Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Control Flow Statement & Overloading

(IT069IU)

Le Duy Tan, Ph.D.

ldtan@hcmiu.edu.vn

leduytanit.com

Previously,

We talked about:

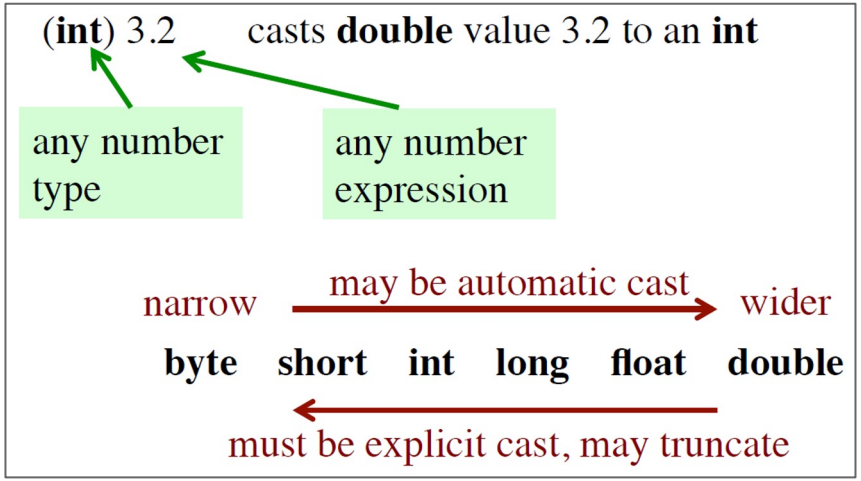
- **Useful Classes**
  - **Scanner**
    - Read input string with nextLine(), next()
    - Read input number with nextDouble()
  - **String**
    - Display string with print(), println(), printf()
  - **Math**
    - pow(), max(), random()
- **Primitive** vs **Reference**
- **Class**
  - **Attributes**
  - **Method** with Parameters
    - **Getters** and **Setters** Methods
  - **Access Modifiers** (Public & Private)
  - **Constructor** with Parameters
  - **UML** Diagram
- **Object**
  - **Create objects** from class with keyword **new**
  - Call **methods** with input arguments
- Bank account application example

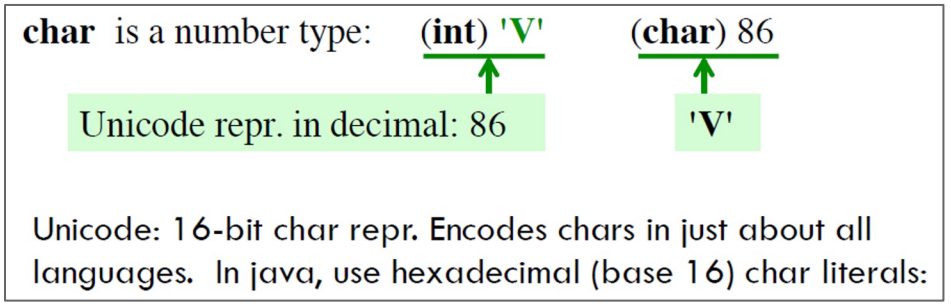
Agenda

- **Casting** types.
- Control flow statements:
  - Decision making statements:
    - **If**
    - **If...else**
    - **Switch**
  - Loop statements:
    - **While**
    - **Do while**
    - **For**
  - Jump statements:
    - **Break** statement
    - **Continue** statement
- **Static** Keyword
  - Static Method
  - Static Variable
- **Overloading**
  - Method Overloading
  - Constructor Overloading
- **Final** Keyword
  - Constant Variable

Casting among types

- Convert one data type to another data type.





[Question]: Which example has automatic cast, and which one has explicit cast?

```
int myInt = 9;
double myDouble = myInt;
System.out.println(myInt);
```

```
double myDouble = 9.4;
int myInt = (int) myDouble;
System.out.println(myInt);
```

Assignment Type Restriction

Rule: In x = e; type of e has to be same as or narrower than type of x.

Reason: To avoid possibly losing info without the programmer realizing it.

```
double y= 5 + 1;
```

The value of 5+1 is automatically cast from type **int** to type **double**.

```
int x= 75.5 + 1;
```

Illegal: The exp value is of type **double**.

```
int x= (int) (75.5 + 1);
```

You can cast to **int** explicitly. 76 will be stored in x.

Control Flow Statements

Decision Making Statements

If, Else, Switch

If Statement

- **Three** types of decision making statements:
  - **if** statement:
    - Performs an action, if a condition is true; skips it, if false.
    - **Single-selection statement**—selects or ignores a single action (or group of actions).
  - **if...else** statement:
    - Performs an action if a condition is true and performs a different action if the condition is false.
    - **Double-selection statement**—selects between two different actions (or groups of actions).
  - **switch** statement
    - Performs one of several actions, based on the value of an expression.
    - **Multiple-selection statement**—selects among many different actions (or groups of actions).

If Single-Selection Example

- Pseudocode:

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

- If the condition is false, the Print statement is ignored.
- Indentation:
  - Optional, but recommended
  - Emphasizes the inherent structure of structured programs
- For single statement, { } can be omitted.
- Java code:

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

```
if ( studentGrade >= 60 ){
   System.out.println( "Passed" );
}
```

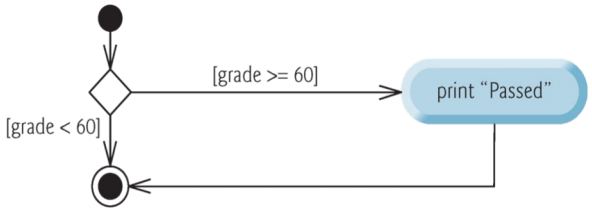


Fig. 4.2 | `if` single-selection statement UML activity diagram.

