5.2 Information Hiding and Encapsulation

- Cornerstones of Object Oriented Programming (OOP)
- Both are forms of abstraction

<u>Information hiding</u>

- protect data inside an object
- do not allo[redacted] access

<u>Encapsulation</u>

- Use classes and objects
- Objects include both data items and methods to act on the data

`public` and `private`

`public`

- any other class or program can [redacted] access or change a public instance variable
- any other class or program can [redacted] a public method

`private`

- only a method in the same class can access a private instance variable
- only a method in the same class can invoke a public method

Instance variables should be [redacted] to prevent inappropriate changes.

public and private

```
// Listing 5.7. A Class with Private Instance Variable

public class SpeciesSecondTry
{
    private String name;        // public ==> private
    public int population;       // public ==> private
    public double growthRate;      // public ==> private

public class SpeciesSecondTryTest
{
        ……SpeciesSecondTry speciesOfTheMonth;
        // ??? Valid or invalid
        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
```

Listing 5.6 A Method that Has a Parameter ( ) - SpeciesSecondTry.java

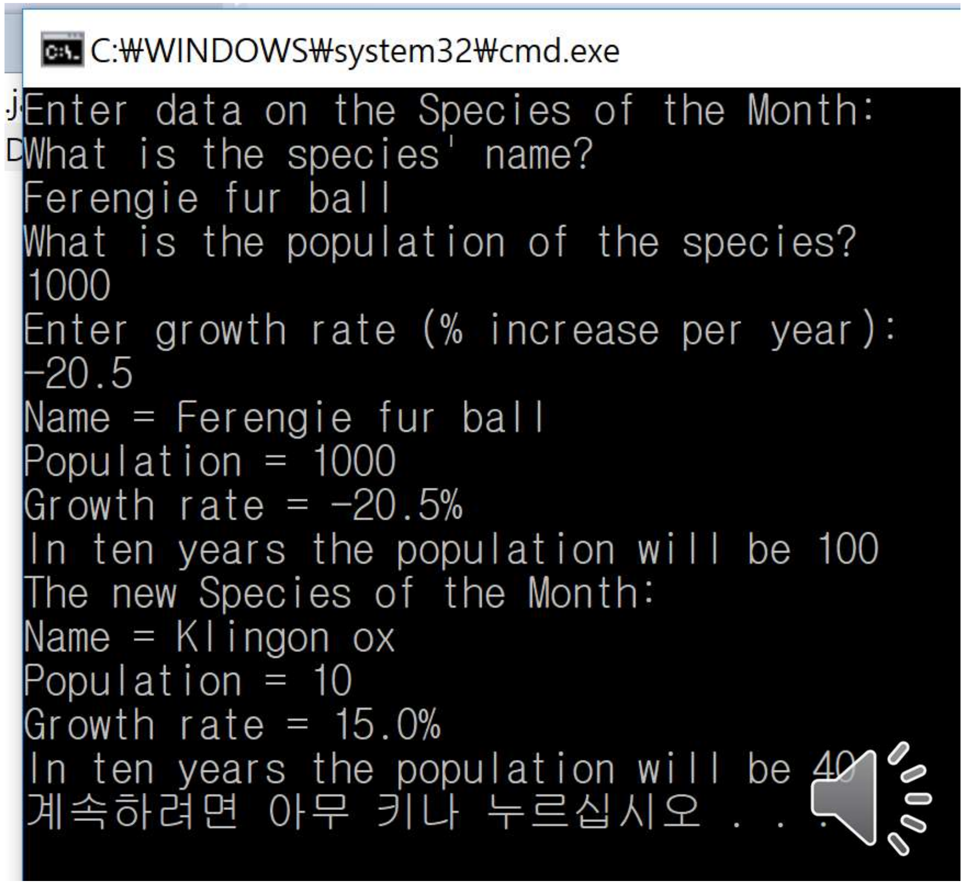
```java
import java.util.Scanner;

public class SpeciesSecondTry
{
    public String name;
    public int population;
    public double growthRate;

    public void readInput( )
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine( );

        System.out.println("What is the population of the species?");
        population = keyboard.nextInt( );

        System.out.println("Enter growth rate (% increase per year):");
        growthRate = keyboard.nextDouble( );
    }
```

```
  public void writeOutput( )
   {
        System.out.println("Name = " + name);
        System.out.println("Population = " + population);
        System.out.println("Growth rate = " + growthRate + "%");
   }

   /**
    Returns the projected population of the calling object
    after the specified number of years.
   */
   public int predictPopulation(int years) // int years <== 10 years
   {
      int result = 0;
       double populationAmount = population;
       int count = years;
       while ((count > 0) && (populationAmount > 0))
       {
           populationAmount = (populationAmount +
                         (growthRate / 100) * populationAmount);
           count--;
       }

       if (populationAmount > 0)
           result = (int)populationAmount;

       return result;
   }
}
```

```
public void writeOutput( )        (X)
   {
        System.out.println("Name = " + name);
        System.out.println("Population = " + population);
        System.out.println("Growth rate = " + growthRate + "%");
   }

   /**
    Returns the projected population of the calling object
    after the specified number of years.
   */
   public int projectedPopulation(int years)    // int years <== 10 years
   {
       double populationAmount = population;
       int count = years;
       while ((count > 0) && (populationAmount > 0))
       {
           populationAmount = (populationAmount +
                         (growthRate/100) * populationAmount);
           count--;
       }
       if (populationAmount > 0)
           return (int)populationAmount;
       else
           return 0;      // we will give an even vetter version of the class later
   }
}
```

