



LECTURE 05

Designing Classes

ITCS209 Object Oriented Programming

Dr. Siripen Pongpaichet

(Some materials in the lecture are done by

Aj. Suppawong Tuarob And Aj. Thanapon Noraset)



Recap – Lecture 04

- **Array -> Fixed size**
 - Array of primitive type and objects
 - Construct & initialize arrays
 - Access & modify elements in array
 - Passing array to and returning array from methods
 - Array algorithms and 2D array
- **ArrayList -> Dynamic size and store objects only**
 - Construct & initialize ArrayList
 - Retrieve, insert, delete, and modify objects in ArrayList
 - Wrappers and auto boxing

In the past week, how many **hours** you spent on coding and reviewing lecture outside the classroom?



Array OR ArrayList

- List of courses registered by a student
- List of points to draw a polygon
- List of items in a shopping cart
- List of months' name
- A bingo game's board (5 x 5)
- To find minimum value from 10 random numbers

Let's Review ArrayList

Syntax To construct an array list: `new ArrayList<typeName>()`

To access an element: `arraylistReference.get(index)`
`arraylistReference.set(index, value)`

Variable type **Variable name** **An array list object of size 0**

```
ArrayList<String> friends = new ArrayList<String>();
```

Use the
get and set methods
to access an element.

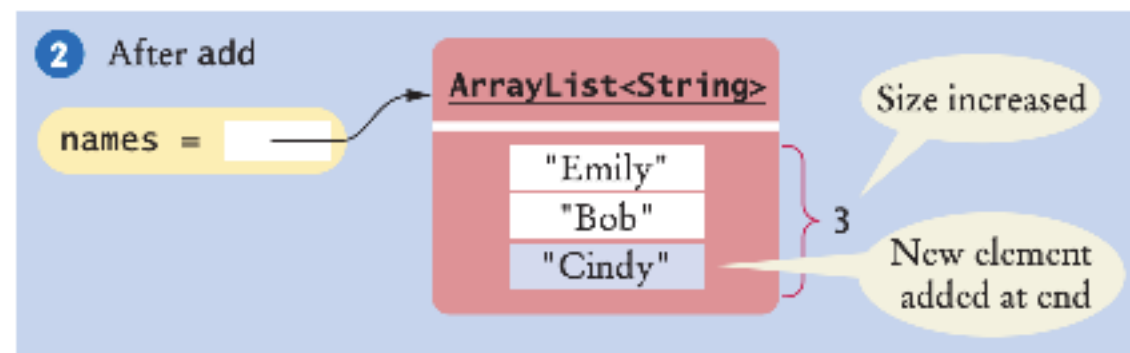
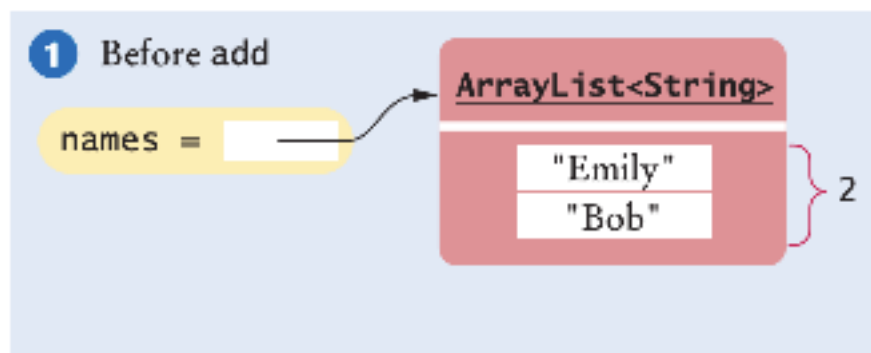
```
friends.add("Cindy");  
String name = friends.get(i);  
friends.set(i, "Harry");
```

The add method
appends an element to the array list,
increasing its size.

The index must be ≥ 0 and $< \text{friends.size}()$.

ArrayList – Add new element at the end

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Emily");    // Now names has size 1 and element "Emily"  
names.add("Bob");      // Now names has size 2 and elements "Emily", "Bob"  
names.add("Cindy");    // names has size 3 and  
                        // elements "Emily", "Bob", and "Cindy"
```



```
System.out.println(names); // Prints [Emily, Bob, Cindy]
```

ArrayList – Get & Set Element