If Double-Selection Example

- Pseudocode:

```
If student's grade is greater than or equal to 60
        Print "Passed"
Else
        Print "Failed"
```

- Specify an action to perform when the condition is true and a different action when the condition is false.
- Java code:

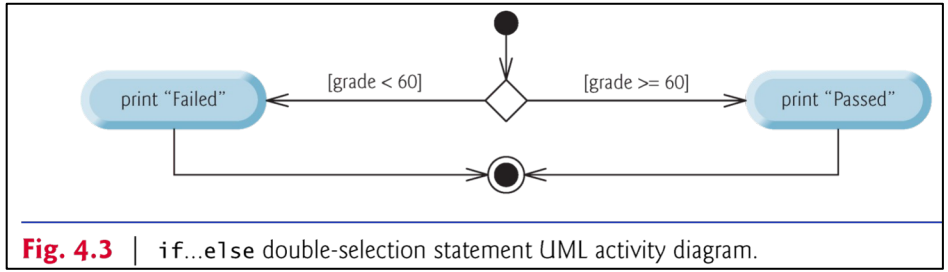


Fig. 4.3 | `if...else` double-selection statement UML activity diagram.

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

```
if ( grade >= 60 ){
   System.out.println( "Passed" );
} else{
   System.out.println( "Failed" );
}
```

Ternary Operator - If...Else short form

- Conditional operator (?:)—**shorthand if...else.**
- Ternary operator (takes three operands)
- Only suitable for simple comparison condition and make sure the conditional expression to be easy to understand.

```
System.out.println( grade >= 60 ? "Passed" : "Failed" );
```

Nested If … Else Statements

- Both styles are corrected. But most Java developers prefer the right one.

```
if ( studentGrade >= 90 )
  System.out.println( "A" );
else
  if ( studentGrade >= 80 )
    System.out.println( "B" );
  else
    if ( studentGrade >= 70 )
      System.out.println( "C" );
    else
      if ( studentGrade >= 60 )
        System.out.println( "D" );
      else
        System.out.println( "F" );
```

```
if ( studentGrade >= 90 )
  System.out.println( "A" );
else if ( studentGrade >= 80 )
  System.out.println( "B" );
else if ( studentGrade >= 70 )
  System.out.println( "C" );
else if ( studentGrade >= 60 )
  System.out.println( "D" );
else
  System.out.println( "F" );
```

This is confusing!

Without braces { }, The Java compiler always **associates an else with the nearest if before it.**
Referred to as the dangling-else problem.

It looks like:

```
if ( x > 5 )
        if ( y > 5 )
                                System.out.println( "x and
y are > 5" );
else
        System.out.println( "x is <= 5" );
```

It actually is:

```
if ( x > 5 )
        if ( y > 5 )
                                System.out.println( "x and
y are > 5" );
        else
                                System.out.println( "x is <=
```

No more confusing! Use braces {} for nest IF!

[Question] the else statement belongs to which if in the first box and second box?

```
if ( x > 5 ){
        if ( y > 5 )
                System.out.println( "x and y are > 5" );
        else
                System.out.println( "x is <= 5" );
}
```

```
if ( x > 5 ){
        if ( y > 5 )
                System.out.println( "x and y are > 5" );
} else
System.out.println( "x is <= 5" );
```

Multiple statements in an IF block

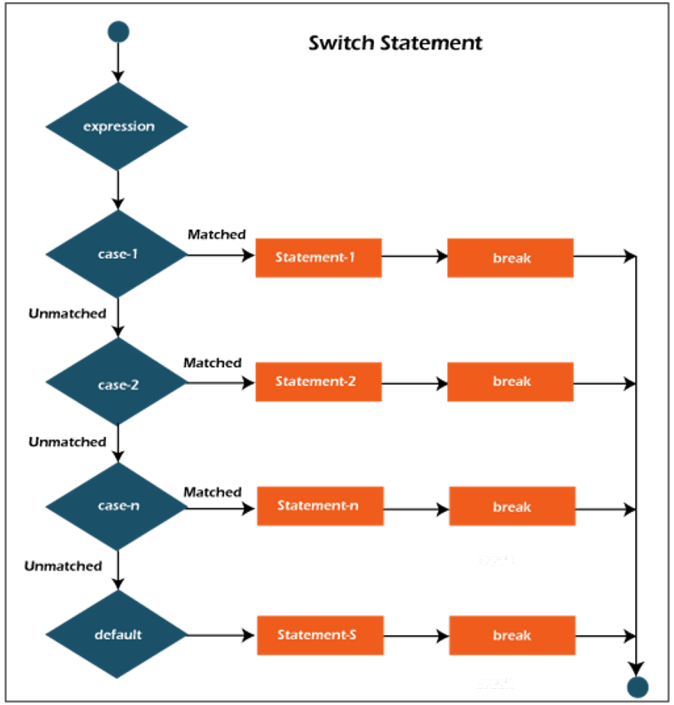
- The if statement normally expects only one statement in its body.
- To **include several statements** in the body of an if (or the body of an else for an if…else statement), **enclose the statements in braces { }.**
- Statements contained in a pair of braces form **a block.**
- A block can be placed anywhere that a single statement can be placed.

```
if ( grade >= 60 )
        System.out.println("Passed");
else {
        System.out.println("Failed");
        System.out.println("You must take this course again.");
}
```

Switch Statement

- To select one of many code blocks to be executed.
- The **value of expression** is **compared** to the **values of each case.**
- If matched, the block of that case is executed.
- **break** keyword will breaks out of the whole block.
- **default** keyword define a default block will run at the end if no case is matched.

```
switch(expression) {
    case x:
        // code block
        break;
    case y:
        // code block
        break;
    default:
        // code block
}
```



Switch Example

```java
int day = 4;
switch (day) {
    case 6:
        System.out.println("Today is Saturday");
        break;
    case 7:
        System.out.println("Today is Sunday");
        break;
    default:
        System.out.println("Looking forward to the Weekday!");
}
```

[Question]

- What happens if you remove “**break**” keyword in one case?
- What happens if grade is “**B**”?

```java
char grade = 'C';
switch(grade) {
    case 'A' :
        System.out.println("Excellent!");
        break;
    case 'B' :
    case 'C' :
        System.out.println("Well done");
        break;
    case 'D' :
        System.out.println("You passed");
    case 'F' :
        System.out.println("Better try again");
        break;
    default :
        System.out.println("Invalid grade");
}
System.out.println("Your grade is " + grade);
```

Loop Statements

While, Do...While, For

Loop Statements

- **Repetition statements** (also called **looping statements**)
- Perform statements repeatedly while **a loop-continuation condition** remains true.
- **while** and **for** statements perform the action(s) in their bodies **zero or more times.**
  - if the loop-continuation condition is initially false, the body will not execute.
- The **do...while** statement performs the action(s) in its body **one or more times.**

While Statement

- Pseudocode

```
While there are more items on my shopping list
    Buy next item and cross it off my list
```

- **Repetition statement—repeats an action while a condition remains true**.
- The repetition statement's body may be a single statement or a block.
- Eventually, **the condition will become false. At this point, the repetition terminates**, and the statements after while block will continue.

```
while (boolean condition)
    single_statement;
```

```
while (boolean condition){
        statement_1;
    statement_2;
    …
    statement_n;
}
```

While Loop Example

```
int product= 1;
while ( product <= 100)
      product = product*3;
System.out.print(product);
```

[Question] What is the value of product when the while loop finishes)?