Listing 5.7 Using a Method that has a Parameter ( ) - SpeciesSecondTryDemo.java

```
/**
 Demonstrates the use of a parameter
 with the method projectedPopulation.
*/
public class SpeciesSecondTryDemo
{
    public static void main(String[] args)
    {
        SpeciesSecondTry speciesOfTheMonth = new SpeciesSecondTry( );

        System.out.println("Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput( );
        speciesOfTheMonth.writeOutput( );
        int futurePopulation = speciesOfTheMonth.predictPopulation(10);
        System.out.println("In ten years the population will be " +
                          futurePopulation);
        //Change the species to show how to change
        //the values of instance variables:
                    speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput( );
        System.out.println("In ten years the population will be " +
                     speciesOfTheMonth.predictPopulation(10));
    }
}
```



```
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (% increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In ten years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In ten years the population will be 40
계속하려면 아무 키나 누르십시오 . . .
```

Listing 5.8 A Class with Private Instance Variables - SpeciesThirdTry.java

```java
import java.util.Scanner;

public class SpeciesThirdTry
{
    private String name; // public ==> private
    private int population; // public ==> private
    private double growthRate; // public ==> private

    public void readInput( )
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine( );

        System.out.println("What is the population of the species?");
        population = keyboard.nextInt( );

        System.out.println("Enter growth rate (% increase per year):");
        growthRate = keyboard.nextDouble( );
    }
```

```
public void writeOutput( )
  {
      System.out.println("Name = " + name);
      System.out.println("Population = " + population);
      System.out.println("Growth rate = " + growthRate + "%");
  }
```

```
 /**
     Precondition: years is a nonnegative number.
     Returns the projected population of the calling object
     after the specified number of years.
    */

    public int predictPopulation(int years)
    {
                          int result = 0;
        double populationAmount = population;
        int count = years;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                          (growthRate / 100) * populationAmount);
            count--;
        }
        if (populationAmount > 0)
            result = (int)populationAmount;

        return result;
    }
}
```

SpeciesThirdTryDemo.java

```
/**
 Demonstrates a classhaving private instance variables.
*/
public class SpeciesThirdTryDemo
{
    public static void main(String[] args)
    {
        SpeciesThirdTry speciesOfTheMonth = new SpeciesThirdTry( );

        System.out.println("Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput( );
        speciesOfTheMonth.writeOutput( );
        int futurePopulation = speciesOfTheMonth.predictPopulation(10);
        System.out.println("In ten years the population will be " +
                            futurePopulation);

//      speciesOfTheMonth.name = "Klingon ox";
//      speciesOfTheMonth.population = 10;
//      speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput( );
        System.out.println("In ten years the population will be " +
                            speciesOfTheMonth.predictPopulation(10));
    }
}
```

Accessors and Mutators

- ***accessor methods***—[redacted] methods that allow instance variables to be read

✓

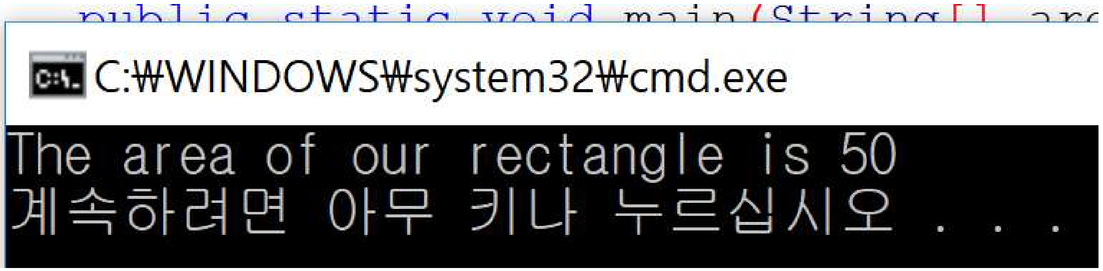
- ***mutator methods***—[redacted] methods that allow instance variables to be modified
  - » Mutator methods should always check to make sure that changes are appropriate.
  - » Providing mutator methods is much better than making instance variables public because a method can check to make sure that changes are appropriate.

Listing 5.9 A Class of Rectangles - Rectangle.java

A Demonstration of why instance Variables should be Private

```
/**
 Class that represents a rectangle.
*/
public class Rectangle
{
    private int width;
    private int height;
    private int area;

    public void setDimensions(int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
             area = width * height;
    }

    public int getArea()
    {
                          return area;
             }
}
```

```
public class RectangleDemo
{
    public static void main(String[] args)
    {
        Rectangle box = new Rectangle( );
        box.setDimensions(10, 5);
        System.out.println("The area of our rectangle is " + box.getArea( ));
    }
}
```