```
String name = names.get(2);
```

```
System.out.println(name); // Cindy
```

```
names.set(2, "Carolyn");
```

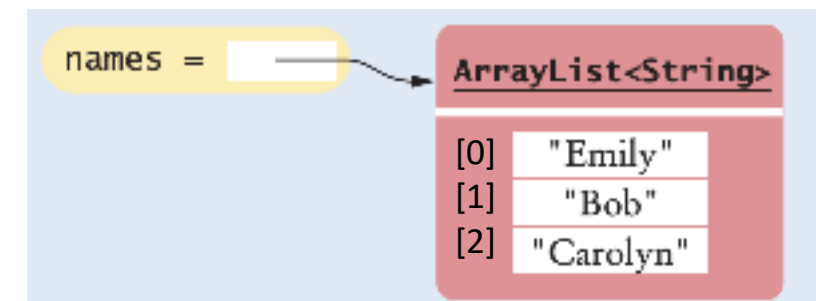
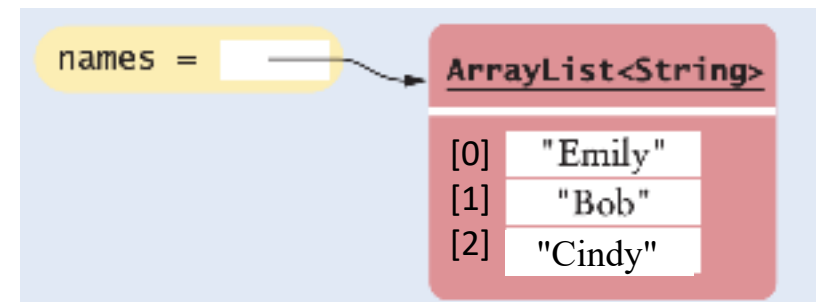
```
name = names.get(2);
```

```
System.out.println(name); // Carolyn
```

```
int n = names.size(); // 3
```

```
name = names.get(n); // error
```

last valid index is `names.size() - 1`



// Trick! To get the last element

```
String name = names.get(names.size() - 1);
```

ArrayList – Add and Remove at Index

```
names.add(1, "Ann");
```

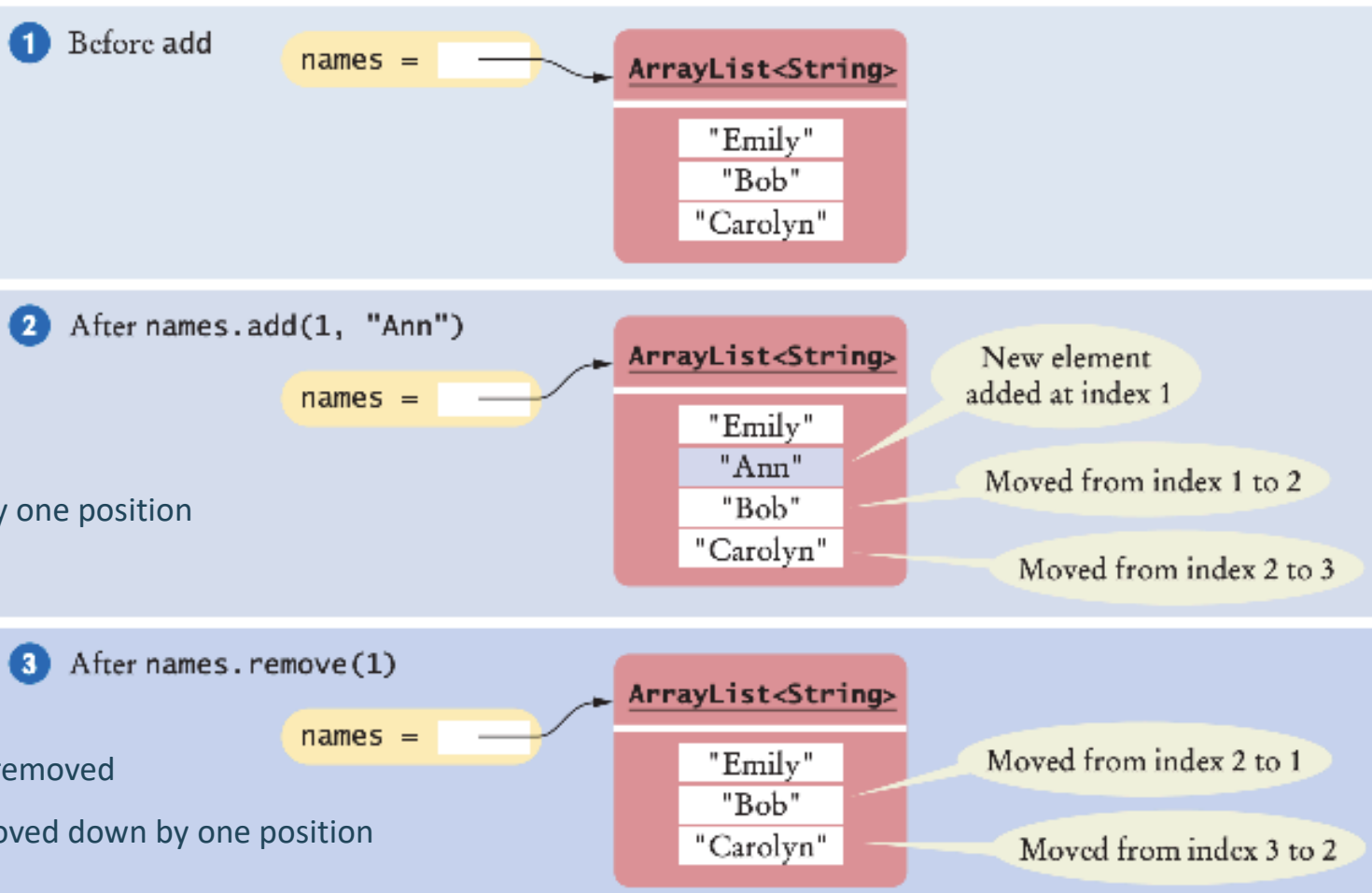
```
// now names has size 4, and "Ann" is at index 1,
```

