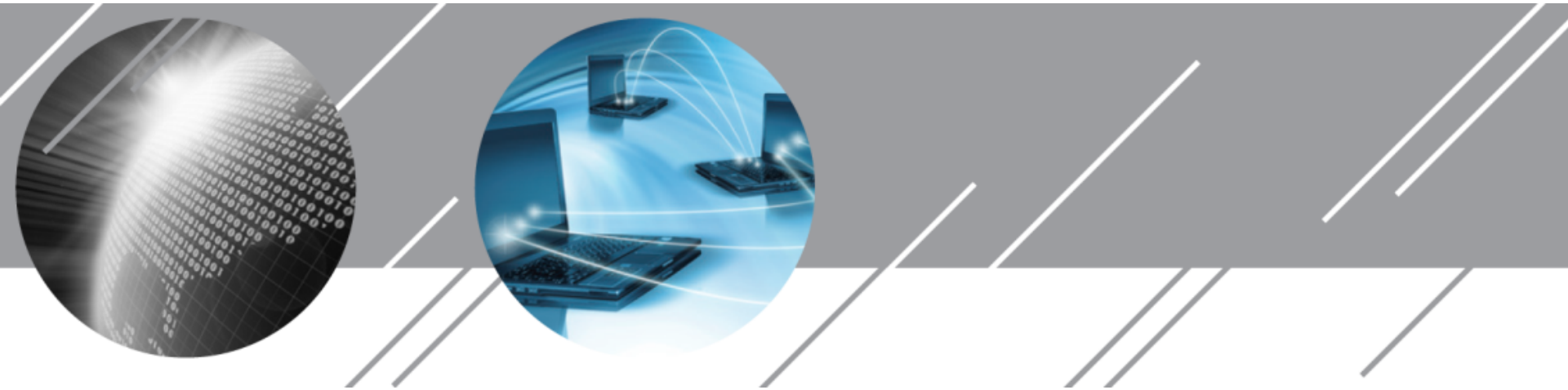
Object Oriented Programming

Introduction to Java

Ch. 5. Classes and Methods (1)



Dept. of Software, Gachon University
Ahyoung Choi, Spring

Introduction : OOP

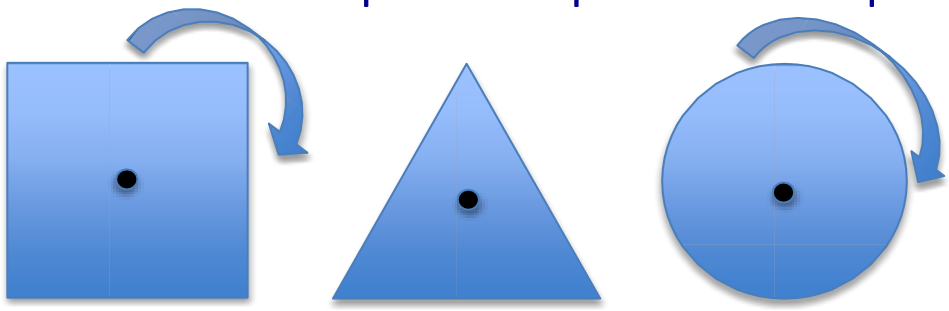
Intro: OOP

- Object-oriented programming (OOP) helps people to organize code and programs
  - How to organize data?
  - How to organize manipulations of data?

- OOP uses classes and objects to get good organization

Why do we need it?

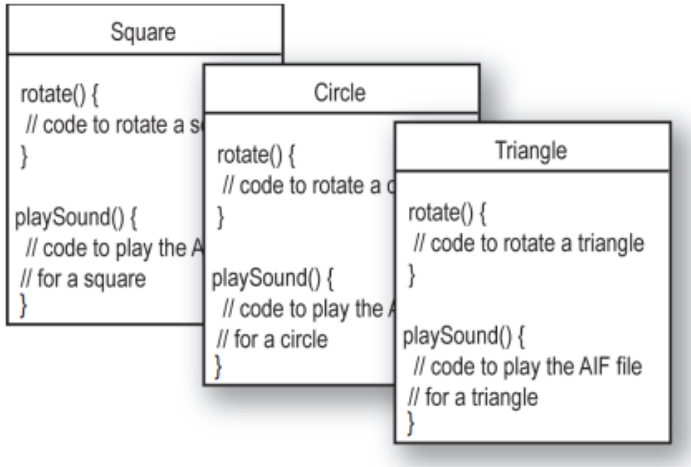
- There will be shapes on a square, a circle, and a triangle. When the user clicks on a shape, the shape will rotate clockwise 360 degree and play an AIF sound file specific to particular shape.



* C version

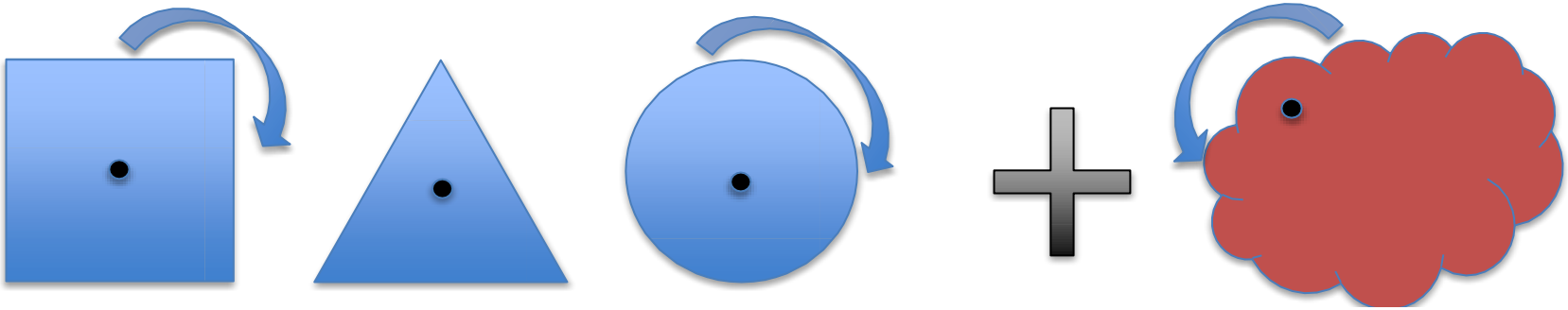
```
rotate(shapeNum) {
  // make the shape rotate 360°
}
playSound(shapeNum) {
  // use shapeNum to lookup which
  // AIF sound to play, and play it
}
```

* Java version



Why do we need it?

- What if new function is added?
  - There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, and play a .hif sound file

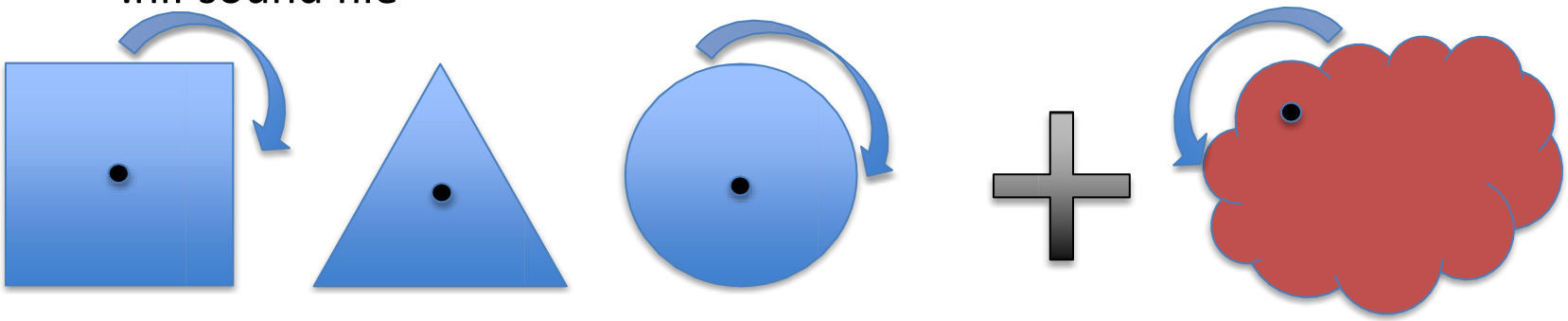


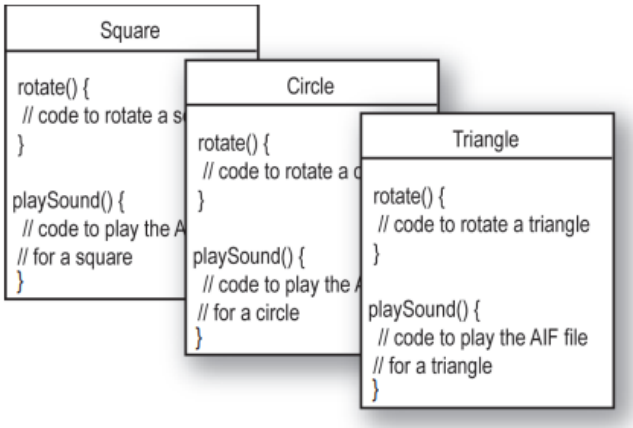
```
playSound(shapeNum) {
   // if the shape is not an amoeba,
      // use shapeNum to lookup which
      // AIF sound to play, and play it
   // else
      // play amoeba .hif sound
}
```

```
rotate(shapeNum, xPt, yPt) {
 // if the shape is not an amoeba,
    // calculate the center point
    // based on a rectangle,
    // then rotate
 // else
    // use the xPt and yPt as
    // the rotation point offset
    // and then rotate
}
```

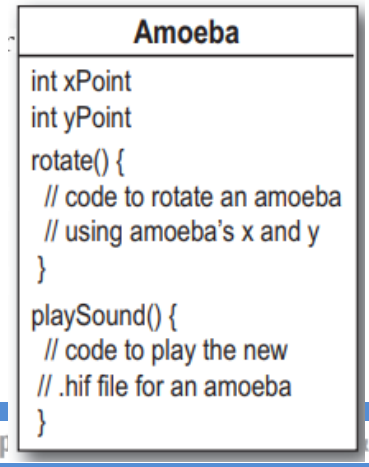
Why do we need it?