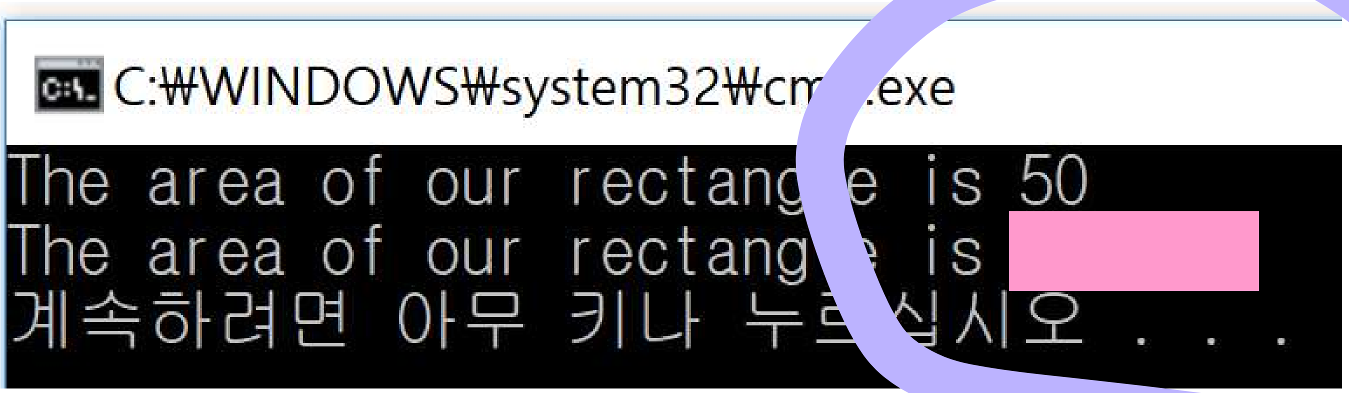


```
The area of our rectangle is 50
계속하려면 아무 키나 누르십시오 . . .
```

A Demonstration of why instance Variables should be Private

```
/**
 Class that represents a rectangle.
*/
public class RectanglePublic
{
    public int width;
    private int height;
    private int area;

    public void setDimensions(int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }

    public int getArea()
    {
                          return area;
            }
}
```

```
public class RectanglePublicDemo
{
    public static void main(String[] args)
    {
        RectanglePublic box = new RectanglePublic( );
        box.setDimensions(10, 5);
        System.out.println("The area of our rectangle is " + box.getArea( ));

        box.width = 6;
        System.out.println("The area of our rectangle is " + box.getArea( ));
    }
}
```



```
The area of our rectangle is 50
The area of our rectangle is
계속하려면 아무 키나 누르십시오 . . .
```

Listing 5.10 Another Class of Rectangles – Rectangle2.java different implementation

```
/**
 Another class that represents a rectangle.
 saves you both execution time and memory requirements
*/
public class Rectangle2
{
    private int width;
    private int height;

    public void setDimensions(int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
    }

    public int getArea()
    {
                      return width * height;

   }
}
```

LISTING 5.11 A Class with Accessor and Mutator Methods - SpeciesFourthTry.java

```
// LISTING  5.9 A Class with Accessor and Mutator Methods
// Yes, we will define an even better version of this class later
public class SpeciesFourthTry
{
    private String name;
    private int population;
    private double growthRate;

    public void readInput( )
    {   .........  }

    public void writeOutput( )
    {  ...........  }

    /**Precondition: years is a nonnegative number.
     Returns the projected population of the calling object
     after the specified number of years.*/
    public int predictPopulation(int years)
    {   ............      }
// mutator
   public void setSpecies(String newName,
                     int newPopulation, double newGrowthRate)
   {
       name = newName;
       // an accessor(mutator) method  can check to make sure that
       // instance variables are not set to improper values.
       if (newPopulation >= 0)
           population = newPopulation;
       else
       { System.out.println("ERROR: using a negative population.");
           System.exit(0);
       }
       growthRate = newGrowthRate;
   }
   public String getName( )  // Accessor
   {
       return name;
   }
   public int getPopulation( )    // Accessor
   {
       return population;
   }
   public double getGrowthRate( )     // Accessor
   {
       return growthRate;
   }
}
```