```
// all elements at index 1 and larger are moved by one position
```

```
names.remove(1);
```

```
// now names has size 3, and "Ann" at index 1 is removed
```

```
// all elements after the removed element are moved down by one position
```

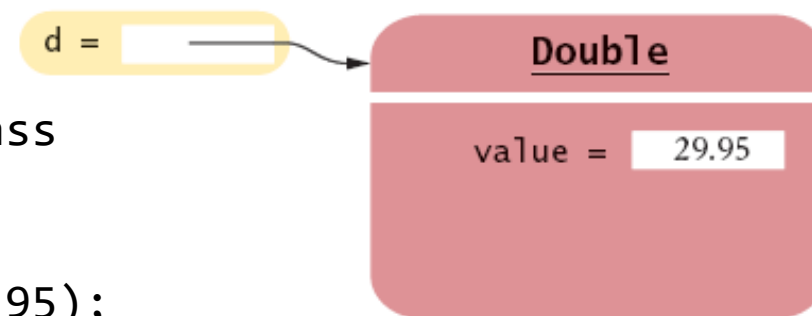


ArrayList – Primitive Type **xxx CANNOT xxx**

- ArrayList can contain only Objects not primitive type
- To treat primitive type values as objects, you must use **wrapper classes**:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

// How to use wrapper class
Double d = 29.95;
// OR
Double d = new Double(29.95);



```
// ArrayList
ArrayList<double> data = new ArrayList<double>(); // ERROR!!

ArrayList<Double> data = new ArrayList<Double>(); // OK
data.add(29.95);
double d = data.get(0);
```

Auto-boxing

- Auto-boxing: Starting with Java 5.0, **conversion between primitive types and the corresponding wrapper classes** is **automatic**.

```
Double wrapper = 29.95;  
// auto-boxing; same as Double wrapper = new Double(29.95);
```

```
double x = wrapper;  
// auto-unboxing; same as double x = wrapper.doubleValue();
```

```
// ArrayList
```

```
ArrayList<Double> values = new ArrayList<Double>();
```

```
values.add(42.0);
```

```
double theAnswer = values.get(0);
```

- Auto-boxing even works inside arithmetic expressions

```
Double e = wrapper + 1;
```

- Means:

- *auto-unbox d into a double*
- *add 1*
- *auto-box the result into a new Double*
- *store a reference to the newly created wrapper object in e*

Array vs ArrayList – How to choose?

- Here are some recommendations.
 - If the size of a collection never changes, use an array.
 - If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
 - Otherwise, use an array list.

Length vs Size – Java Syntax

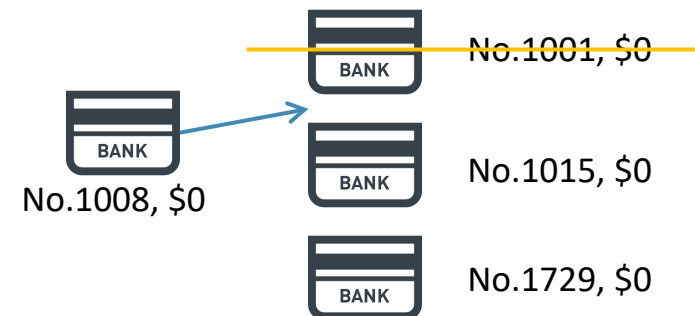
Data Type	Number of Elements
Array	a.length
Array list	a.size()
String	a.length()

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
    if (values[i] > largest)
    {
        largest = values[i];
    }
}
```

```
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
    if (values.get(i) > largest)
    {
        largest = values.get(i);
    }
}
```