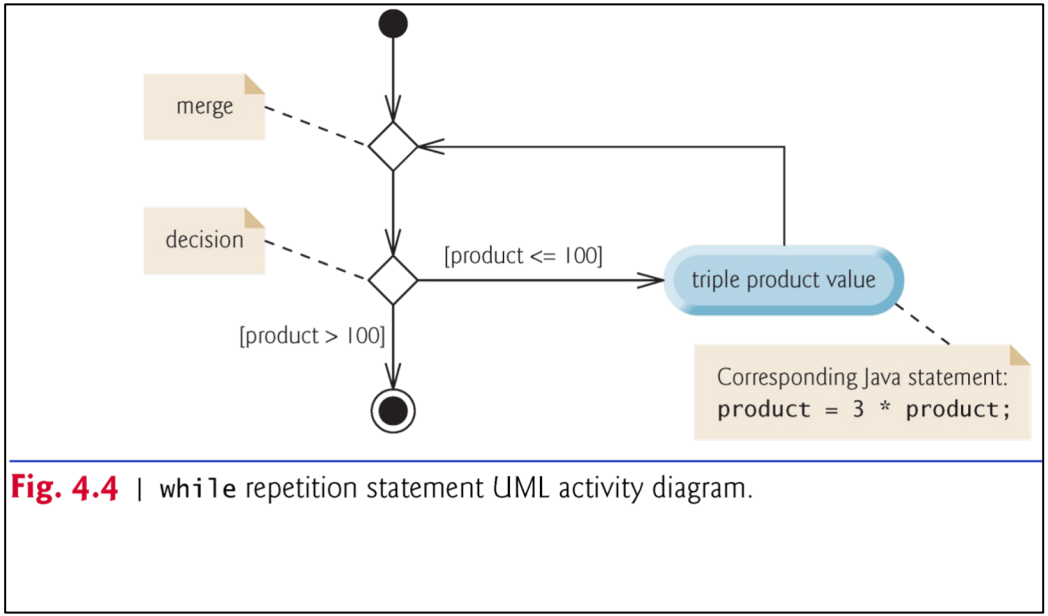


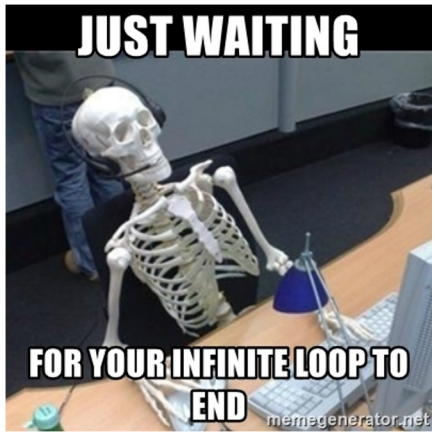
Fig. 4.4 | `while` repetition statement UML activity diagram.

Danger of Infinite Loop

- Beware about your stopping condition of any loop.

```
int product= 1;
while ( product >=0)
      product = product*3;
System.out.print(product);
```

[Question] What is the value of product when the while loop finishes)?



Analysis Class

- Write an Analysis Class:
  - Let instructor to input performance status of 10 student (pass or fail)
  - Output the number of students passed and failed
  - If the number of students passed equal or larger than 9 then give instructor an bonus.

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Analysis.java

```
1  // Fig. 4.12: Analysis.java
2  // Analysis of examination results using nested control statements.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis
6  {
7     public static void main( String[] args )
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner( System.in );
11
12       // initializing variables in declarations
13       int passes = 0; // number of passes
14       int failures = 0; // number of failures
15       int studentCounter = 1; // student counter
16       int result; // one exam result (obtains value from user)
17
```

Analysis.java

```
18          // process 10 students using counter-controlled loop
19          while ( studentCounter <= 10 )
20          {
21             // prompt user for input and obtain value from user
22             System.out.print( "Enter result (1 = pass, 2 = fail): " );
23             result = input.nextInt();
24
25             // if...else is nested in the while statement
26             if ( result == 1 )           // if result 1,
27                passes = passes + 1;      // increment passes;
28             else                         // else result is not 1, so
29                failures = failures + 1; // increment failures
30
31             // increment studentCounter so loop eventually terminates
32             studentCounter = studentCounter + 1;
33          } // end while
34
```

Analysis.java

```
35          // termination phase; prepare and display results
36          System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );
37
38          // determine whether more than 8 students passed
39          if ( passes > 8 )
40             System.out.println( "Bonus to instructor!" );
41       } // end main
42    } // end class Analysis
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Do...While Loop

- Instead of checking loop condition at the start, do...while will **check loop condition at the end.**
- To ensure the **loop block will start at least once** even if the loop condition is even false at the beginning.