LISTING 5.12 using a Mutator method - SpeciesFourthTryDemo.java

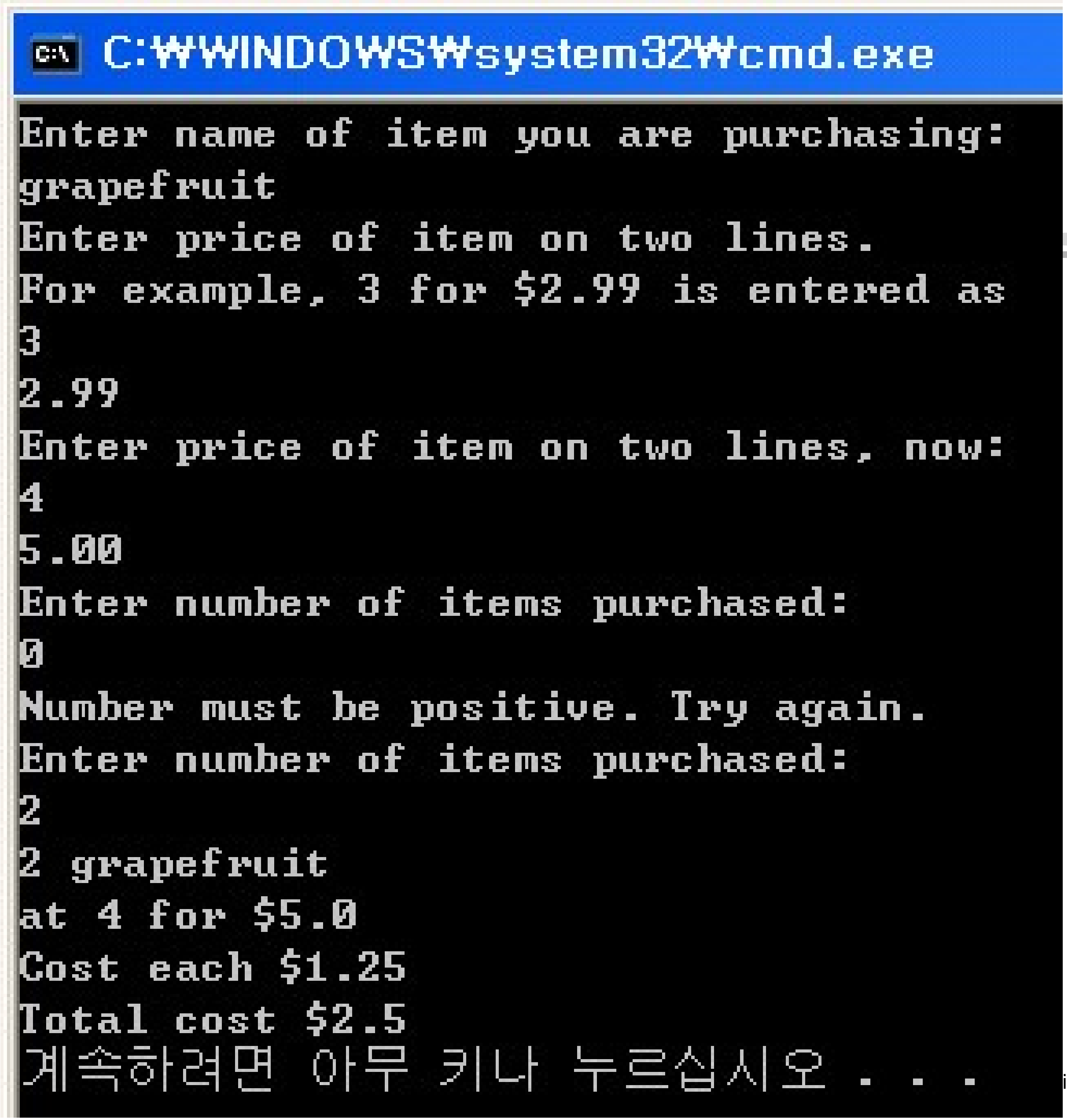
```
/**
 Demonstrates the use of the mutator method setSpecies.
*/
public class SpeciesFourthTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFourthTry speciesOfTheMonth = new SpeciesFourthTry( );

        System.out.println("Enter number of years to project:");
        Scanner keyboard = new Scanner(System.in);
        int numberOfYears = keyboard.nextInt( );

        .............

        //Change the species to show how to change
        //the values of instance variables:
                          speciesOfTheMonth.setSpecies("Klingon ox", 10, 15);
```

LISTING 5.13 Purchase Class - Purchase.java

```
/**
 Class for the purchase of one kind of item, such as 3 oranges.
 Prices are set supermarket style, such as 5 for $1.25.
*/
public class Purchase
{
    private String name;
    private int groupCount; //Part of price, like the 2 in 2 for $1.99.
    private double groupPrice;
                  //Part of price, like the $1.99 in 2 for $1.99.
    private int numberBought; //Total number being purchased.

    public void setName(String newName)
    {
        name = newName;
    }
    /**
     Sets price to count pieces for $costForCount.
     For example, 2 for $1.99.
    */
    public void setPrice(int count, double costForCount)
    {………
public void setNumberBought(int number)
   {
      .............
   }
   /**
    Gets price and number being purchased from keyboard.
   */
   public void readInput( )
   {
    .............
    }
/**
    Outputs price and number being purchased to screen.
   */
   public void writeOutput( )
   {
   ............
   }
   public String getName( )
   {
      return name;
   }
   public double getTotalCost( )
   {
      return ((groupPrice/groupCount)*numberBought);
   }
   public double getUnitCost( )
   {
      return (groupPrice/groupCount);
   }
   public int getNumberBought( )
   {
      return numberBought;
   }
}
```

LISTING 5.14 use of the Purchase Class - Purchase.java

```
public class PurchaseDemo
{
   public static void main(String[] args)
   {
      Purchase oneSale = new Purchase( );

      oneSale.readInput( );
      oneSale.writeOutput( );
      System.out.println("Cost each $" + oneSale.getUnitCost( ));
      System.out.println("Total cost $" + oneSale.getTotalCost( ));
   }
}
```



```
Enter name of item you are purchasing:
grapefruit
Enter price of item on two lines.
For example, 3 for $2.99 is entered as
3
2.99
Enter price of item on two lines, now:
4
5.00
Enter number of items purchased:
0
Number must be positive. Try again.
Enter number of items purchased:
2
2 grapefruit
at 4 for $5.0
Cost each $1.25
Total cost $2.5
계속하려면 아무 키나 누르십시오 . . .
```