- What if new function is added?
  - There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, and play a .hif sound file



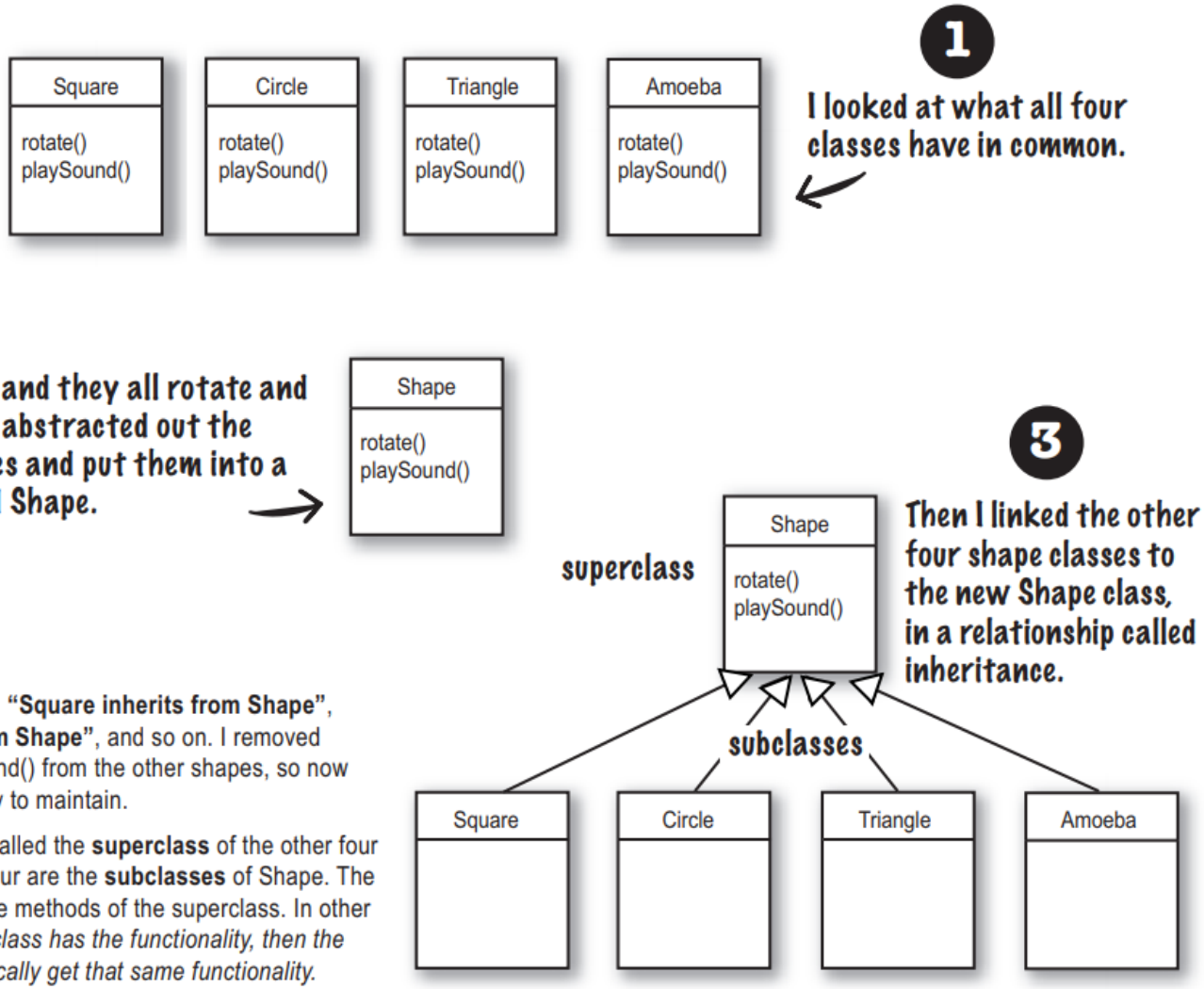


No need to change here!



Why do we need it?

Inheritance!

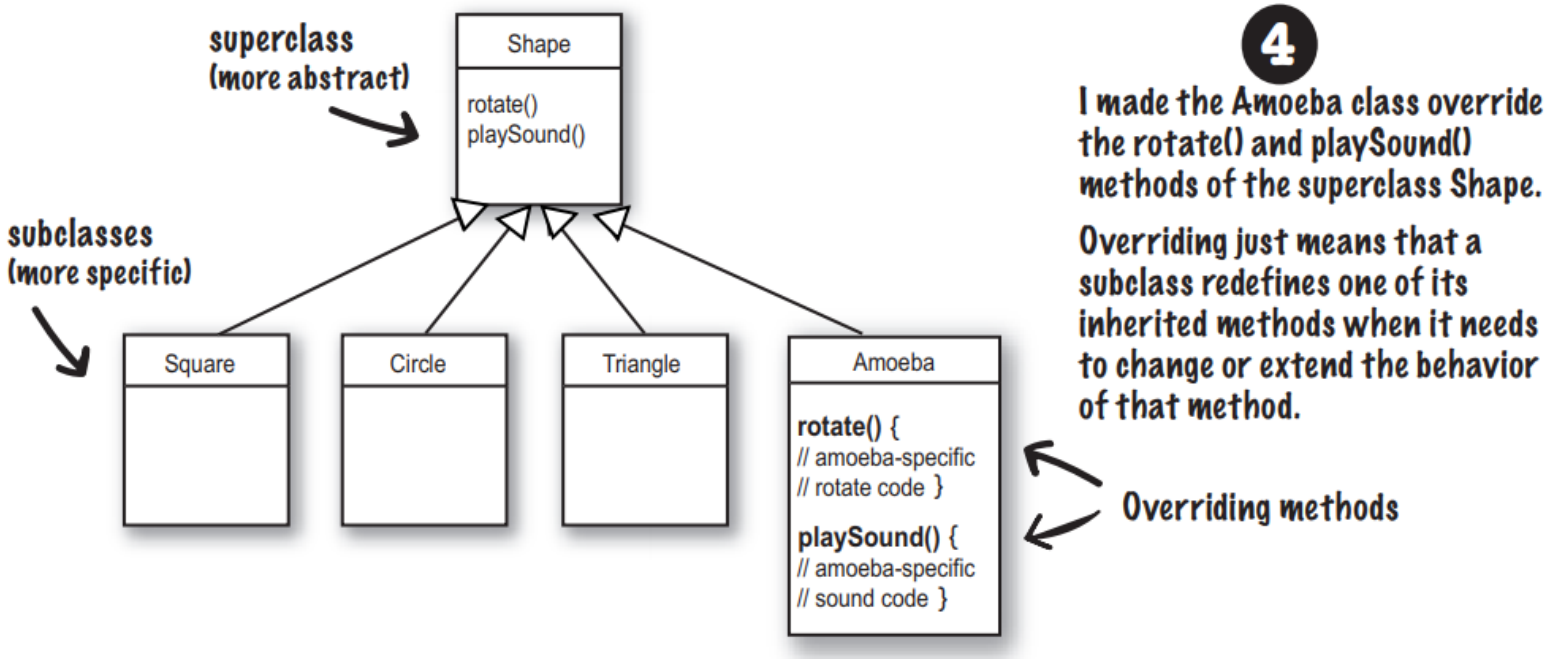


You can read this as, **"Square inherits from Shape"**, **"Circle inherits from Shape"**, and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

Why do we need it?

Overriding!



Reusability

- How does good organization (or usually called “good design”) help you?
  - If I can make it work, it is a good design?
- Good design means better **reusability**
  - You can use part of your program in another program
  - You can use part of your program in a new version
    - You can change only one part if you know other parts are good
  - Others can use part of, or the whole of your program
    - They don’t even have to know the details if they trust you
    - That’s how programmers collaborate

Cont.

- You have seen many program components that you can use without knowing the details
  - Scanner
    - next(), nextLine(), nextInt()
  - String
    - length(), indexOf(), substring(), trim()
- Scanner class has more than 1500 lines of code
  - But you can use it without copying a single line

Cont.

- The rules of reusability
  - Generic design
    - A component (a class in Java) should perform a general function
  - High cohesion (높은 결합력)
    - What's in a class (data and methods) should be closely related to each other
  - Low coupling (느슨한 의존성)
    - Classes should be independent of other classes

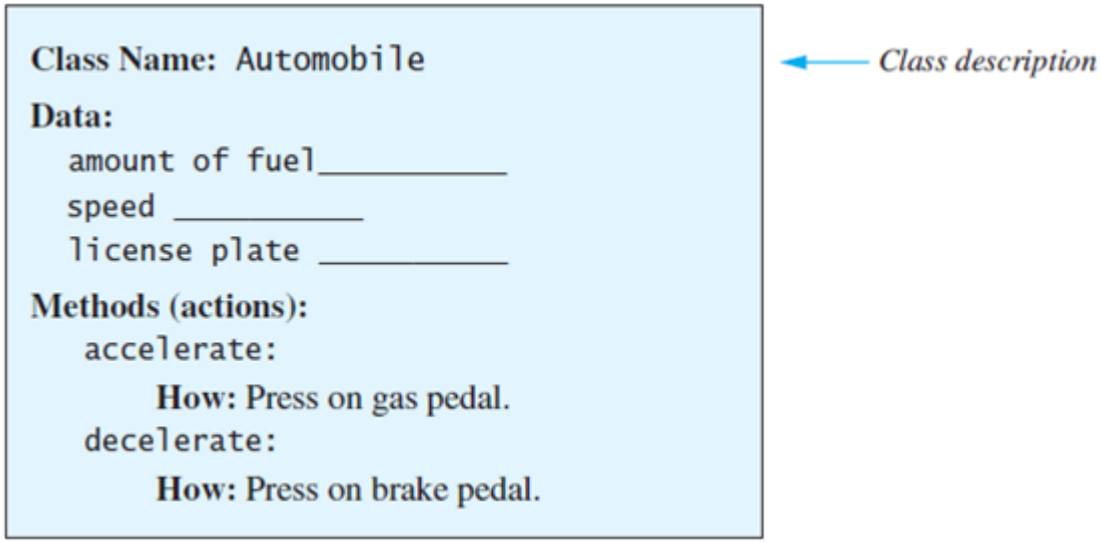
5.1 Class and Method Definition

Class and Method Definitions

- Java program consists of objects
  - Objects of class types
  - Objects that interact with one another
- Program objects can represent
  - Objects in real world
  - Abstractions

