceding step. Add all digits of the resulting numbers. For example, with the number given above, doubling the digits, starting with the next-tolast one, yields

18 18 10 8

Adding all digits in these values yields 1 + 8 + 1 + 8 + 1 + 0 + 8 = 27.

⮚ Add the sums of the two preceding steps. If the last digit of the result is 0, the number is valid. In our case, 23 + 27 = 50, so the number is valid.

Solution:

import java.util.Scanner;

public class Q3

{

public static int sumOtherDigit(int num)

{

int sum = 0;

int altSum = 0;

boolean addDigit = true; //start fom first digit

num = Math.abs(num);

while (num>0)

{

int digit = num % 10; // extract last digit e.g 45 % 10 = 5

if(addDigit)

{

sum+=digit;

}

else

{

altSum = digit\*2; // double and add every other digit

}

addDigit = !addDigit; //Skip Next digit

num /= 10; //truncate last digit

}

return sum;

}

public static int sumDoubleDigit(int num)

{

int sum = 0;

int altSum = 0;

boolean addDigit = true; //start fom first digit

while (num>0)

{

int digit = num % 10; // extract last digit e.g 45 % 10 = 5

if(addDigit)

{

sum+=digit;

}

else

{

// double and add every other digit

int doubledDigit = digit \* 2;

if (doubledDigit >= 10)

{

// If it becomes double digit, add its individual digits

altSum += (doubledDigit % 10) + (doubledDigit / 10);

}

else

{

// Otherwise, just add the doubled digit

altSum += doubledDigit;

}

}

addDigit = !addDigit; //Skip Next digit

num /= 10; //truncate last digit

}

return altSum;

}

public static void main(String[] args)

{

Scanner in = new Scanner(System.in);

int cardNum;

System.out.println("Please enter your 8 digit card number: ");

cardNum = in.nextInt();

System.out.println("Sum: " + sumOtherDigit(cardNum));

System.out.println("Alt Sum: " + sumDoubleDigit(cardNum));

if((sumOtherDigit(cardNum) + sumDoubleDigit(cardNum)) % 10 == 0){

System.out.println("Card Number is Valid");

}

else

{

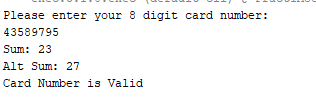
System.out.println("Card Number is Invalid!");

}

in.close();

}

}



# Question 4

1. Briefly describe the steps in the problem-solving process for Object-Oriented Design.

1. Distinguish between Structured Programming and Object-Oriented Programming in terms of their focus area.

Solution:

A)

1. **Understand the Problem (Analysis):** This initial phase involves thoroughly understanding the requirements and the domain of the problem. It's about identifying what the system needs to do, who its users are, and what constraints exist. Techniques like use cases, user stories, and requirements gathering are crucial here.
2. **Identify Objects and Classes (Conceptualization):** Based on the problem understanding, the next step is to identify the core entities (nouns) that interact within the system. These entities will typically become classes. For each identified class, their attributes (data they hold) and behaviors/responsibilities (actions they can perform or that can be performed on them) are defined.
3. **Define Relationships between Objects (Design):** Once classes are identified, their relationships are established. This includes:
   * **Association:** How objects interact with each other (e.g., a Customer *places* an Order).
   * **Aggregation/Composition:** "Has-a" relationships where one object is part of another (e.g., an Order *has* LineItems).
   * **Inheritance:** "Is-a" relationships where one class is a specialized version of another (e.g., a SavingsAccount *is a* type of BankAccount).
   * **Dependency:** When one class uses another, but doesn't "contain" it.
4. **Design Object Interactions (Collaboration):** This step focuses on how objects collaborate to fulfill the system's responsibilities. It involves defining the methods (functions) within each class that allow objects to communicate and work together. Sequence diagrams and communication diagrams can be used to model these interactions.
5. **Refine and Iterate (Implementation & Testing Considerations):** OOD is often an iterative process. Initial designs are rarely perfect. This step involves reviewing, refining, and optimizing the design. Considerations include:
   * **Encapsulation:** Ensuring data is hidden and accessed only through well-defined interfaces.
   * **Polymorphism:** Using common interfaces for different object types.
   * **Modularity:** Breaking down the system into manageable, independent components.
   * **Flexibility and Extensibility:** Designing for future changes and additions.
   * This stage often overlaps with actual coding and testing, where issues with the design may become apparent and lead back to earlier steps for refinement.

B)

| Feature | Structured Programming | Object-Oriented Programming (OOP) |
| --- | --- | --- |
| **Focus Area** | **Functions/Procedures and Data Separation** | **Objects (Data + Behavior) and Their Interactions** |
| **Primary Goal** | Breaking down a program into smaller, manageable procedures or functions to improve code readability, maintainability, and reusability through control structures (sequence, selection, iteration). | Modeling real-world entities as objects that encapsulate both data and the operations that can be performed on that data, promoting modularity, reusability, and extensibility. |
| **How Problems Are Decomposed** | Top-down decomposition: Problems are broken down into a series of steps, and each step is implemented as a function or procedure. Data is often global or passed between functions. | Bottom-up decomposition: Problems are modeled by identifying objects, defining their attributes and behaviors, and then designing how these objects interact. |
| **Data Handling** | Data and functions are typically separated. Functions operate on data that may be globally accessible or passed as parameters. Less emphasis on data hiding. | Data and the functions (methods) that operate on that data are bundled together within objects (encapsulation). Data is often hidden and accessed only through defined interfaces. |
| **Key Principles** | - Modularity (through functions) - Sequence, Selection, Iteration (control flow) | - **Encapsulation:** Bundling data and methods into a single unit (object). - **Inheritance:** Creating new classes from existing ones. - **Polymorphism:** Objects taking on many forms. - **Abstraction:** Hiding complex implementation details. |
| **Suitability** | Ideal for smaller to medium-sized applications with well-defined, sequential processes. | Highly suitable for complex, large-scale applications where system evolution, reusability, and maintainability are critical. Often used for GUI applications, simulations, and systems modeling real-world entities. |