LISTING 5.15 Methods calling Other Methods- Oracle.java

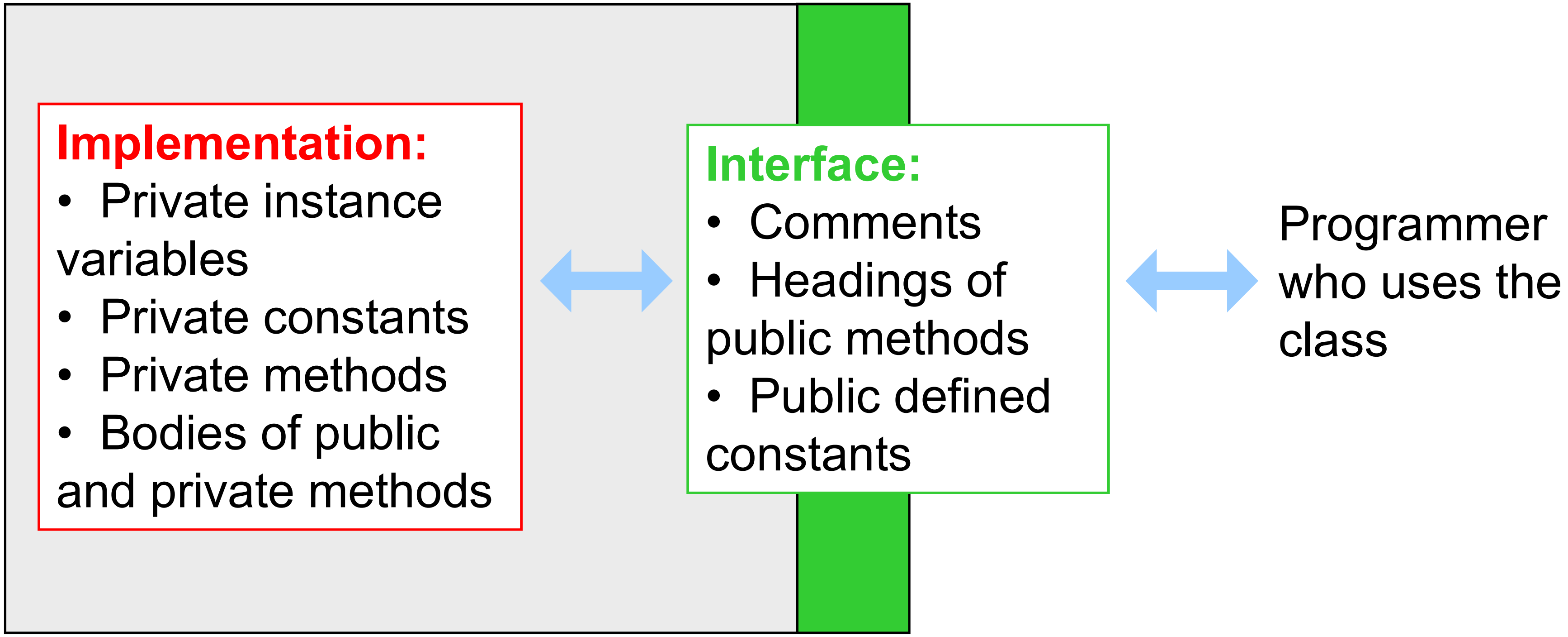
```
import java.util.Scanner;

public class Oracle
{
    private String oldAnswer = "The answer is in your heart.";
    private String newAnswer;
    private String question;

    public void chat( )
    {
        System.out.print("I am the oracle. ");
        System.out.println("I will answer any one-line question.");
        Scanner keyboard = new Scanner(System.in);
        String response;
        do
        {
            answer( );
            System.out.println("Do you wish to ask another question?");
            response = keyboard.next( );
        } while (response.equalsIgnoreCase("yes"));
        System.out.println("The oracle will now rest.");
    }
```

```java
    private void answer( )
    {
        System.out.println("What is your question?");
        Scanner keyboard = new Scanner(System.in);
        question = keyboard.nextLine();
        seekAdvice( );
        System.out.println("You asked the question:");
        System.out.println("  " + question);
                          System.out.println("Now, here is my answer:");
        System.out.println("  " + oldAnswer);
        update( );
    }

    private void seekAdvice( )
    {
        System.out.println("Hmm, I need some help on that.");
        System.out.println("Please give me one line of advice.");
        Scanner keyboard = new Scanner(System.in);
        newAnswer = keyboard.nextLine();
        System.out.println("Thank you. That helped a lot.");
    }

    private void update( )
    {
        oldAnswer = newAnswer;
    }
}
```

Encapsulation

- Encapsulation
  - » the process of [redacted] of a class definition that are not necessary to understanding how objects of the class are used.
  - » Encapsulation is a form of information hiding
  - » Encapsulation divides a class definition into two parts, which we will call the [redacted] and [redacted].

A Well-Encapsulated Class Definition (Figure 5.3)



Implementation:

- Private instance variables
- Private constants
- Private methods
- Bodies of public and private methods

Interface:

- Comments
- Headings of public methods
- Public defined constants

Programmer who uses the class

A programmer who uses the class can only access the instance variables indirectly through public methods and constants.