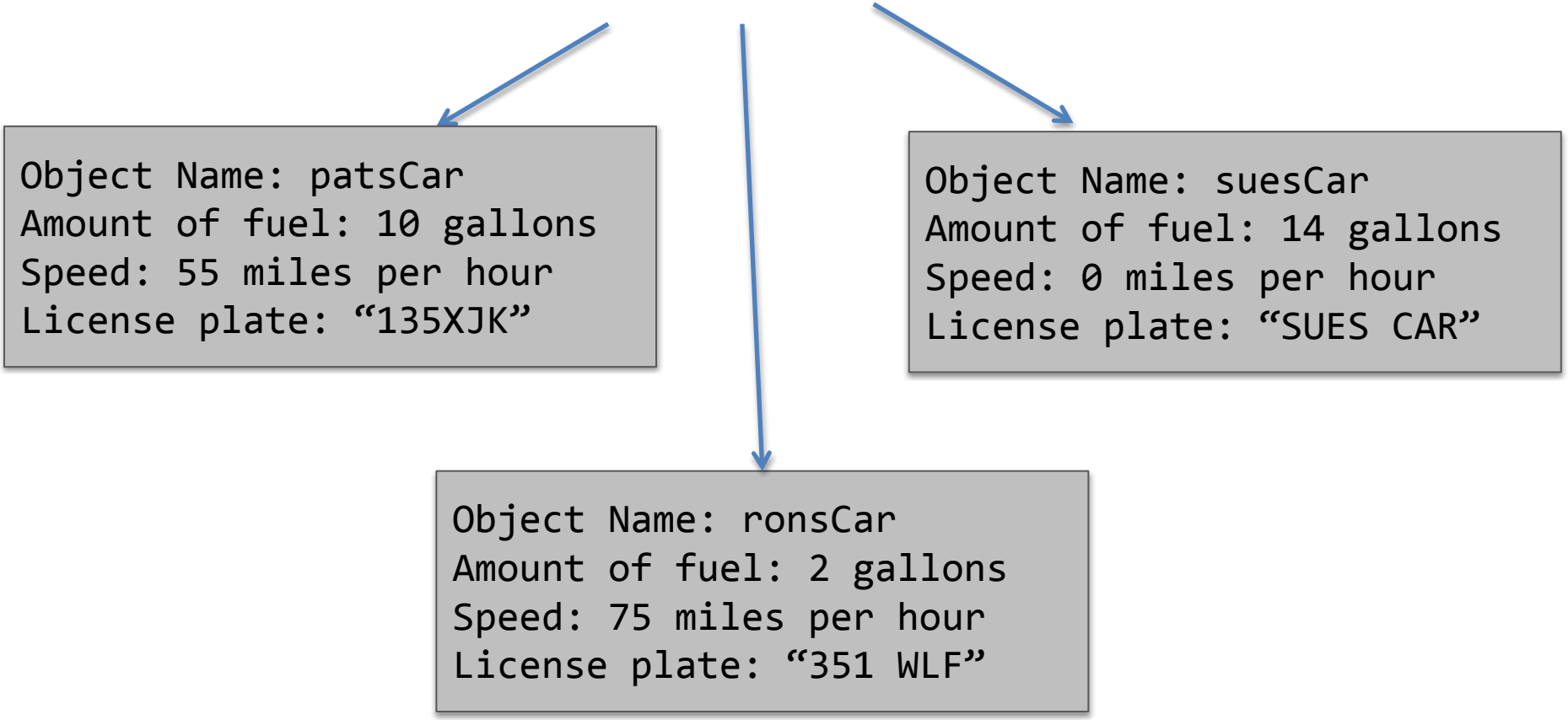
Class and Method Definitions

- Figure 5.1 A class as a blueprint

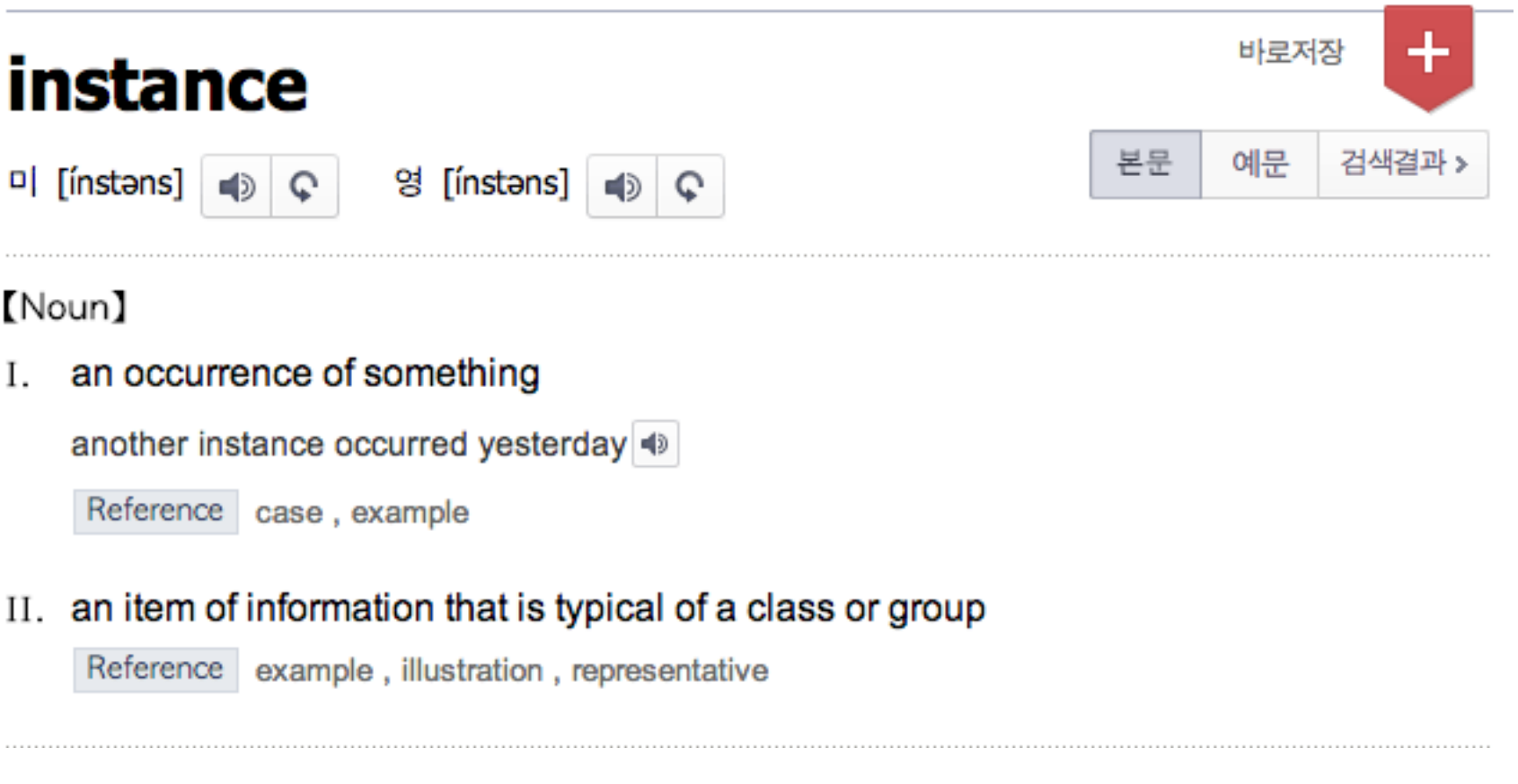


Objects (Instances)

Instances of
the class **Automobile**



Hey..



instance

바로저장

미 [ínstəns] 영 [ínstəns]

본문 | 예문 | 검색결과 >

【Noun】

I. an occurrence of something

another instance occurred yesterday

Reference case , example

II. an item of information that is typical of a class or group

Reference example , illustration , representative

Objects

- **Important!**

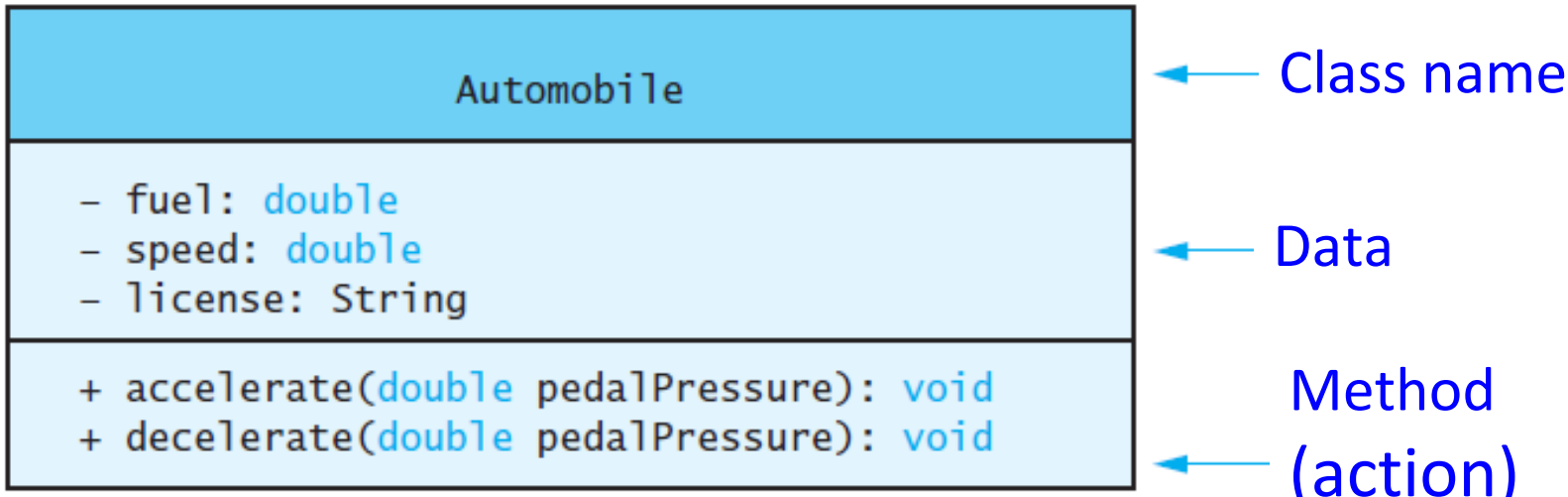
- Classes do not have data; individual objects have data