ArrayList of Your Own Objects

```
public class BankAccount
{
    public BankAccount(int anAccountNumber)
    {
        accountNumber = anAccountNumber;
        balance = 0;
    }
    public int getAccountNumber() { return accountNumber; }
}
```



```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>(); // initialize arraylist (empty arraylist)
accounts.add(new BankAccount(1001)); // add new BankAccount object into arraylist
accounts.add(new BankAccount(1015)); // add another BankAccount object at the end of arraylist
accounts.add(new BankAccount(1729)); // add another BankAccount object at the end of arraylist
accounts.add(1, new BankAccount(1008)); // add anew BankAccount at index 1, and shift other accounts
accounts.remove(0); // remove a BankAccount at index 0

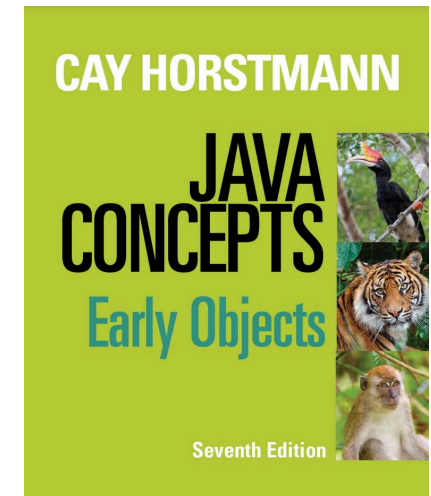
System.out.println("Size: " + accounts.size()); // 3
BankAccount first = accounts.get(0) // get the first BankAccount object
System.out.println("First account number: " + first.getAccountNumber()); // 1008
BankAccount last = accounts.get(accounts.size() - 1); // get the last BankAccount object
System.out.println("Last account number: " + last.getAccountNumber()); // 1729
```

Problem Solving with Arrays

Consider this example problem: You are given the quiz scores of a student. You are to compute the final quiz score, which is the sum of all scores after dropping the lowest one. For example, if the scores are

8 7 8.5 9.5 7 4 10

then the final score is 50.



Book reference : Java Concepts by Cay S. Horstmann – Chapter 7, Arrays and Array List (Section 7.4 & HOW TO 7.1)

For more examples please visit -> <http://wiley.com/go/javaexamples>

A final quiz score is computed by adding all the scores, except for the **lowest one**.

- **Step 1:** Decompose your task into steps
 - Read inputs
 - Remove the minimum
 - Calculate the sum
- **Step 2:** Determine which algorithms(s) you need
 - Read inputs
 - Find the minimum
 - Find it position
 - Remove the minimum
 - Calculate the sum

Can we do better?

A final quiz score is computed by adding all the scores, except for the **lowest one**. (cont.)

- **Step 1:** Decompose your task into steps
 - Read inputs
 - Remove the minimum
 - Calculate the sum
- **Step 2:** Determine which algorithms(s) you need
 - Read inputs
 - Find the minimum
 - Find its position
 - Remove the minimum
 - Calculate the sum

Can we do better?

*It is easy to **compute the sum and subtract the minimum**. So the new plan is*

- Read inputs
- Find the minimum
- Calculate the sum
- Subtract the minimum

A final quiz score is computed by adding all the scores, except for the **lowest one**. (cont.)

- **Step 3:** Use classes and method to structure the program
 - It may be possible to put all steps in your plan into the **main** method and run. However, it is better to carry out each step in a separate method. And it is a good idea to come up with a class that is responsible for collecting and processing the data.
 - Can you think of any classes for this problem?
 - A class for storing (contains *array* of quiz scores) and processing the data
 - A class for reading the user input scores and displaying the result (contains *main* method)
 - Etc.

A final quiz score is computed by adding all the scores, except for the **lowest one**. (cont.)

In our example, let's provide a class Student. A student has an array of scores.

```
public class Student
{
    private double[] scores;
    private double scoresSize;
    . . .
    public Student(int capacity) { . . . }
    public boolean addScore(double score) { . . . }
    public double finalScore() { . . . }
}
```

Student class contains array of quiz scores and other methods to process the quiz scores.

A second class, ScoreAnalyzer, is responsible for reading the user input and displaying the result. Its main method simply calls the Student methods:

```
Student fred = new Student(100);
System.out.println("Please enter values, Q to quit:");
while (in.hasNextDouble())
{
    if (!fred.addScore(in.nextDouble()))
    {
        System.out.println("Too many scores.");
        return;
    }
}
System.out.println("Final score: " + fred.finalScore());
```

ScoreAnalyzer class
contains a main method

A final quiz score is computed by adding all the scores, except for the **lowest one**. (cont.)

In our example, let's provide a class Student. A student has an array of scores.

```
public class Student
{
    private double[] scores;
    private double scoresSize;
    . . .
    public Student(int capacity) { . . . }
    public boolean addScore(double score) { . . . }
    public double finalScore() { . . . }
}
```

Array or ArryList?