The most important guidelines for defining a well-encapsulated class

- 1) Place a comment before the class definition that describes how the programmer should think about the class data and methods.
- 2) all the instance variable in the class should be marked [redacted]
- 3) provide [redacted] accessor and mutator methods to read and change the data in an object.
- 4) fully specify [redacted] each public method with a comment placed before the method heading
- 5) Make any helping methods [redacted]
- 6) some of the comments in a class definition are part of the user interface, describing how to use the class.
  - » use the /** */ types of comments for [redacted] comments
  - » use the // types of comments for [redacted] comments.
  - » ➔ Ex) Display 2.11

Formalized Abstraction: ADTs

ADT: Abstract data type

- An Object-Oriented approach used by several languages
- A term for *class* implementation
  - » a container for both data items and methods to act on the data
- Implements information hiding and encapsulation
- Provides a public *use*[redacted] so the user knows how to use the class
  - » descriptions, parameters, and names of its methods

Implementation:

- » private[redacted] variables
- » method definitions are usually public but always hidden from the user
- » the user cannot see or change the implementation
- » the user only sees the interface

Sound Complicated?

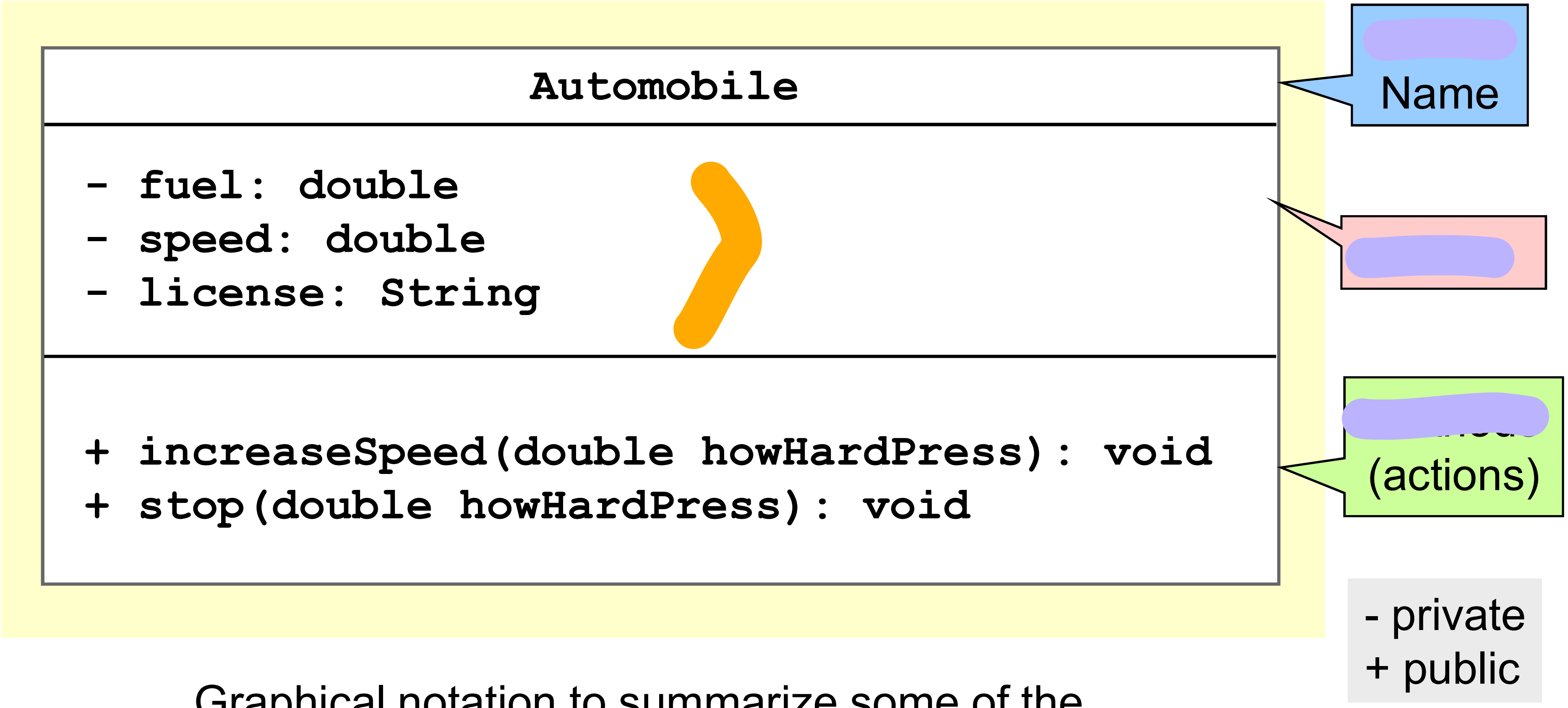
Not really! Just create classes as previously described, except:

- Use the `private` modifier when declaring instance variables
- Do *not* give the user [redacted]
- *Do* give the user [redacted] - a file with just the class and method descriptions and headings
  - » the headings give the names and parameters of the methods
  - » it tells the user how to use the class and its methods
  - » it is all the user needs to know

Automatic Documentation with [redacted]

- [redacted] ✓
  - automatically generate documentation for the user interfaces to your classes.
  - Use web-browser

UML Class Diagrams (Figure 5.4)