```
1  // Fig. 5.7: DoWhileTest.java
2  // do...while repetition statement.
3
4  public class DoWhileTest
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // initialize counter
9
10       do
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter;
14       } while ( counter <= 10 ); // end do...while
15
16       System.out.println(); // outputs a newline
17    } // end main
18 } // end class DoWhileTest
```

Condition tested at end of loop, so loop always executes at least once

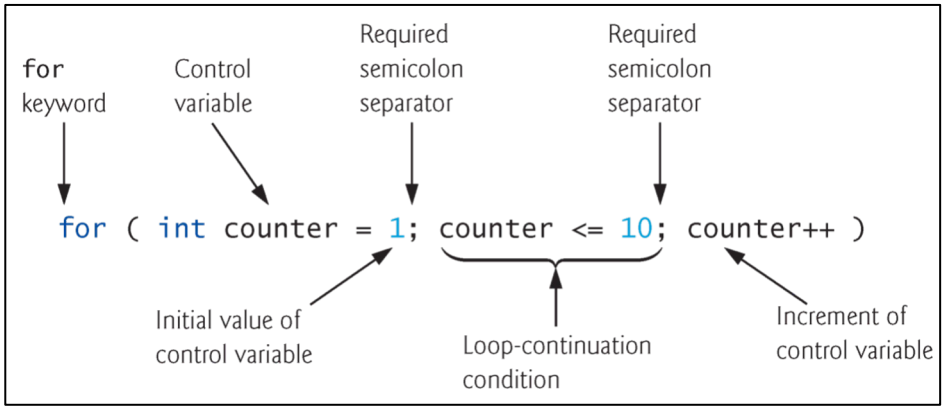
```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.7 | do...while repetition statement.

For Loop - Counter-Controlled Repetition

- For Loop is for when we **know exactly how many times we want to loop** through a block of code. (While is for when we only know stopping condition).
- initial_statement: **executed (one time)** at the beginning.
- stop_condition: **loop condition** to allow the loop to continue.
- Update_statement: **executed (every time)** after the code block has been executed.

```
for (initial_statement; stop_condition; update_statement) {
  // code block to be executed
}
```



For Loop Example

```
for (int i = 0; i < 5; i+=1) {
      System.out.println(i);
}
```

```
for (int i = 0; i <= 10; i=i + 2) {
   System.out.println(i);
}
```

[**Question**] What is the output for these two code?

For Loop Example

Sum.java

```
1  // Fig. 5.5: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum
5  {
6     public static void main( String[] args )
7     {
8        int total = 0; // initialize total
9
10       // total even integers from 2 through 20
11       for ( int number = 2; number <= 20; number += 2 )
12          total += number;
13
14       System.out.printf( "Sum is %d\n", total ); // display results
15    } // end main
16 } // end class Sum
```

[Question] What happens if we change "number +=2" to "number -=2" ?

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

Nested For Loop

- If a loop exists inside the body of another loop, it's called a nested loop.

```java
int weeks = 3;
int days = 7;

// outer loop prints weeks
for (int i = 1; i <= weeks; ++i) {
    System.out.printf("Week %d: ",i);

    // inner loop prints days
    for (int j = 1; j <= days; ++j) {
        System.out.printf("%d ",j);
    }
    System.out.println();
}
```

```java
int weeks = 3;
int days = 7;
int i = 1;

// outer loop prints weeks
while (i<= weeks){
    System.out.printf("Week %d: ",i);

    // inner loop prints days
    for (int j = 1; j <= days; ++j) {
        System.out.printf("%d ",j);
    }
    i+=1;
    System.out.println();
}
```

```
Week 1: 1 2 3 4 5 6 7
Week 2: 1 2 3 4 5 6 7
Week 3: 1 2 3 4 5 6 7
```

- These two code uses different ways of loopings but the output is the same!

Banking Compound Interest Application

Compound interest application

- A person invests $1000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p * (1 + r) * n$$

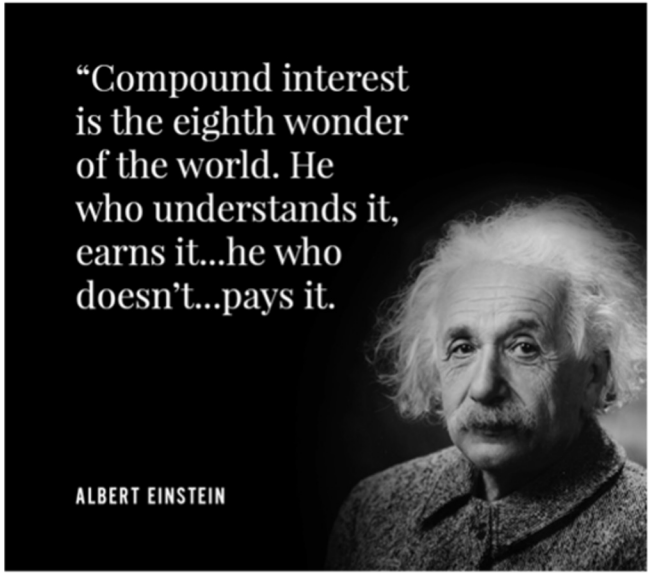
where

p is the original amount invested (i.e., the principal)

r is the annual interest rate (e.g., use 0.05 for 5%)

n is the number of years

a is the amount on deposit at the end of the nth year.



Interest.java

```
1  // Fig. 5.6: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest
5  {
6     public static void main( String[] args )
7     {
8        double amount; // amount on deposit at end of each year
9        double principal = 1000.0; // initial amount before interest
10       double rate = 0.05; // interest rate
11
12       // display headers
13       System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
14
15       // calculate amount on deposit for each of ten years
16       for ( int year = 1; year <= 10; year++ )
17       {
18          // calculate new amount for specified year
19          amount = principal * Math.pow( 1.0 + rate, year );
20
21          // display the year and the amount
22          System.out.printf( "%4d%,20.2f\n", year, amount );
23       } // end for
24    } // end main
25 } // end class Interest
```

Year	Amount on deposit
1	1,050.00
2	1,102.50
3	1,157.63
4	1,215.51
5	1,276.28
6	1,340.10
7	1,407.10
8	1,477.46
9	1,551.33
10	1,628.89

Java treats floating-point literals as `double` values

Uses `static` method `Math.pow` to help calculate the amount on deposit

Comma in format specifier indicates that large numbers should be displayed with thousands separators

Jump Statements

Break & Continue

Keyword "break"

- Can break out of any type loop (while, do...while, for, switch)

```
for (int i = 1; i<=10; i+=1){
    if (i==7){
        break;
    }
    System.out.println(i);
}
```

```
int i = 1;
do{
    if (i == 7) {
        break;
    }
    System.out.println(i);
    i+=1;
}while (i <= 10);
```