- Classes specify what kind of data objects have

Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram

FIGURE 5.2 A Class Outline as a UML Class Diagram



Class Student

- UML class specification

Class Name: Student
- Name - Year - GPA - Major - Credits - GPA sum
+ getName + getMajor + printData + increaseYear Action: increase year by 1

- In Java

Class Name: Student
- name: String - year: int - gpa: double - major: String - credits: int - gpaSum: double
+ getName(): String + getMajor(): String + printData(): void + increaseYear(): void

Class Files and Separate Compilation

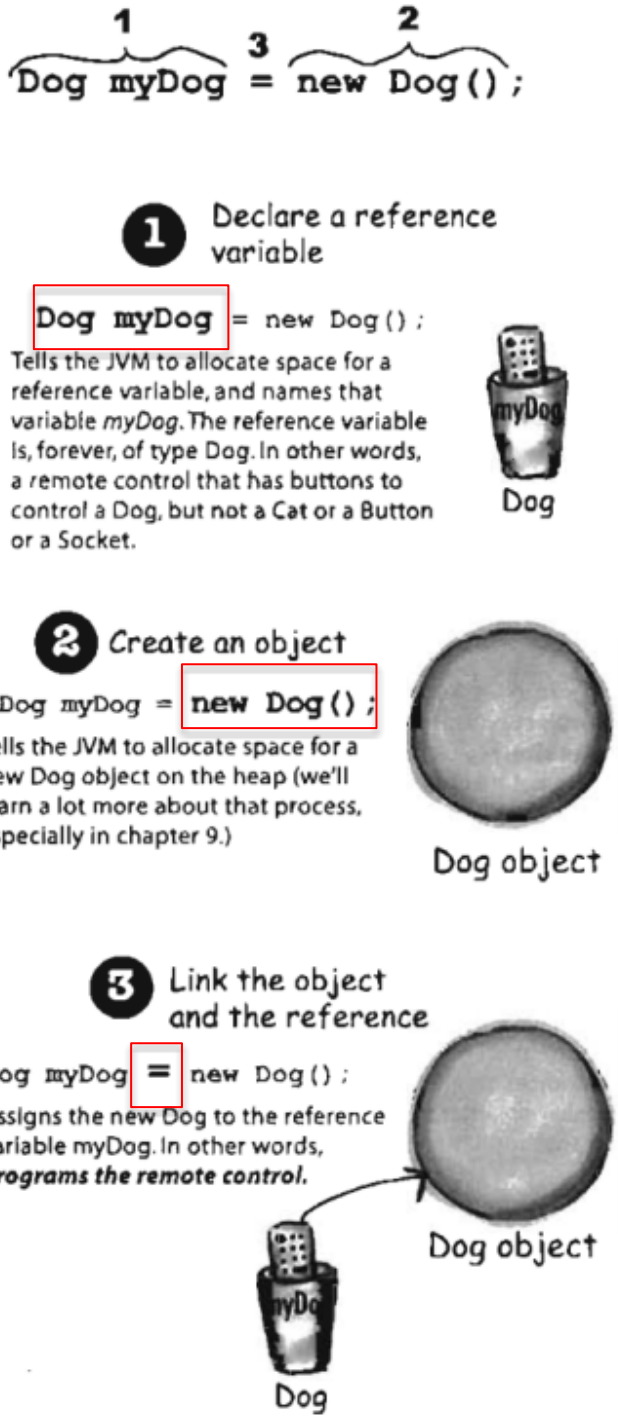
- Each **`Java`** class definition usually in a file by itself
  - File begins with name of the class
  - Ends with **`.java`**
- Class can be compiled separately
- Helpful to keep all class files used by a program in the same directory
- What happens when you compile a .java file?
  - .java file gets compiled into a .class file
    - Contains Java bytecode (instructions)
- You can send the .class file to people who use it, without revealing your actual code

Creating an Object

- Syntax rule

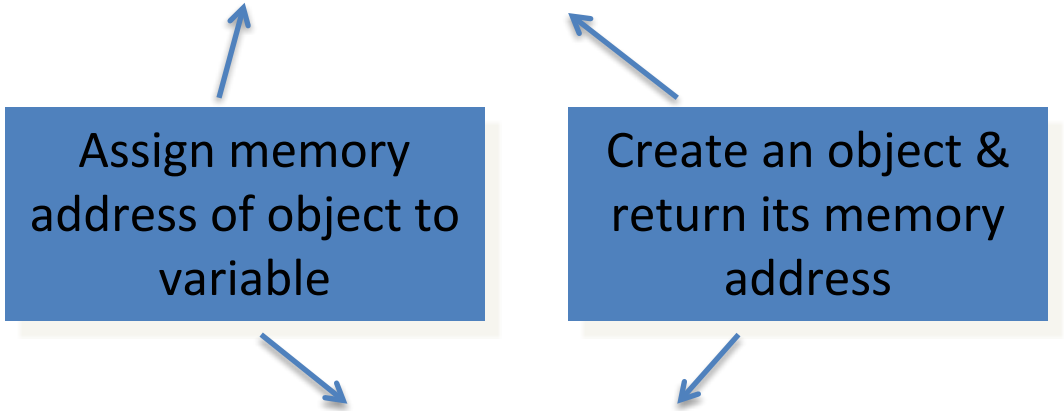
```
ClassName ObjectName= new
ClassName();
```

- What does the statement do?
  - The computer will create a new object, and assign its memory address to **ObjectName**
  - **ObjectName** is sometimes called an class type variable
    - It is a variable of class type **ClassName**
- Why do we need new?
  - So we know ClassName() is not executing a method but creating an object



Creating an Object

- Creating an object *ppopy* of class *Dog*
  - `Dog ppopy = new Dog();`



  - Scanner keyboard = `new Scanner(System.in);`
- Creating an object *keyboard* of class *Scanner*

Instance Variable vs. Local Variable

- *Instance* variables
  - Declared in a class (outside of methods)
  - Confined(국한됨) to the class
    - Can be used anywhere in the class that declares the variable, including inside the class' methods
  - (Java does not have global variables)
- *Local* variables
  - Declared in a method
  - Confined(국한됨) to the method
    - Can only be used inside the method that declares the variable

Instance Variables

- Data defined in the class are called *instance variables*
  - (defined outside of methods)
  - Java does not have global variables!

```
public String name;
public String breed;
public int    age;
```

Variable names

type: int, double, String …

`public`: no restrictions on how these instance variables are used (more details later!)

Using Instance Variables inside a Class

```
public class Dog
{
    public String name;
    public String breed;
    public int age;

    public int getAgeInHumanYears()
    {
        int humanAge = 0;
        if (age <= 2)
        {
            humanAge = age * 11;
        }
        else
        {
            humanAge = 22 + ((age-2) * 5);
        }
        return humanAge;
    }
}
```

- Any instance variables can be freely used in the class definition.
- Any method can freely access all the instance variables in the object as if they were declared in the method

Local Variables

- Variables declared inside a method are called *local* variables
  - May be used only inside the method
  - All variables declared in method `main` are local to `main`

- Local variables having the same name and declared in different methods are different variables

Local Variables

- listing 5.5A & 5.5B

```
With interest added, the new amount is $105.0
I wish my new amount were $800.0
```

```
public class BankAccount {
    public double amount;
    public double rate;

    public void showNewBalance() {
        double newAmount = amount + (rate / 100.0) * amount;
        System.out.println("With interest added, the new amount is $" + newAmount);
    }

    public static void main(String[] args)
    {
        BankAccount myAccount = new BankAccount();
        myAccount.amount = 100.00;
        myAccount.rate = 5;

        double newAmount = 800.00;
        myAccount.showNewBalance();
        System.out.println("I wish my new amount were $" + newAmount);
    }
}
```

Two different variables named *newAmount*

Methods

```
public String getMajor()
{
    return major;
}
```

```
public void increaseYear()
{
    classYear++;
}
```

What is the difference between these??

Methods

- Two kinds of Java methods
  - Methods that return a value
    - Examples: String's **substring()** method, String's **indexOf()** method
  - Methods that return nothing – a **`void`** method
    - Perform some other action
    - Example: **System.out.println()**
- "return" means "produce"
  - A method can produce a value so that other parts of the program can use it, or simply perform some actions
- When you use a method you "invoke" or "call" it

Methods

```
public String getMajor()
{
    return major;
}
```

Returns a String

Return type

```
public void increaseYear()
{
    classYear++;
}
```

Returns nothing

Defining `void` Methods

- Most method definitions we will see as **`public`**
- Method does not return a value
  - Specified as a **`void`** method
- Heading includes parameters
- Body enclosed in braces **`{ }`**
- Think of method as defining an action to be taken

```
public void writeOutput()
{
    System.out.println("Name: " + name);
    System.out.println("Breed: " + breed);
    System.out.println("Age in calendar years: " +
                       age);
    System.out.println("Age in human years: " +
                       getAgeInHumanYears());
    System.out.println();
}
```

Methods That Return a Value

- Consider method **`getAgeInHumanYears( )`**

```
public int getAgeInHumanYears()
{
    int humanAge = 0;
    if (age <= 2)
    {
        humanAge = age * 11;
    }
    else
    {
        humanAge = 22 + ((age-2) * 5);
    }
        return humanAge;
}
```

- Heading declares type of value to be returned
- Last statement executed is **`return`**

`return` Statement

- A method that returns a value must have *at least one* `return` statement
- Terminates the method, and returns a value
- Syntax:
  - `return Expression;`

```
public String getClassYear()
{
    if (classYear == 1)
        return "Freshman";
    else if (classYear == 2)
        return "Sophomore";
    else if ...
}
```

Expression can be any expression that produces a value of type specified by **the return type** in the method heading

Lab: Dog class test

- Implement a class *Dog* and write a test program
  - Note class has
    - Three pieces of data (**instance variables**)
      - String name, String breed, int age
    - Two behaviors (methods)
      - writeOutput(), getAgeInHumanYea
        ()
  - Each instance of this type has its own copies of the data items
  - Use of **`public`**
    - No restrictions on how variables used
    - Later will replace with **`private`**

LISTING 5.1 Definition of a Dog Class

```
public class Dog
{
    public String name;
    public String breed;
    public int age;
    public void writeOutput()
    {
        ...
    }
    public int getAgeInHumanYears()
    {
        ...
    }
}
```