Automobile

- fuel: double
- speed: double
- license: String

+ increaseSpeed(double howHardPress): void
+ stop(double howHardPress): void

[redacted] Name

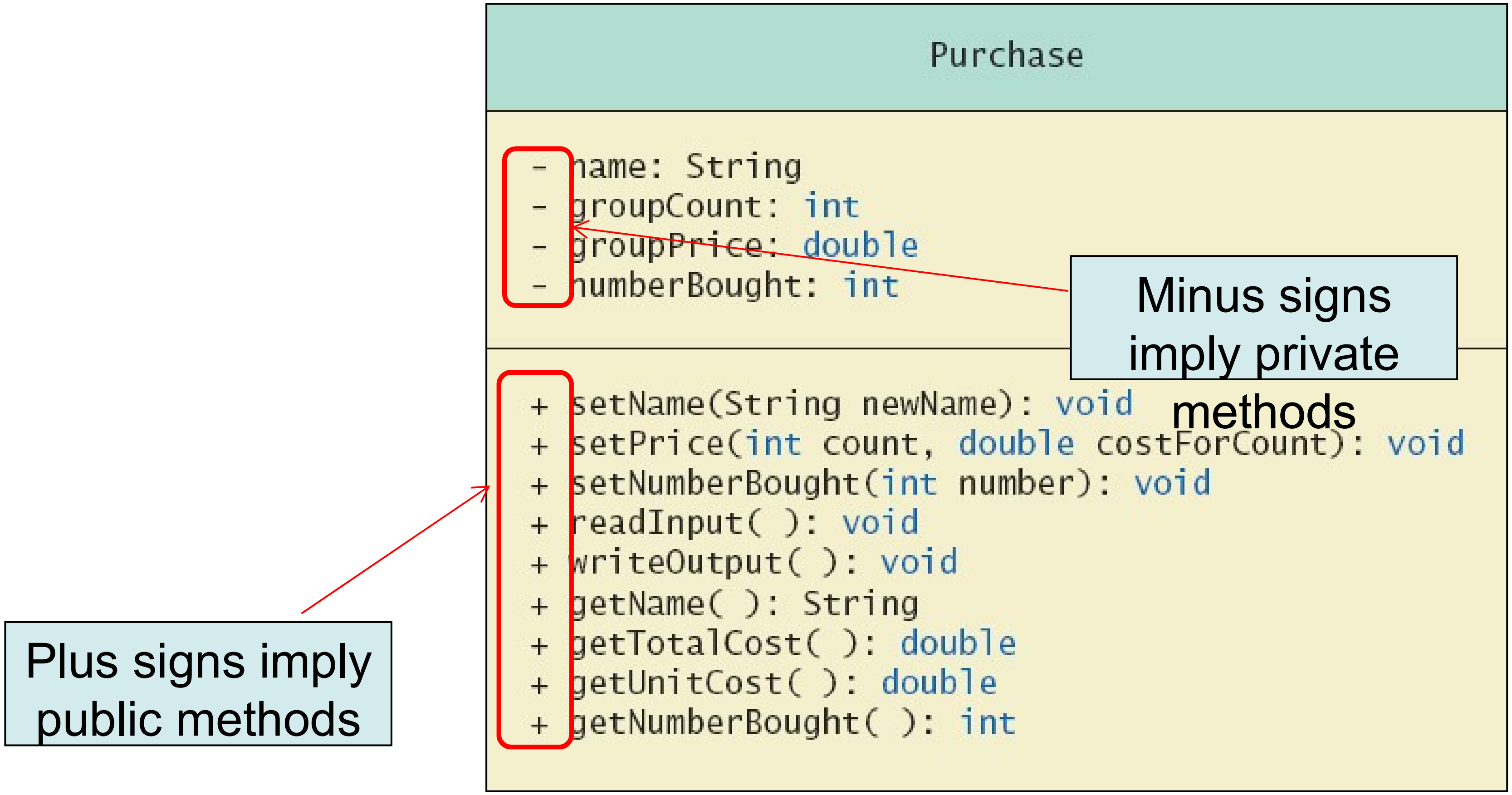
[redacted]

[redacted] (actions)

- private
+ public

Graphical notation to summarize some of the main properties of a class

Figure 5.4 A UML Class Diagram for the Class Purchase (Listing 5.13)



```
Purchase

- name: String
- groupCount: int
- groupPrice: double
- numberBought: int

+ setName(String newName): void
+ setPrice(int count, double costForCount): void
+ setNumberBought(int number): void
+ readInput( ): void
+ writeOutput( ): void
+ getName( ): String
+ getTotalCost( ): double
+ getUnitCost( ): double
+ getNumberBought( ): int
```

Minus signs imply private methods

Plus signs imply public methods

끝

Precondition and Postcondition Comments

- efficient and standard way to tell what a method does
- ***precondition—***states conditions that must be true before method is invoked
- ***postcondition***—tells the effect of a method call
- Example:

```
/**
Precondition: years is a nonnegative number
Postcondition: Returns the projected population
  after the specified number of years
*/
```

- Note that the terms preconditions and postconditions are not always used, particularly if the only postcondition describes the return value of the method.