[Question] What is the output for each loop?

```
int i = 1;
while (i <= 10) {
    if (i == 7) {
        break;
    }
    System.out.println(i);
    i+=1;
}
```

“break” in nested loop

- **Each break** keyword can only **break out of one layer** of loop.

```
int weeks = 3;
int days = 7;
// outer loop prints weeks
for (int i = 1; i <= weeks; ++i) {
    System.out.printf("Week %d: ",i);

    // inner loop prints days
    for (int j = 1; j <= days; ++j) {
        if (j == 4)
            break;
        System.out.printf("%d ",j);
    }
    System.out.println();
}
```

```
Week 1: 1 2 3
Week 2: 1 2 3
Week 3: 1 2 3
```

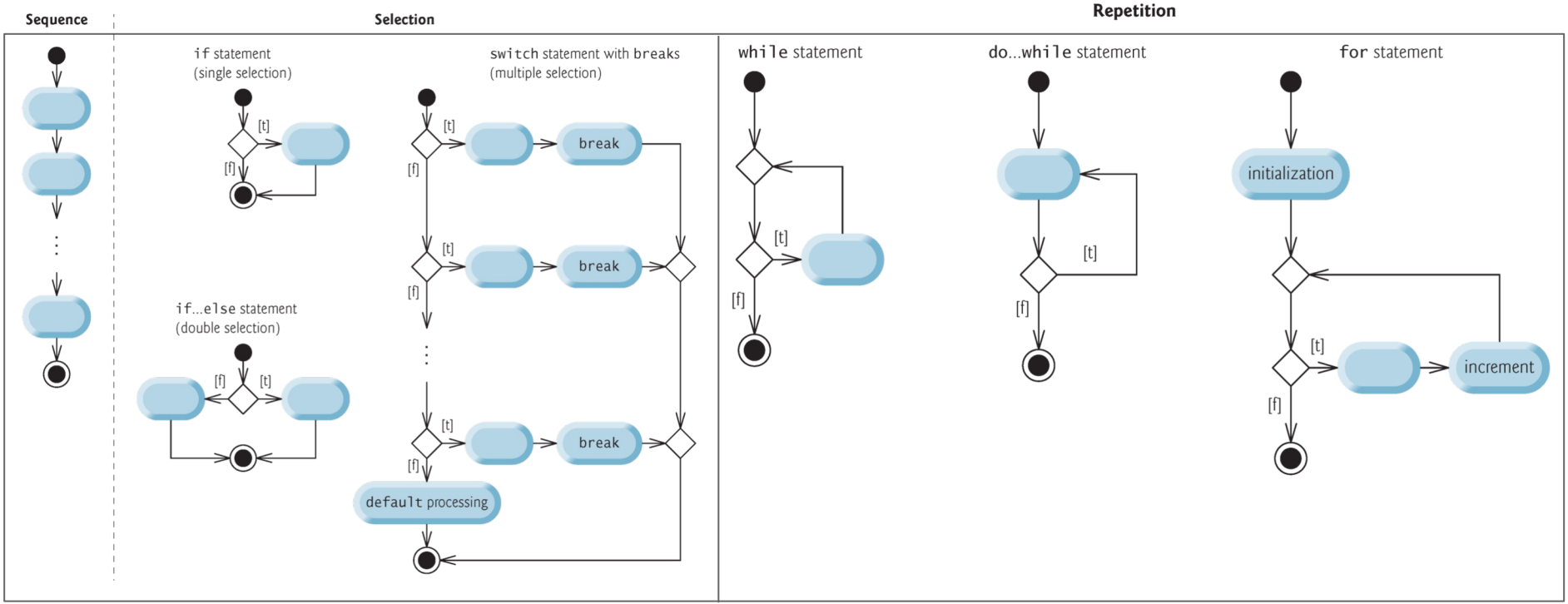
Keyword “continue”

- “Continue” will skips the remaining statements in the loop body and proceeds with the next iteration of the loop.

```
1  // Fig. 5.14: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest
4  {
5     public static void main(String[] args)
6     {
7        for (int count = 1; count <= 10; count++) // loop 10 times
8        {
9           if (count == 5)
10             continue; // skip remaining code in loop body if count is 5
11
12          System.out.printf("%d ", count);
13       }
14
15       System.out.printf("%nUsed continue to skip printing 5%n");
16    }
17 } // end class ContinueTest
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Control & Loop Flow Diagram



Let's Wrap Up! GradeBook Class!

- Write an GradeBook Class:
  - Let instructor to input integer value for grade of any number of student in a class. (Only stopped if he/she input -1)
  - Output the number of grades & the total sum.
  - Also, calculate the average of grades for the whole class.

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

GradeBook.java

```
1  // Fig. 4.9: GradeBook.java
2  // GradeBook class that solves the class-average problem using
3  // sentinel-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
9
10    // constructor initializes courseName
11    public GradeBook( String name )
12    {
13       courseName = name; // initializes courseName
14    } // end constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 1 of 4.)

GradeBook.java

```
22      // method to retrieve the course name
23      public String getCourseName()
24      {
25         return courseName;
26      } // end method getCourseName
27
28      // display a welcome message to the GradeBook user
29      public void displayMessage()
30      {
31         // getCourseName gets the name of the course
32         System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33            getCourseName() );
34      } // end method displayMessage
35
36      // determine the average of an arbitrary number of grades
37      public void determineClassAverage()
38      {
39         // create Scanner to obtain input from command window
40         Scanner input = new Scanner( System.in );
41
42         int total; // sum of grades
43         int gradeCounter; // number of grades entered
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 2 of 4.)

GradeBook.java

```
44          int grade; // grade value
45          double average; // number with decimal point for average
46
47          // initialization phase
48          total = 0; // initialize total
49          gradeCounter = 0; // initialize loop counter
50
51          // processing phase
52          // prompt for input and read grade from user
53          System.out.print( "Enter grade or -1 to quit: " );
54          grade = input.nextInt();
55
56          // loop until sentinel value read from user
57          while ( grade != -1 )
58          {
59             total = total + grade; // add grade to total
60             gradeCounter = gradeCounter + 1; // increment counter
61
62             // prompt for input and read next grade from user
63             System.out.print( "Enter grade or -1 to quit: " );
64             grade = input.nextInt();
65          } // end while
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 3 of 4.)

```
66
67          // termination phase
68          // if user entered at least one grade...
69          if ( gradeCounter != 0 )
70          {
71             // calculate average of all grades entered
72             average = (double) total / gradeCounter;
73
74             // display total and average (with two digits of precision)
75             System.out.printf( "\nTotal of the %d grades entered is %d\n",
76                gradeCounter, total );
77             System.out.printf( "Class average is %.2f\n", average );
78          } // end if
79          else // no grades were entered, so output appropriate message
80             System.out.println( "No grades were entered" );
81       } // end method determineClassAverage
82    } // end class GradeBook
```

Fig. 4.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 4 of 4.)