Sample screen output

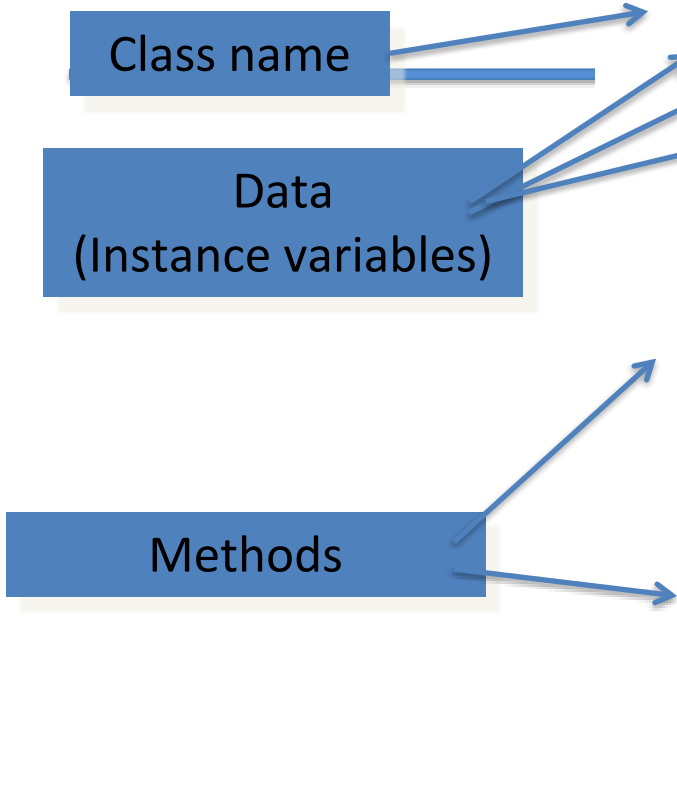
```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

Lab: Dog class test

- Implement a class *Dog* and write a test program
  - writeOutput(): print all instance variable data in dog class
  - getAgeInHumanYears(): convert dog age to human years

- Write a test class DogDemo
  - Add separate java file in project
  - Make instances and set the values
  - Print matched human age and output

```
int humanAge = 0;
if (age <= 2)
{
    humanAge = age * 11;
}
else
{
    humanAge = 22 + ((age-2) * 5);
}
```



LISTING 5.1 Definition of a Dog Class

```
public class Dog
{
    public String name;
    public String breed;
    public int age;
    public void writeOutput()
    {
        System.out.println("Name: " + name);
        System.out.println("Breed: " + breed);
        System.out.println("Age in calendar years: " +
                           age);
        System.out.println("Age in human years: " +
                           getAgeInHumanYears());
        System.out.println();
    }
    public int getAgeInHumanYears()
    {
        int humanAge = 0;
        if (age <= 2)
        {
            humanAge = age * 11;
        }
        else
        {
            humanAge = 22 + ((age-2) * 5);
        }
            return humanAge;
    }
}
```

Later in this chapter we will see that the modifier `public` *for instance variables should be replaced with* `private`.

LISTING 5.2 Using the Dog Class and Its Methods

```
public class DogDemo
{
    public static void main(String[] args)
    {
        Dog balto = new Dog();
        balto.name = "Balto";
        balto.age = 8;
        balto.breed = "Siberian Husky";
        balto.writeOutput();

        Dog scooby = new Dog();
        scooby.name = "Scooby";
        scooby.age = 42;
        scooby.breed = "Great Dane";
        System.out.println(scooby.name + " is a " +
                           scooby.breed + ".");
        System.out.print("He is " + scooby.age +
                         " years old, or ");
        int humanYears = scooby.getAgeInHumanYears();
        System.out.println(humanYears + " in human years.");
    }
}
```

Sample Screen Output

```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

Lab: Species class

- Implement a class *Species* and write a test program (List 5.3 & 5.4)
  - SpeciesFirstTry Class
    - Data: Name, population, and growth rate
  - Action
    - readInput(): read data
    - writeOutput(): print result
    - getPopulationIn10: compute the population in 10 years

Sample Screen Output

```
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
```

Lab: Species class

LISTING 5.3 **A Species Class Definition—First Attempt** *(part 1 of 2)*

We will give a better version of this class later in this chapter.

```
import java.util.Scanner;
public class SpeciesFirstTry
{
    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();
```

Later in this chapter you will see that the modifier `public` *for instance variables should be replaced with* `private`.

```
        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 + "%");
    }
    public int getPopulationIn10()
    {
        int result = 0;
        double populationAmount = population;
        int count = 10;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = populationAmount +
                               (growthRate / 100) *
                               populationAmount;
            count--;
        }
        if (populationAmount > 0)
                result = (int)populationAmount;
        return result;
    }
}
```

LISTING 5.4 Using the Species Class and Its Methods *(part 1 of 2)*

```
public class SpeciesFirstTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
        System.out.println("Enter data on the Species of "+
                           "the Month:");

        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();
        int futurePopulation =
            speciesOfTheMonth.getPopulationIn10();
        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.getPopulationIn10());
    }
}
```

Keyword this

- Within a method definition, you can use the keyword **this** as a name for the object receiving the method call. (객체 자기자신)
- Example

```
this.name = keyboard.nextLine();
```

```
import java.util.Scanner;
public class SpeciesFirstTry
{
    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();
```

class later in this chapter.

Later in this chapter you will see that the modifier **public** *for instance variables should be replaced with* **private**.

Keyword this

```java
public Fruit(String name, String color, double weight, int count) {
    this.name = name;
    this.color = color;
    this.weight = weight;
    this.count = count;
}
```

```java
public class Fruit {
    public String name;
    public String color;
    public double weight;
    public int count;

    public Fruit(String name, String color, double weight, int count) {
        name = name;
        color = color;
        weight = weight;
        count = count;
    }

    public static void main(String[] args) {
        Fruit banana = new Fruit("banana", "yellow", 5.0, 10);
        System.out.println("name : " + banana.name);        // name : null
        System.out.println("color : " + banana.color);      // color : null
        System.out.println("weight : " + banana.weight);    // weight : 0.0
        System.out.println("count : " + banana.count);      // count : 0
    }
}
```

Blocks { }

- Recall compound statements
  - Enclosed in braces { }
- When you declare a variable within a compound statement
  - The compound statement is called a *block*
  - The scope of the variable is from its declaration to the end of the block
- Variable declared outside the block usable both outside and inside the block

Methods with Parameters

- We can make it more versatile by giving the method a parameter to specify how many years
  - Note sample program, listing 5.6

```
public int getPopulationIn10()
{
    int result = 0;
    double populationAmount = population;
    int count = 10;
    while ((count > 0) && (populationAmount > 0)) {
        populationAmount = populationAmount +
            (growthRate / 100) * populationAmount;
        count--;
    }
    if (populationAmount > 0)
        result = (int)populationAmount;
    return result;
}
```

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

Parameters of Primitive Type

- Note the declaration
  ```
  public int predictPopulation(int years)
  ```
  - The *formal* parameter is `years`

- Calling the method
  ```
  int futurePopulation =
   speciesOfTheMonth.predictPopulation(10);
  ```
  - The *actual* parameter (=argument) is the integer 10
- View sample program, listing 5.7

  `class SpeciesSecondClassDemo`

LISTING 5.7 Using a Method That Has a Parameter

```
/**
Demonstrates the use of a parameter
with the method predictPopulation.
*/
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));
    }
}
```

Sample Screen Output

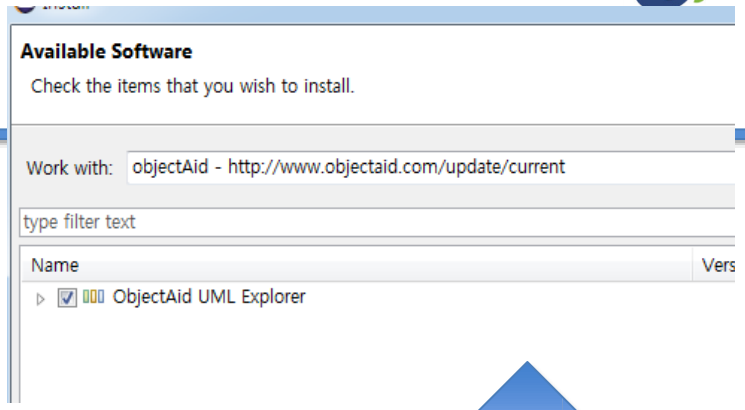
The output is exactly the same as in Listing 5.4.