Assertion Checks

- ***assertion***—statement that should be true if there are no mistakes in the program
- Preconditions and postconditions are examples of assertions.
- Can use assert to see if assertion is true.
- Syntax:

```
assert Boolean_Expression;
```

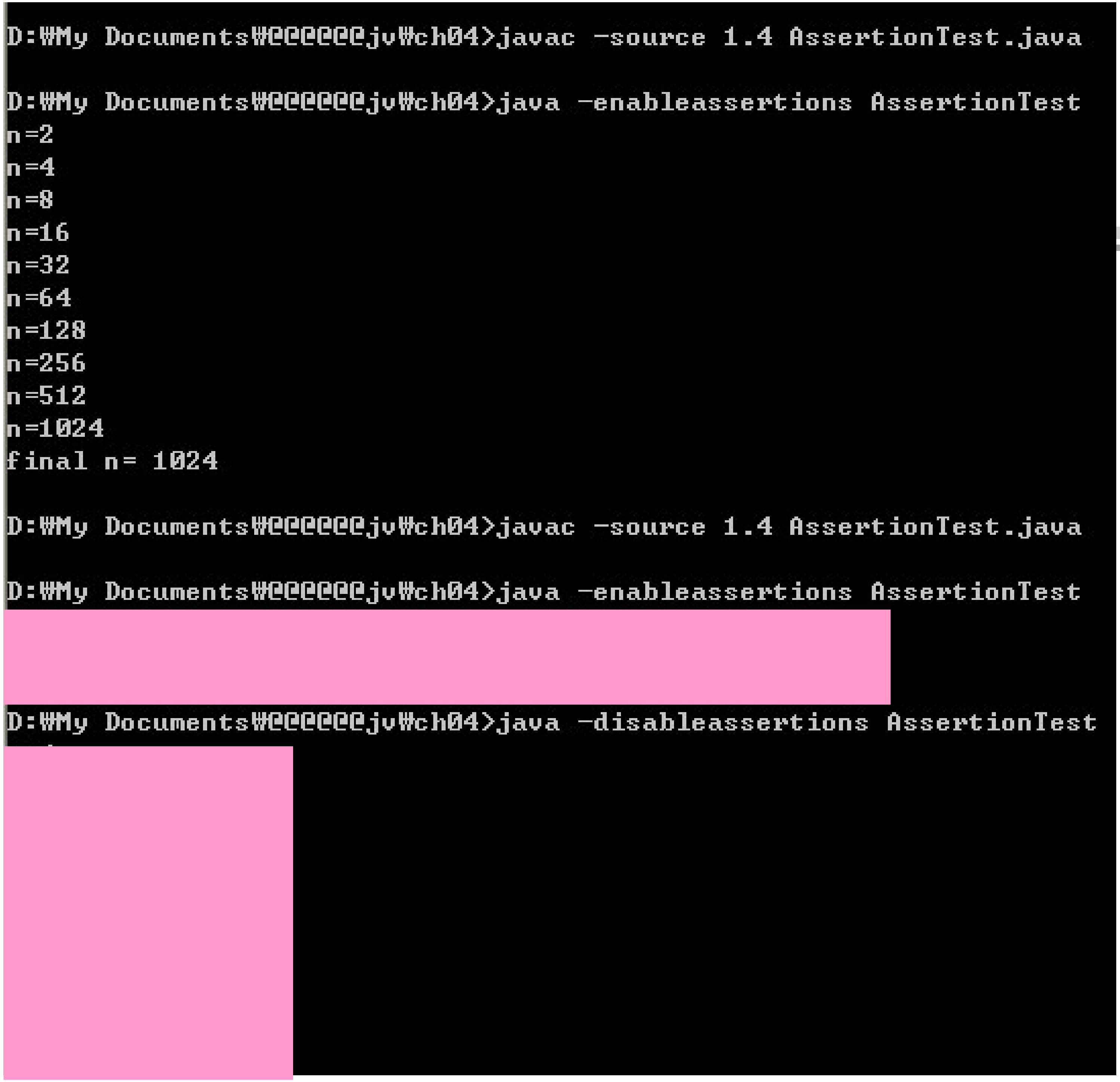
- Example:

```
assert n >= limit;
```

- If assertion is false when checked, the program ends and an error message is printed.
- Assertion checking can be turned on and off.
  - » The exact way to enable or disable assertions depends on your development environment.

Assertion Test

```
public class AssertionTest
{

    public static void main(String[] args)
    {
                          int limit = 1000;
                          // int n = 1;  //뒤면 첫번째 예
                          int n = 2;   //뒤면 두번째 예

                          assert n == 1;
                          while (n <= limit)
                          {
                                       n = 2 * n ;
                                       System.out.println("n="+n);
                          }
                          assert n >= limit;
//      n is the smallest power of 2 >= limit.

        System.out.println("final n= " + n);
    }
}
```



```
D:₩My Documents₩@@@@@@jv₩ch04>javac -source 1.4 AssertionTest.java

D:₩My Documents₩@@@@@@jv₩ch04>java -enableassertions AssertionTest
n=2
n=4
n=8
n=16
n=32
n=64
n=128
n=256
n=512
n=1024
final n= 1024

D:₩My Documents₩@@@@@@jv₩ch04>javac -source 1.4 AssertionTest.java

D:₩My Documents₩@@@@@@jv₩ch04>java -enableassertions AssertionTest
[redacted]

D:₩My Documents₩@@@@@@jv₩ch04>java -disableassertions AssertionTest
[redacted]
```