GradeBookTest.java

```
1  // Fig. 4.10: GradeBookTest.java
2  // Create GradeBook object and invoke its determineClassAverage method
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.determineClassAverage(); // find average of grades
15    } // end main
16 } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. 4.10 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.9) and invokes its `determineClassAverage` method. (Part 1 of 2.)

Static Keyword

Class Method & Class Variable

Static Methods (Class Method)

- Some classes also provide **methods** that **perform common tasks** and **do not require** you to **create objects of those classes**.
- Instead, we call the method from an object of a class:

```
ClassObjects.methodName(arguments)
```

- We can call class method right from a class:

```
ClassName.methodName(arguments)
```

- For example, you can calculate the square root of 900.0 with the static method call “sqrt” of the Math class:

```
Math.sqrt(900.0)
```

Static Method Example

Calculator.java

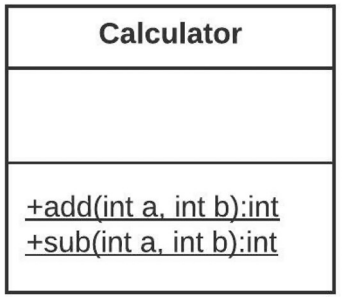
```java
public class Calculator {
    public static int add(int a, int b){
        return a+b;
    }

    public static int sub(int a, int b){
        return a-b;
    }
}
```

CalculatorTest.java

```java
public class CalculatorTest {
    public static void main(String[] args) {
        int mySum = Calculator.add( a: 5,  b: 6);
        int mySub = Calculator.sub( a: 8,  b: 5);
        System.out.println(mySum);
        System.out.println(mySub);
    }
}
```

UML for Calculator Class



[**Question**] What is the output of mySum and mySub?

Static Variables (Class Variables)

- Remember, each object of a class maintains its own values of every instance variable (attribute value) of the class.
- **Since static variables are initialized only once, and they're shared between all instances of that class.**
- For example, Class Math have two static constants, **Math.PI** (3.14159) and **Math.E** (2.71828)
- Making these fields **static** allows them to be accessed via the class name Math and a dot (.)

Static Variable & Method Example

Account.java

```java
public class Account {
    private static int count;

    private String name;
    private int id;

    public Account(String personName, int personID){
        name = personName;
        id = personID;
        count+=1;
    }

    public static int getNumberOfAccounts(){
        return count;
    }

    public String getName() {
        return name;
    }

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

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

EdusoftSystem.java

```
public class EdusoftSystem {
    public static void main(String[] args) {
        System.out.printf("Before, %d accounts.\n",Account.getNumberOfAccounts());
        Account tomAccount =  new Account( personName: "Tom",  personID: 54573);
        Account jerryAccount =  new Account( personName: "Jerry",  personID: 12345);
        System.out.printf("After, %d accounts.\n", Account.getNumberOfAccounts());

        System.out.printf("Student 1: %s - %d\n",tomAccount.getName(),tomAccount.getId());
        System.out.printf("Student 2: %s - %d\n",jerryAccount.getName(),jerryAccount.getId());
    }
}
```

Output:

```
Before, 0 accounts.
After, 2 accounts.
Student 1: Tom - 54573
Student 2: Jerry - 12345
```

[Question] Why do we use static variable and static method in Account class?

Why Method main is static?

- Remember, we have:

```
public static void main(String args[])
```

- When you execute the Java Virtual Machine (**JVM**) with the java command, the JVM attempts to **invoke the main method** of the class you specify—**at this point no objects of the class have been created**. **Declaring main as static allows the JVM to invoke main without creating an instance of the class**.
- Remember, when we execute an application, we need to specific class name as an argument to java command:

```
java ClassName argument1 argument2 ...
```

- By having main method to be static, the JVM loads the class specified by ClassName and uses that class name to invoke method main.

Overloading

Method Overloading & Constructor Overloading

Method Overloading

- **Methods** of the **same name** can be declared in the same class, as long as they have **different sets of parameters**.

```
public return_type method_name () {
...
}

public return_type method_name (parameter_1) {
...
}

public return_type method_name (parameter_1, parameter_2) {
...
}
```

Method Overloading Example

SumCalculator.java

```java
public class SumCalculator {

    private int sum(int a, int b){
        return a+b;
    }

    private int sum(int a, int b, int c){
        return a+b+c;
    }

    public static void main(String[] args) {
        SumCalculator myCal = new SumCalculator();
        System.out.println("Calling first sum method: " + myCal.sum( a: 4, b: 7));
        System.out.println("Calling second sum method: " + myCal.sum( a: 1, b: 2, c: 3));
    }
}
```

[Question]
how can Java knows which method to call when they have the same name?

Output:

```
Calling first sum method: 11
Calling second sum method: 6
```

MethorOverload.java

```
1  // Fig. 6.10: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public static void main(String[] args)
8     {
9        System.out.printf("Square of integer 7 is %d%n", square(7));
10       System.out.printf("Square of double 7.5 is %f%n", square(7.5));
11    }
12
13    // square method with int argument
14    public static int square(int intValue)
15    {
16       System.out.printf("%nCalled square with int argument: %d%n",
17          intValue);
18       return intValue * intValue;
19    }
20
21    // square method with double argument
22    public static double square(double doubleValue)
23    {
24       System.out.printf("%nCalled square with double argument: %f%n",
25          doubleValue);
26       return doubleValue * doubleValue;
27    }
28 } // end class MethodOverload
```

Output:

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

[Question]
In this example, the number of parameters of the square method is the same, then how can Java know which method should be used?