Student class contains array of quiz scores and other methods to process the quiz scores.

How about Quiz class?

A second class, ScoreAnalyzer, is responsible for reading the user input and displaying the result. Its main method simply calls the Student methods:

```
Student fred = new Student(100);
System.out.println("Please enter values, Q to quit:");
while (in.hasNextDouble())
{
    if (!fred.addScore(in.nextDouble()))
    {
        System.out.println("Too many scores.");
        return;
    }
}
System.out.println("Final score: " + fred.finalScore());
```

ScoreAnalyzer class
contains a main method

A final quiz score is computed by adding all the scores, except for the **lowest one**. (cont.)

- **Step 4:** Assemble and **test** the program
 - Place your methods into a class. Review your code and check that you handle both **normal** and **exceptional** situations.
 - What happens with an empty array? One that contains a single element? When there are multiple quizzes with the minimum score? Consider these boundary conditions and make sure that your program works correctly

Test Case	Expected Output	Comment
8 7 8.5 9.5 7 5 10	50	See Step 1.
8 7 7 9	24	Only one instance of the low score should be removed.
8	0	After removing the low score, no score remains.
(no inputs)	Error	That is not a legal input.

Outcomes of this lecture – Designing Classes

- Can **identify** *classes* from the given problem
- Can **design** good *methods* for each class to solve the given problem
- Can **implement** a *program* to solve the given problem

1. Discovering Classes

- Designing a class to solve a problem can be challenging. It is not straightforward to tell whether this class is good or not.
- The most important thing is a good class should **represent a single concept** from a problem domain. For example,
 - Classes for a concept from mathematics: Point, Rectangle
 - Classes are abstractions of real-life entities: Student, BankAccount, Bank
- And classes should also work well together.
 - E.g., Bank contains arraylist of BankAccount

1.1 Categories of Classes

- The name of the class should be now that describe a concept. There are many kinds of classes

Entity
Classes

Actor
Classes

Utility
Classes

1.1 Categories of Classes (cont.)

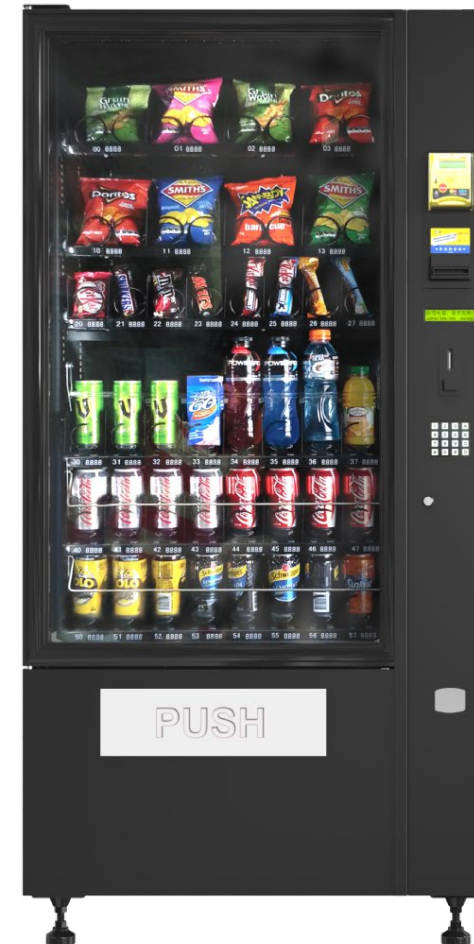
- The name of the class should be now that describe a concept. There are many kinds of classes
- **Entity classes.** Their objects hold data items and perform data manipulation. For example,
 - `Student` object contains array of quiz scores. You can add a new score and compute final quiz score.
 - `BankAccount` object contains account number and balance. You can deposit or withdraw money from the balance.
- **Actor classes.** Their objects carry out relevant tasks. For example,
 - `Scanner` object reads input from a stream for us (see Lecture 5),
 - `RandomGenerator` object generates random numbers,
- **Utility classes.** A class with no object. It only contains a collection of related methods and constants. For example,
 - `Math` contains many mathematical methods (`log(...)`) and constants (`PI`)
 - `Arrays` contains many *static* methods for manipulating arrays such as `toString(...)` or `copyOf(...)`.

1.2 A Case Study – Vending Machine

- You have to write a program of the vending machine and compute the change.
- **Mistake 1:** A class for everything

```
public class VendingMachineProgram {  
    ...  
}
```

This class may contain many responsibilities e.g., keeping data of each item in the machine, keeping how many items are there, letting user choose products, computing change, and etc. There are many concepts in this program, so it should be broken down into some smaller classes.