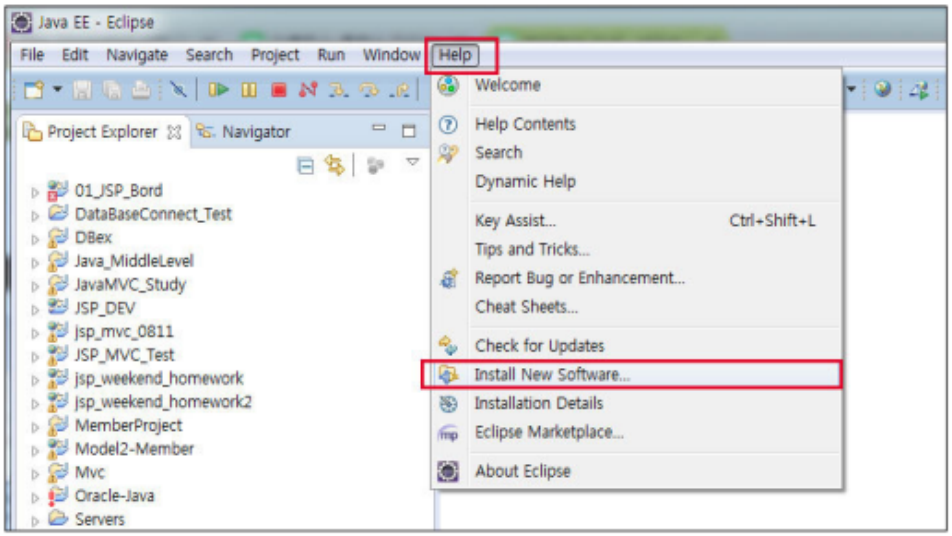
Parameter vs. Argument

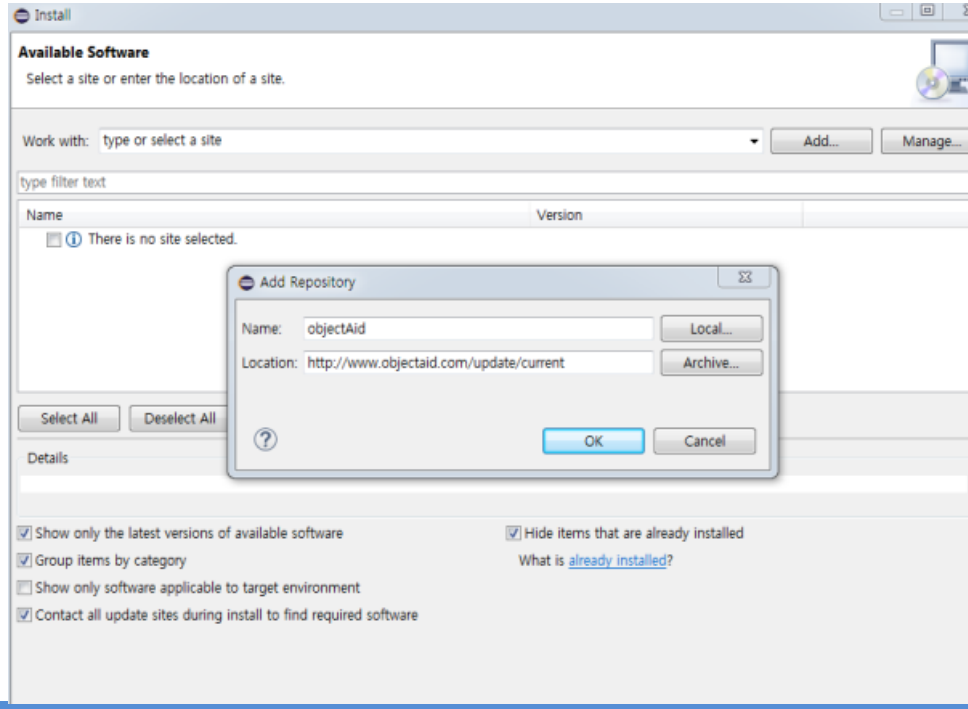
- These two terms *parameter* and *argument* are sometimes loosely used interchangeably
- Difference ?
  - ***parameter*** (sometimes called *formal parameter*) is often used to refer to the variable as found in the function definition
  - ***argument*** (sometimes called *actual parameter*) refers to the actual input passed.

UML diagram

- Add plug-in for UML
  - ObjectAid plug-in
  - http://www.objectaid.com/update/current

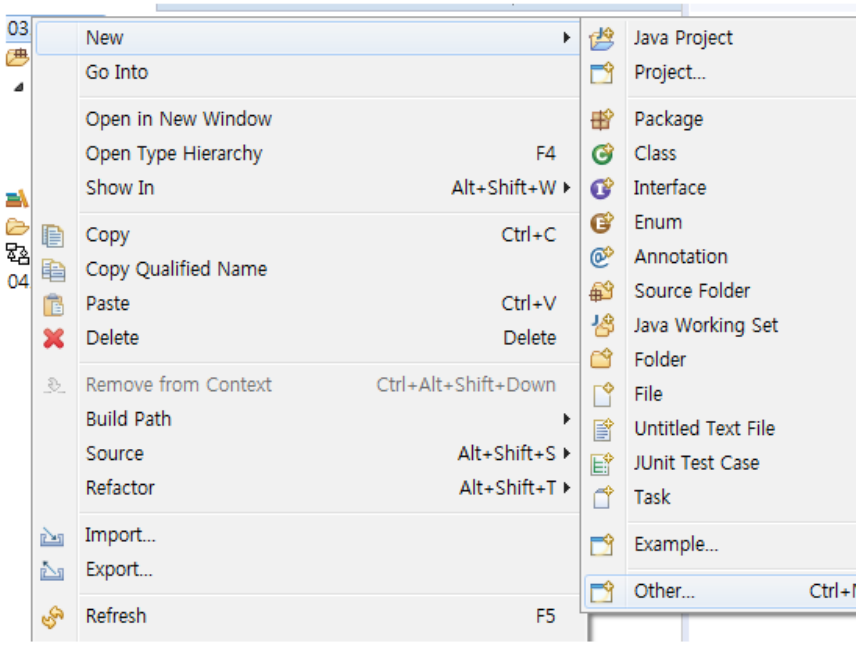


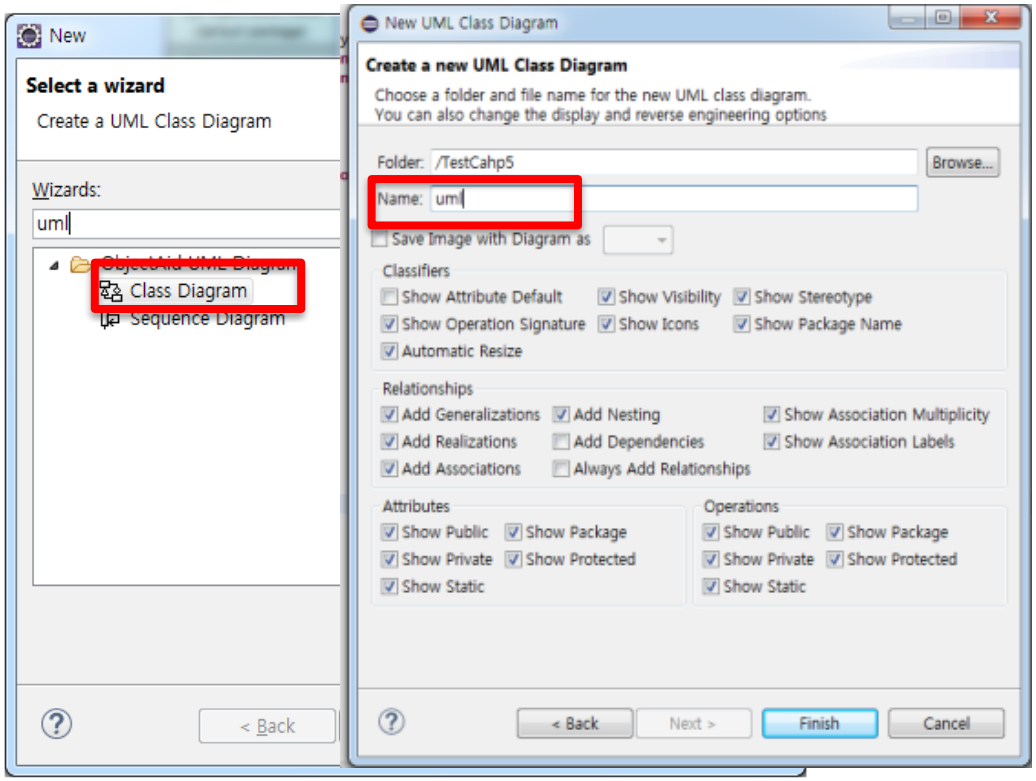




Install UML diagram

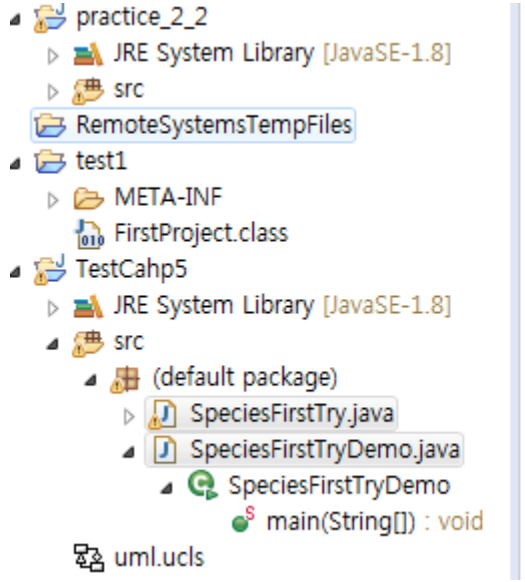
- Project right click-> new-> other->type “UML”-> select “Class Diagram” ->define name anything

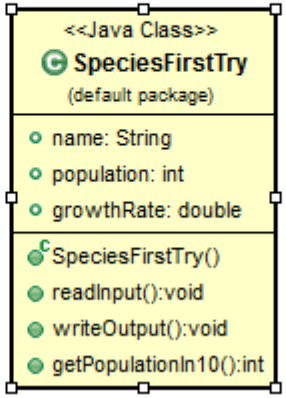


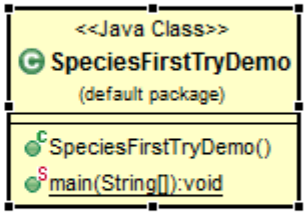


Install UML diagram

- Drag java files to uml.ucls window







5.2 Information Hiding, Encapsulation

Information Hiding

- Programmer using a class method need **not** know details of implementation
  - Only needs to know *what* the method does
- Information hiding:
  - Designing a method so it can be used without knowing details
- Also referred to as *abstraction*
- Method design should separate *what* from *how*

public/private Modifier

```
public class Student
{
    public int classYear;
    public void setMajor();
    private String major;
    private void setYear();
}
```

- **public**
  - there is no restriction on how you can use the method or instance variable
- **private**
  - can not directly use the method or instance variable's name outside the class

public/private Modifier

```
public class Student
{
    public  int classYear;
    private String major;
}

public class StudentTest{
   public static void main(String[] args){
      Student jack = new Student();
      jack.classYear = 1;
      jack.major = “Computer Science”;
   }
}
```

OK, classYear is public

Error!!! *major* is *private*

`public` and `private` Modifiers

- Type specified as `public`
  - Any other class can directly access that object by name
- Classes generally specified as `public`
- Instance variables usually not `public`
  - Instead specify as `private`

More about `private`

- **Hides instance variables and methods inside** the class/object. The private variables and methods are still there, holding data for the object.