Constructors Overloading

- As we know, a **constructor** to **specify how objects of a class** should be **initialized**.
- A class with several **overloaded constructors** that enable objects of that class to be **initialized in different ways**.
- To overload constructors, simply provide multiple constructor declarations with **different parameters**.
- Three **overloaded constructors** having **different parameters lists**:

```
public class School {
        public School() {
                ...
        }
        public School(String name) {
                ...
        }
        public School(String name, int year) {
                ...
        }

}
```

Zero parameter Constructor

One parameter Constructor

Two parameter Constructor

Overloaded Constructors Example

Account.java

```java
public class Account {
    private String name;
    private int id;

    public Account(){
        name = "No Name";
        id = -1;
    }

    public Account(String personName){
        name = personName;
        id = -1;
    }

    public Account(int personID){
        name = "No Name";
        id = personID;
    }

    public Account(String personName, int personID){
        name = personName;
        id = personID;
    }

    public String getName() {
        return name;
    }

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

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

Overloaded Constructors Example

EdusoftSystem.java

```
public class EdusoftSystem {
    public static void main(String[] args) {
        Account emptyAcount =  new Account();
        Account tomAccount =  new Account( personName: "Tom");
        Account onlyIDAccount =  new Account( personID: 12345);
        Account JackAccount =  new Account( personName: "Jack",  personID: 45342);

        System.out.printf("Student 1: %s | %d\n",emptyAcount.getName(),emptyAcount.getId());
        System.out.printf("Student 2: %s | %d\n",tomAccount.getName(),tomAccount.getId());
        System.out.printf("Student 3: %s | %d\n",onlyIDAccount.getName(),onlyIDAccount.getId());
        System.out.printf("Student 4: %s | %d\n",JackAccount.getName(),JackAccount.getId());
    }
}
```

Output:

```
Student 1: No Name | -1
Student 2: Tom | -1
Student 3: No Name | 12345
Student 4: Jack | 45342
```

Reuse constructors with "this"

- **"this"** keyword in constructor will refer back to itself to reuse an specific constructor.
- Help to make the code neater.

Before

```
public Account(){
    name = "No Name";
    id = -1;
}

public Account(String personName){
    name = personName;
    id = -1;
}

public Account(int personID){
    name = "No Name";
    id = personID;
}

public Account(String personName, int personID){
    name = personName;
    id = personID;
}
```

After

```
public Account(){
    this( personName: "No Name",  personID: -1);
}

public Account(String personName){
    this(personName,  personID: -1);
}

public Account(int personID){
    this(  personName: "No Name", personID);
}

public Account(String personName, int personID){
    name = personName;
    id = personID;
}
```

Final Keyword

“This cannot be changed!”

Final Keyword

- **“Final”** Indicates **“This cannot be changed”**.
- **Final** can be used: for attribute variables, methods and for a class.
- However, in this course, we mostly use final for attribute variable to have them as **constants**.

Constants (keyword final)

- A constant is a variable which **cannot have its value** changed after declaration.
- It uses the '**final**' keyword.
- Syntax:

Global constant:

accessModifier **final** dataType variableName = value;

Class constant:

accessModifier **static final** dataType variableName = value;

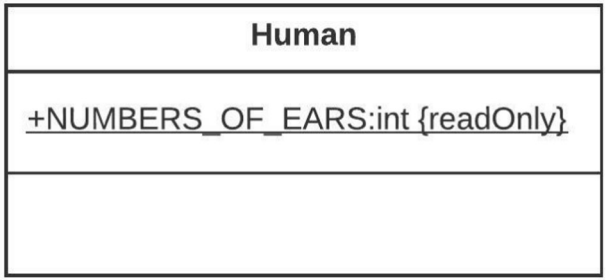
Constant Examples

```
//global constant, outside of a class
public final double PI = 3.14;
```

```
//class constant within a class
public class Human {
   public static final int NUMBER_OF_EARS = 2;
}