1.2 A Case Study – Vending Machine (cont.)

- **Mistake 2:** A class for a single operation

```
public class ComputeChange {  
    ...  
}
```

This class cannot be visualized as an object. And it is not a noun :P

- **Mistake 3:** Choosing the wrong level of abstraction

```
public class OrangeJuiceBrandXXX {  
    ...  
}
```

The class should represent a **kind of things**, but not a specific things. **Book** could be a good class, but **JavaConceptBook** couldn't be a class. It should be an *objet* of the **Book** class.

1.2 A Case Study – Vending Machine (cont.)

- A good design class could be:

```
public class Item {...}
```

```
// represent a item's data such as brand, price, etc.
```

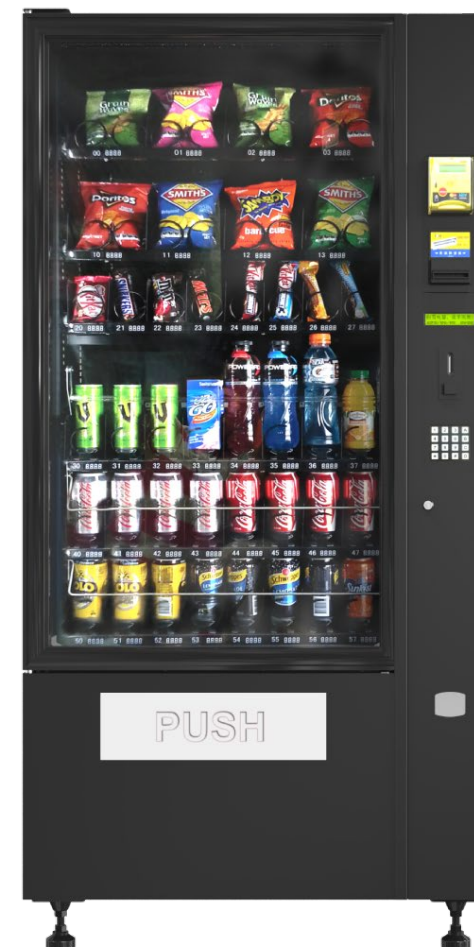
```
public class VendingMachine {...}
```

```
// represent a machine to interact with customers
```

```
// such as greeting message, compute change, etc.
```

```
public class Inventory {...}
```

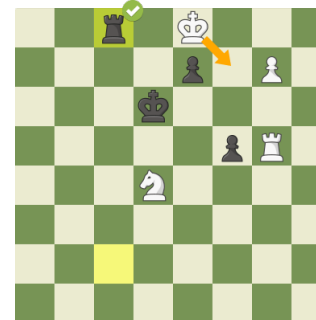
```
// represent items stock
```



Check Point – Class or Not a class



Chess

1. `public class ChessBoard { . . . }`
2. `public class MovePiece { . . . }`



Shopping Cart

3. `public class AddItemIntoCart { . . . }`
4. `public class VNeckTShirt { . . . }`
5. `public class OrderItem { . . . }`
6. `public class Order { . . . }`

CURRENTLY IN YOUR SHOPPING CART			
YOUR ITEMS	ITEM PRICE	QUANTITY	PRICE
 Lacoste S/S Pima Jersey V-Neck T-Shirt SKU: #7691910 COLOR: BLACK SIZE: M (EUR 5)	\$49.50	<input type="text" value="1"/> UPDATE Remove Move to Favorites	\$49.50
 Original Penguin Melt SKU: #7977382 COLOR: TOTAL ECLIPSE SIZE: 8.5 WIDTH: D	\$99.00 ONLY 3 LEFT! This item may become unavailable if someone else purchases it.	<input type="text" value="1"/> UPDATE Remove Move to Favorites	\$99.00
SUBTOTAL (2 ITEMS):			\$148.50
STANDARD NEXT BUSINESS DAY SHIPPING:			FREE
* ESTIMATED TAX TO BE COLLECTED:			\$0.00
GRAND TOTAL:			\$148.50

Check Point – Class or Not a class

Chess



1. `public class ChessBoard { . . . }`



2. `public class MovePiece { . . . }`



Shopping Cart



3. `public class AddItemIntoCart { . . . }`





4. `public class VNeckTShirt { . . . }`



5. `public class OrderItem { . . . }`



6. `public class Order { . . . }`

CURRENTLY IN YOUR SHOPPING CART			
YOUR ITEMS	ITEM PRICE	QUANTITY	PRICE
 Lacoste S/S Pima Jersey V-Neck T-Shirt SKU: #7691910 COLOR: BLACK SIZE: M (EUR 5)	\$49.50	<input type="text" value="1"/> UPDATE Remove Move to Favorites	\$49.50
 Original Penguin Melt SKU: #7977382 COLOR: TOTAL ECLIPSE SIZE: 8.5 WIDTH: D	\$99.00 ONLY 3 LEFT! This item may become unavailable if someone else purchases it.	<input type="text" value="1"/> UPDATE Remove Move to Favorites	\$99.00
SUBTOTAL (2 ITEMS):			\$148.50
STANDARD NEXT BUSINESS DAY SHIPPING:			FREE
* ESTIMATED TAX TO BE COLLECTED:			\$0.00
GRAND TOTAL:			\$148.50