- **Invisible to external users of the class**
  - Users cannot access private class members directly

- **Information hiding !**

Programming Example

```java
// Rectangle.java
public class Rectangle {
    private int width, height;
    private int area;

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

```java
// RectangleTest.java
public class RectangleTest {
    public static void main(String[] args) {
        Rectangle box = new Rectangle();
        box.setDimensions(10, 5);
        System.out.println("Area is: " + box.getArea());
        box.width = 6;
        System.out.println("Area is: " + box.getArea());
    }
}
```

Output 50!

Error!!

Statement such as

`box.width = 6;`

is illegal since width is `private` Keeps remaining elements of the class consistent in this example

Programming Example

- Another implementation of a Rectangle class
- Note **`setDimensions`** method
  - Only way the **`width`** and **`height`** may be altered outside the class

```
// Rectangle.java
public class Rectangle {
    private int width, height;

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

```
// RectangleTest.java
public class RectangleTest {
    public static void main(String[] args) {
        Rectangle box = new Rectangle();
        box.setDimensions(10, 5);
        System.out.println("Area is: " + box.getArea());
        box.setDimensions(6, 5);
        System.out.println("Area is: " + box.getArea());
    }
}
```

Now prints a correct area

Accessor and Mutator Methods

- How do you access **private** instance variables?

- Accessor methods (a.k.a. get methods, **getters**)
  - Typically named ***getSomeValue***
  - Allow you to look at data in private instance variables
- Mutator methods (a.k.a. set methods, **setters**)
  - Allow you to change data in private instance variables
  - Typically named ***setSomeValue***

Accessor and Mutator Methods

```
public class Student
{
     private String name;
     private int age;

     public void setName(String studentName) {
          name = studentName;
     }
     public void setAge(int studentAge) {
          age = studentAge;
     }

     public String getName() {
          return name;
     }
     public int getAge() {
          return age;
     }
}
```

Mutators

Accessors

Accessor and Mutator Methods

```
public class Student
{
     private String name;
     private int age;

     public void setName(String studentName) {
          name = studentName;
     }
     public void setAge(int studentAge) {
          if( studentAge > 0 )
               age = studentAge;
          else
               System.out.println("The input for
age shuld be positive");
     }

     public String getName() {
          return name;
     }
     public int getAge() {
          return age;
     }
}
```

Mutators

Accessors

Accessor and Mutator Methods

- Consider an example class with accessor and mutator methods
- Lab: View sample code, listing 5.11
  `class SpeciesFourthTry`
- Note the mutator method
  - `setSpecies`
- Note accessor methods
  - `getName`, `getPopulation`, `getGrowthRate`

LISTING 5.11 A Class with Accessor and Mutator Methods

```
import java.util.Scanner;
public class SpeciesFourthTry
{
    private String name;
    private int population;
    private double growthRate;

    <The definitions of the methods readInput, writeOutput, and
     predictPopulation go here. They are the same as in Listing
     5.3 and Listing 5.6.>
    public void setSpecies(String newName, int newPopulation,
                           double newGrowthRate)
    {
        name = newName;
        if (newPopulation >= 0)
            population = newPopulation;
        else
        {
                System.out.println(
                        "ERROR: using a negative population.");
                System.exit(0);
        }
        growthRate = newGrowthRate;
    }
    public String getName()
    {
        return name;
    }
    public int getPopulation()
    {
        return population;
    }
    public double getGrowthRate()
    {
        return growthRate;
    }
}
```

Yes, we will define an even better version of this class later.

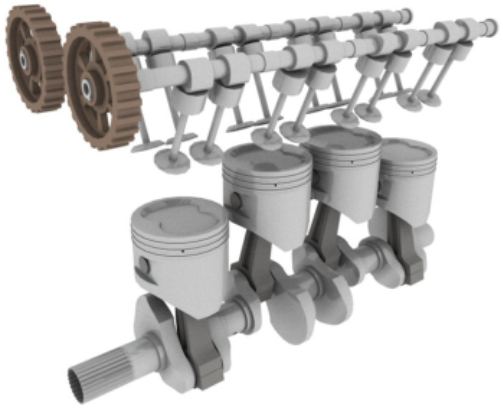
A mutator method can check to make sure that instance variables are set to proper values.

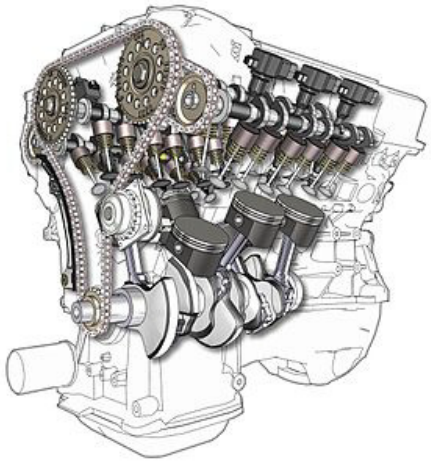
Why make methods `private`?

- **Why make methods private?**

- Helper methods that will only be used from inside a class should be private
  - External users have no need to call these methods

- **Encapsulation**





Encapsulation

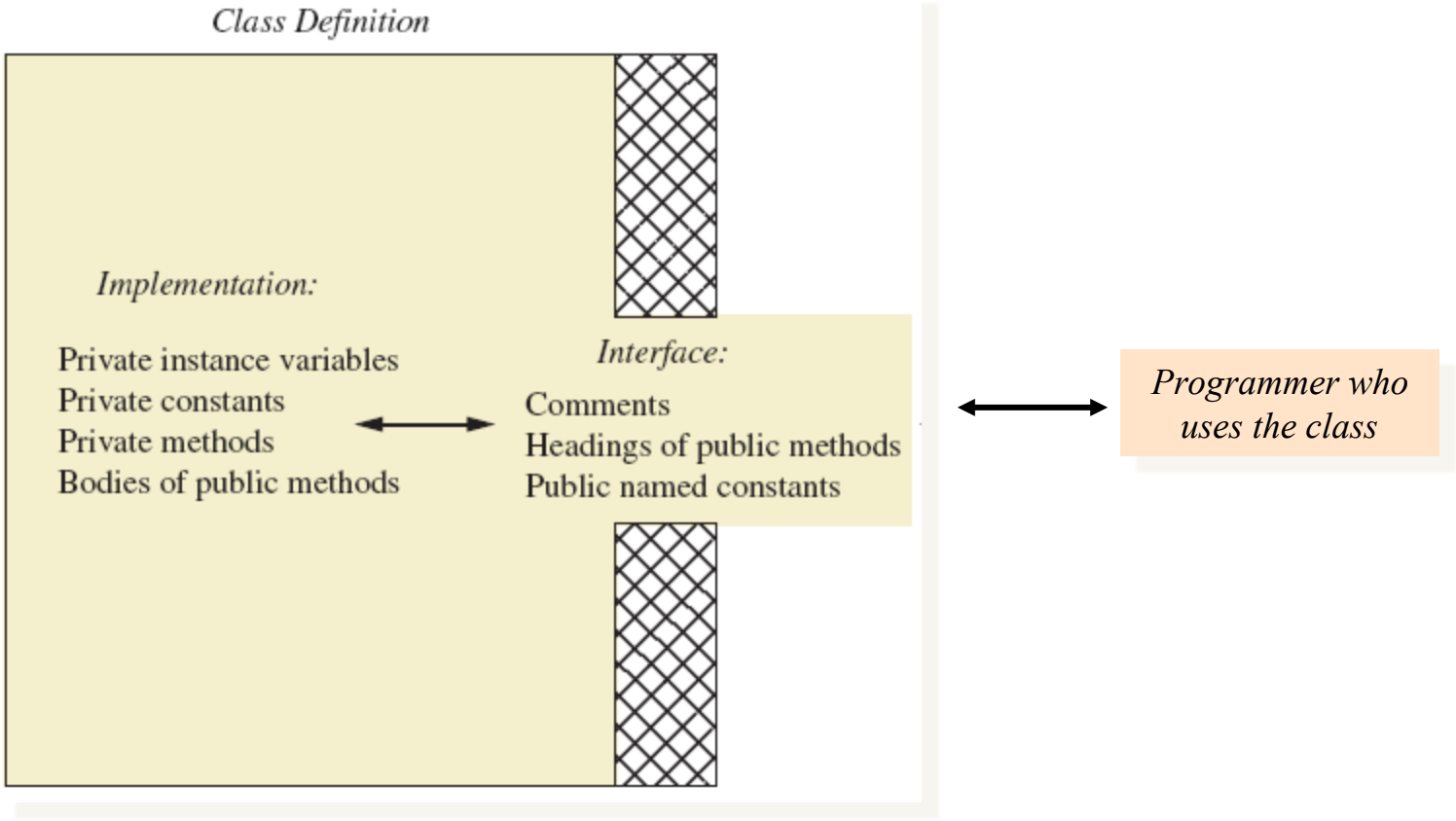
- Consider example of **driving a car**
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs
- Encapsulation divides class definition into
  - Class interface
  - Class implementation

Encapsulation

- **A *class interface***
  - Tells **what the class does**
  - Gives headings for public methods and comments about them
- **A *class implementation***
  - Contains private variables
  - Includes definitions of public and private methods