//accessing a class constant by class name
int ears = Human.NUMBER_OF_EARS;
```

UML for Human Class



Previously,

- A lottery **ticket** is **$4**.
- A ticket having **6** different numbers (from **1 to 49**) (Can be repeated)
- On a Saturday, they draw the lottery, and the winning numbers are:

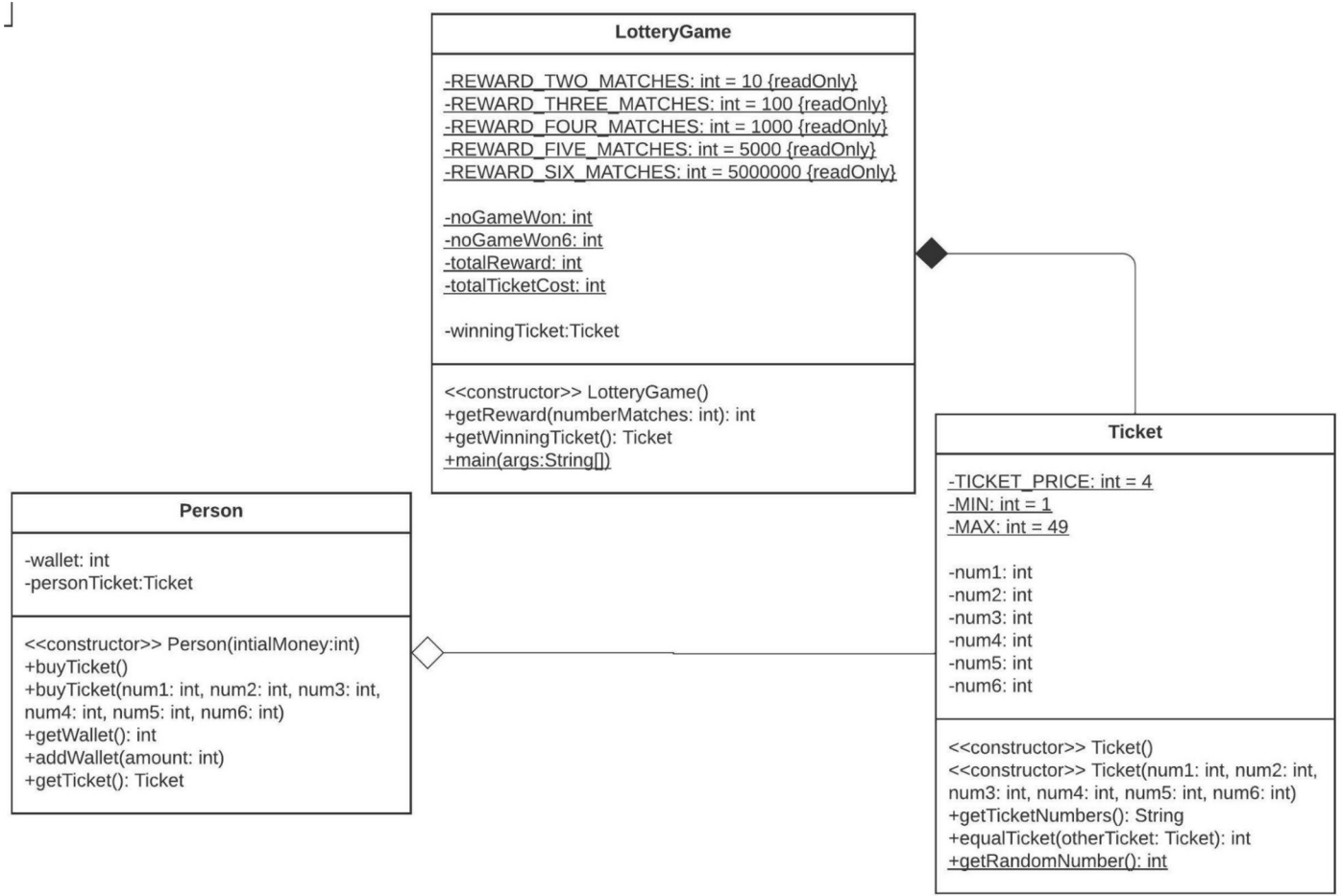
11, 43, 24, 30, 60, 43

- Match at each position:
  - Match **One or Two** numbers to get a small prize. (**$10**)
  - Match **Three** numbers to get a small prize. (**$100**)
  - Matching **Four** numbers gets a bigger prize. (**$1000**)
  - Matching **Five** is even bigger. (**$5000**)
  - Matching **ALL SIX** of the numbers you might win millions. (**$5 million** in cash)
- In the example, we got matches at position 1, 3, 4, 6 (4 numbers) = $1000
- **Homework Task:**
  - Write a simple program allows you to buy a ticket with six random numbers, and generate the winning numbers and return what kind of prize you won (one game).
  - **Imagine you buy up to 100,000 tickets, can you figure out if you actually profit or loss in a long run?**

Homework Coding for this week

- This week, you have learned enough to build a complete lottery 649 game!!!
- You should practice:
  - Build **multiple classes** Java project (Person class, Ticket class, LotteryGame class,...)
  - Use **if…else** to check different user inputs.
  - Use **loop (while or for loop)** to let user **play any amount of games** and display **winning or losing statistics**.
  - Use **keyword "final"** variables to **store constants** of the game (reward money, the price of the ticket, …)
  - Use keyword **"static"** to create **helper class methods** to make your code look neater.
  - Create an **UML class diagram** with https://www.lucidchart.com/pages/landing/uml-diagram-software

Here is an UML of Lottery Game Example



To make things easier for you,

- **Option 1**: You can choose to build the whole things by yourself (freedom).
- **Option 2**: You can download a skeleton/template of my Java projects:
  - You have **the completed LotteryGame class file** with the main method.
  - But you **have to complete all attributes and methods** of the **Person class** and **Ticket class**.
  - So that without changing anything of LotteryGame class file, the project will execute fine.
  - The given project should have the same UML I showed before (using that as a hint to complete the project).
  - Read the comments in the files provided for you.

Output Example

```
Starting Money: $1000
How many games do you want to play?
1
Do you want to pick your own ticket numbers (true/false)?
true
Enter 6 numbers of your ticket (1-49):
32 12 5 23 45 18
You have picked the ticket: 32 12 5 23 45 18
The winning ticket is: 20 26 32 18 2 1
Your ticket has matched 0 number(s)
You have won $0
Ending Money: $996
```

```
Starting Money: $1000
How many games do you want to play?
1
Do you want to pick your own ticket numbers (true/false)?
false
You have picked the ticket: 14 4 46 20 47 42
The winning ticket is: 14 44 38 30 39 4
Your ticket has matched 1 number(s)
You have won $10
Ending Money: $1006
```

```
Starting Money: $1000
How many games do you want to play?
100
The number of games won any money: 13
The number of games matched all 6 numbers: 0
Total money won: $130
Total ticket cost: $400
Profit/Loss (won-cost): $-270
Ending Money: $730
```

```
Starting Money: $1000
How many games do you want to play?
100000
The number of games won any money: 11854
The number of games matched all 6 numbers: 0
Total money won: $120610
Total ticket cost: $400000
Profit/Loss (won-cost): $-279390
Ending Money: $-278390
```

Submission for Advance 649 Lottery Homework

- The submission file (one single zip file) should contains:
  - Screenshots of the interaction outputs (Like my examples).
  - All *.java files of the projects. (80 points)
  - A readme txt file to explain about: (20 points)
    - Which option do you choose (option 1 or 2)?
    - Is there any interesting new feature?
- The file must be submitted on the blackboard.
- Bonus Tasks: (extra 30 points)
  - An UML image for class diagrams for your project (if you choose option 1).

Recap

- **Casting** types.
- Control flow statements:
  - Decision making statements:
    - **If**
    - **If...else**
    - **Switch**
  - Loop statements:
    - **While**
    - **Do while**
    - **For**
  - Jump statements:
    - **Break** statement
    - **Continue** statement
- **Static** Keyword
  - Static Method
  - Static Variable
- **Overloading**
  - Method Overloading
  - Constructor Overloading
- **Final** Keyword
  - Constant Variable

Thank you for your listening!

“One who never asks
Either knows everything or nothing”

Malcolm S. Forbes