2. Designing Good Methods

- There are several useful criteria to consider when designing the public interface of a class.
 - Cohesive
 - Dependencies
 - Accessors and Mutators
 - Side Effects



2.1 Cohesive Public Interface

- All public interface of a class are related to the concept that the class represents. **A public interface is said to be “cohesive” if it related to the single concept of the class.**
 - The more focused a class is, the more is the cohesiveness of that class. – Easier to maintain
- If you find that the public interface refers to multiple concepts, then you should separate the class.
- For example, the vending machine takes the payment from the customer (assuming that the machine can only take coins)

```
public class VendingMachine {  
    public static final double QUARTER_VALUE = 0.25;  
    public static final double DIME_VALUE = 0.1;  
    public static final double NICKEL_VALUE = 0.05;  
    // . . .  
    public void receivePayment(int dollars, int quarters,  
                                int dimes, int nickels, int pennies) {  
        // . . .  
    }  
    // . . .  
}
```

The **receivePayment** method has two concepts:

1. Taking in coins and compute total
2. The value of individual coin

What if we have more Coin Type? So it might make sense to have another **Coin** class to responsible for knowing their values.

Separate Two Concepts to Different Classes

```
public class Coin
{
    public Coin (double aValue, String aName) {
        // . . .
    }
    public double getValue() {
        // . . .
    }
}
```

```
public class VendingMachine {
    // . . .
    public void receivePayment(int coinCount, Coin coinType)
    {
        payment = payment + coinCount * coinType.getValue();
    }
    // . . .
}
```

Now the VendingMachine class no longer needs to worry about coin values. This class can later handle **Thai** coin too!

This is clearly a better solution, because it separates the responsibilities of the the vending machine and the coins.

```
// Java program to illustrate  
// high cohesive behavior
```

```
class Multiply {  
  
    int a = 5;  
    int b = 5;  
  
    public int mul(int a, int b)  
    {  
        this.a = a;  
        this.b = b;  
        return a * b;  
    }  
}  
  
class Display {  
    public static void main(String[] args)  
    {  
        Multiply m = new Multiply();  
        System.out.println(m.mul(5, 5));  
    }  
}
```

```
// Java program to illustrate  
// low cohesive behavior
```

```
class Display {  
    public static void main(String[] args)  
    {  
        int a = 5;  
        int b = 5;  
        int mul = a * b;  
        System.out.println(mul);  
    }  
}
```

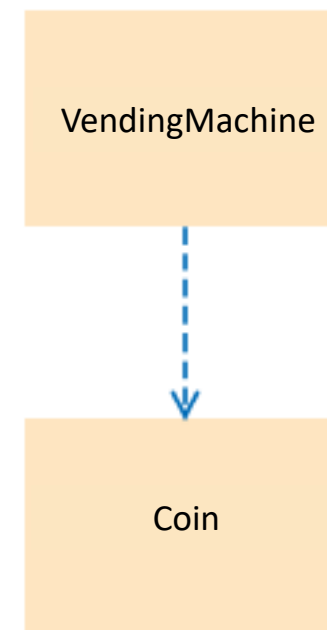
This is an example of a low cohesive class because the window/print console and the multiplication operation don't have much in common.

2.2 Minimizing Dependencies

- Many methods need other classes in order to do their job.

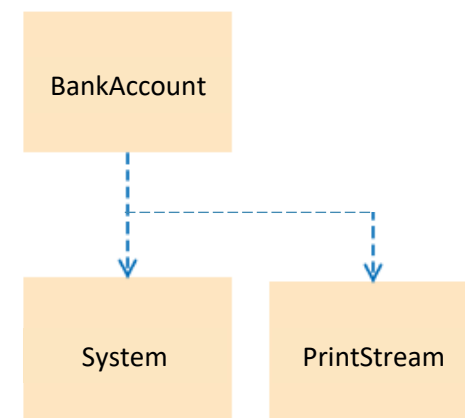
```
// Vending Machine Example  
public void receivePayment(int coinCount, Coin coinType)  
{  
    payment = payment + coinCount * coinType.getValue();  
}
```

- We say that the **VendingMachine** class depends on the **Coin** class since its method (receivePayment) uses a Coin class.
- When we design a public interface, we should try to minimize the dependencies.