Encapsulation

- Figure 5.3 A well encapsulated class definition



Example : private Method

- Make **helping methods** ***private***

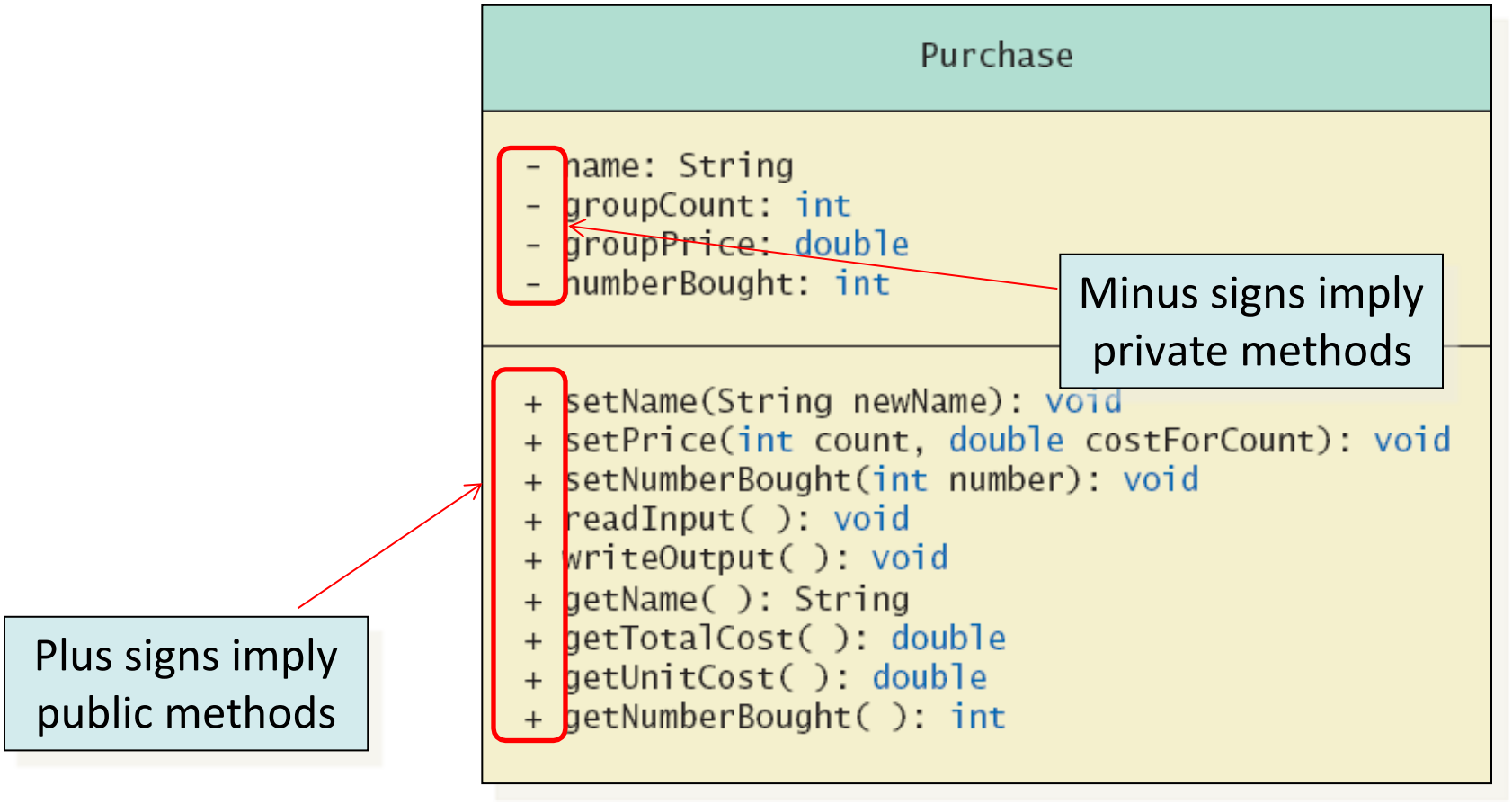
```
public class RightTriangle {
     private double side_a;
     private double side_b;

     private double square(double d) {
          // some calculation
     } // don't want others to use - rounded for rounded output

     private double sqrt(double d) {
          // some complicated calculation
     } // don't want others to use - optimized for triangle only

     public double getSideC() {
          return this.sqrt(this.square(side_a) + this.square(side_b));
     }
}
```

Guidelines When You Define a Class

- Comments before class definition (this is your header) and before method
- **Instance variables are *private***
- Provide ***public*** **accessor** and **mutator methods**
- Make **helping methods *private***
- **/* */ for user-interface comments and // for implementation comments**

UML Class Diagrams

- Recall a UML class diagram



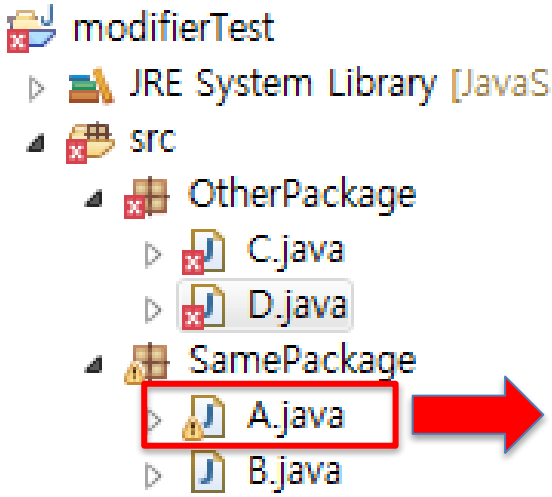
Access modifier

- private
- default
- protected
- public

Easy to access!

	Current Class Only	Same package		Different package		All Classes
		Subclass (상속)	Not subclass	Subclass (상속)	Not subclass	
private	O	X	X	X	X	X
(default)	O	O	O	X	X	X
protected	O	O	O	O	X	X
public	O	O	O	O	O	O

Lab: Access modifier

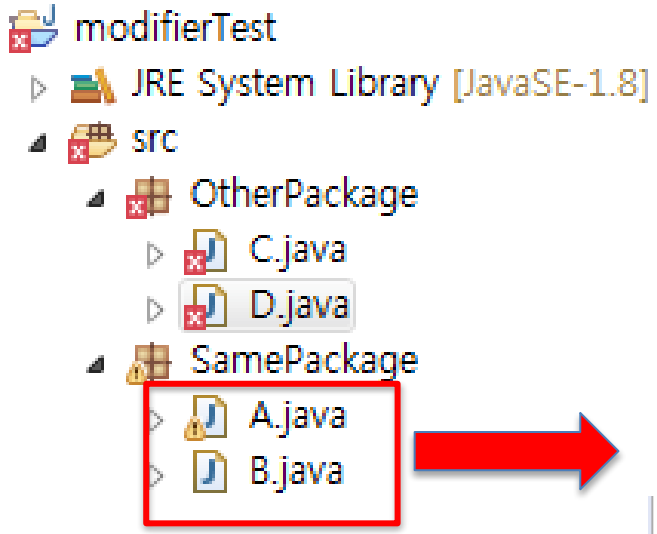


	Current Class Only
private	O
(default)	O
protected	O
public	O

A.java

```
1  package SamePackage;
2
3  public class A {
4
5      public void _pub() {
6          System.out.println("public");
7      }
8      private void _pri() {
9          System.out.println("private");
10     }
11     protected void _prot() {
12         System.out.println("protected");
13     }
14         void _def() {
15         System.out.println("default");
16     }
17
18     void a() {
19         _pub();
20     }
21     void b() {
22         _pri();
23     }
24     void c() {
25         _prot();
26     }
27     void d() {
28         _def();
29     }
30 }
31
```

Lab: Access modifier



	Same package	
	Subclass (상속)	Not subclass
private	X	X
(default)	O	O
protected	O	O
public	O	O

A.java

```
package SamePackage;

public class A {

    public void _pub() {
        System.out.println("public");
    }
    private void _pri() {
        System.out.println("private");
    }
    protected void _prot() {
        System.out.println("protected");
    }
        void _def() {
        System.out.println("default");
    }

}
```

A.java

```
package SamePackage;

public class A {

    public void _pub() {
        System.out.println("public");
    }
    private void _pri() {
        System.out.println("private");
    }
    protected void _prot() {
        System.out.println("protected");
    }
        void _def() {
        System.out.println("default");
    }

}
```

B.java

```
package SamePackage;

public class B extends A {

    void a() {
        _pub();
    }
    void b() {
        _pri();
    }
    void c() {
        _prot();
    }
    void d() {
        _def();
    }
}
```

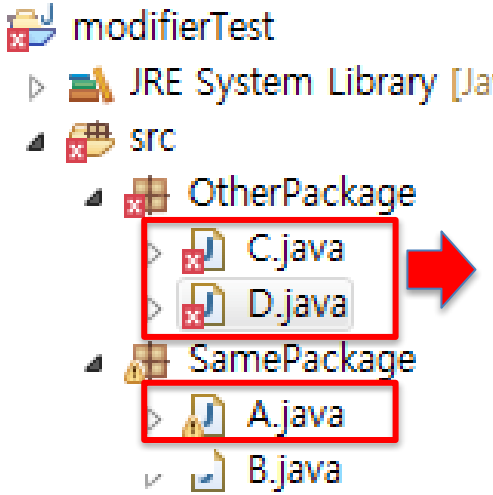
Private type access error!

B.java C.java

```
package SamePackage;

public class B {

    A test = new A();
    void a() {
        test._pub();
    }
    void b() {
        test._pri();
    }
    void c() {
        test._prot();
    }
    void d() {
        test._def();
    }
}
```

Private type access error!

Lab: Access modifier



	Different package	
	Subclass (상속)	Not subclass
private	X	X
(default)	X	X
protected	O	X
public	O	O

A.java

```
1  package SamePackage;
2
3  public class A {
4
5      public void _pub() {
6          System.out.println("public");
7      }
8      private void _pri() {
9          System.out.println("private");
10     }
11     protected void _prot() {
12         System.out.println("protected");
13     }
14     void _def() {
15         System.out.println("default");
16     }
17
18 }
19
```

B.java C.java

```
1  package OtherPackage;
2
3  import SamePackage.A;
4
5  public class C extends A {
6
7      void a() {
8          _pub();
9      }
10     void b() {
11         _pri();
12     }
13     void c() {
14         _prot();
15     }
16     void d() {
17         _def();
18     }
19 }
20
```

Private & default type access error!

A.java

```
1  package SamePackage;
2
3  public class A {
4
5      public void _pub() {
6          System.out.println("public");
7      }
8      private void _pri() {
9          System.out.println("private");
10     }
11     protected void _prot() {
12         System.out.println("protected");
13     }
14     void _def() {
15         System.out.println("default");
16     }
17
18 }
19
```

B.java C.java D.java

```
1  package OtherPackage;
2  import SamePackage.A;
3
4  public class D {
5      A test = new A();
6      void a() {
7          test._pub();
8      }
9      void b() {
10         test._pri();
11     }
12     void c() {
13         test._prot();
14     }
15     void d() {
16         test._def();
17     }
18 }
```

Private & default & protected type access error!

Practice 5

- Ex5_1. Implement a class *MotorBoat* and write a test program
  - Attributes (public):
    - Capacity of fuel tank (*C*), Amount of fuel in the tank (*f*)
    - Maximum speed (*M*), Current speed (*s*)
    - Efficiency of the boat's motor (*e*)
  - Methods:
    - Given a parameter (time *t*), print the amount of fuel used at the maximum and current speeds ($f=e*s^2*t$)
    - Given a time *t*, print the travel distance ($s*t$)
    - For current speed *s* and fuel amount *f*, print the travel distance

Practice 5

- Ex5_2. Modify Ex5_1
  - Make tank capacity (C) and maximum speed (M) constants
    - public static final double tankCapacity = 60.0;
  - Make all other instance variables private, and implement their getter/setter methods
    - Check if fuel amount (f) and current speed (s) exceeds C and M; if so, print an error message
  - Modify the test program accordingly