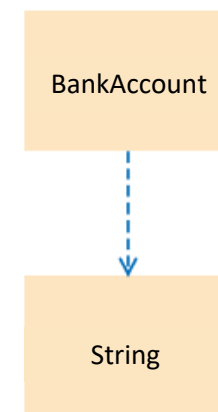
Bank Account Example

```
public class BankAccount {  
    // . . .  
    public void showAccountSummary() {  
        System.out.println(this.accountNumber + " has " + this.balance + "baht.");  
    }  
    // . . .  
}
```



- This is why we **do not usually "print" in a class method**. We should just return a String instead:

```
public class BankAccount {  
    // . . .  
    public String showAccountSummary() {  
        return this.accountNumber + " has " + this.balance + "baht.";  
    }  
    // . . .  
}
```



2.3 Separating Accessors and Mutators

- **Mutator method:** changes the state of an object [change object's instance variables], and **without returning value (return void)**.
- **Accessor method:** asks on object to compute a result, **without changing the state**.
- *Recall: **String** is an *immutable class* since it has only accessor methods.*
- An immutable class is more safe and suitable for representing values such as strings, dates, colors, and so on. No one can change the object's value. However, not every class should be immutable.

```
String name = "John Q. Public";
```

```
String uppercased = name.toUpperCase(); // Accessor: name is not changed  
// uppercased -> JOHN Q. PUBLIC
```

Sometimes, this rule is bent a bit . . .

- In mutable classes, it is still a good idea to cleanly separate accessors and mutators, in order to avoid accidental mutation.
- **Rule of thumb:** a method that **returns a value should not be a mutator**. For example
 - One would **NOT** expect that calling **getBalance** on BankAccount object would change the balance.

```
ArrayList<String> names = new ArrayList<>();  
names.add("Lucy");           // Mutator: change state, no return  
String name = names.get(0);  // Accessor: return string, do not change anything  
boolean exist = names.contains("Lucy"); // Accessor: return boolean, do not change anything  
boolean success = names.remove("Lucy"); // Mutator: change state (remove string), but also return!
```

Here `.remove(. . .)` **break** the “separating” rule! This method return true if the removal was successful. This might be a bad design. However, it is okay to return the success status after changing the state.

2.4 Minimizing Side Effects

- A **side effect** of a method is any kind of modification of data that is **observable outside** the method.
- Mutator methods usually have a side effect, and it is okay. However, you should avoid methods that modify its parameter variables.

```
/**
 * Computes the total balance of the given accounts.
 */
public double getTotalBalance(ArrayList<BankAccount> accounts)
{
    double sum = 0;
    while(accounts.size()>0)
    {
        BankAccount account = accounts.remove(0); //Not recommended
        sum = sum + account.getBalance();
    }
    return sum;
}
```

This method removes all BankAccount from the accounts parameter variable

```
public class BankAccount {
    // . . .
    public void showAccountSummary() {
        System.out.println(this.accountNumber
            + " has " + this.balance + "baht.");
    }
    // . . .
}
```

This method modifies System.out object which is not a part of BankAccount object.

Check Point – Good or Bad Method

Which statements are correct?

- Methods with high cohesion is that such classes are much easier to maintain.
- It is a good idea to have “System.out.println(…)” in the instance method
- A method that return value must be a mutable method
- You should try to avoid method that modifies its parameter variables
- String and wrapper classes (i.e., Integer, Boolean, Byte, Short) are immutable class

```
String food = "noodle;salad;snack";  
String[] foodList = food.split(";");  
System.out.println(foodList.length); // 3  
System.out.println(food);           // Accessor: food is not changed
```

Check Point – Good or Bad Method

Which statements are correct?

- ✓ • Methods with high cohesion is that such classes are much easier to maintain.
- ✗ • It is a good idea to have “System.out.println(…)” in the instance method
- ✗ • A method that return value must be a mutable method
- ✓ • You should try to avoid method that modifies its parameter variables
- ✓ • String and wrapper classes (i.e., Integer, Boolean, Byte, Short) are immutable class

```
String food = "noodle;salad;snack";  
String[] foodList = food.split(";");  
System.out.println(foodList.length); // 3  
System.out.println(food);             // Accessor: food is not changed
```

3. Parameters

```
public class Car{  
    private String color;  
    private int price;  
  
    public Car(String c, int p){  
        color = c;  
        price = p;  
    }  
  
    public void setColor(String c){  
        color = c;  
    }  
}
```

```
public class CarTester{  
    public static void main(String[] args){  
        Car myCar = new Car("white",220000);  
        myCar.setColor("black");  
    }  
}
```

Car

color = white
price = 220000

setColor("black")

c = black
color = ~~white~~black

Explicit
Parameter

Implicit
Parameter

3.1 Explicit Parameters

- Recall that **a parameter is a value that is given to a method as input.**
- Methods can have one or more parameters.
- TWO different kinds of parameters:
 - **Explicit parameter:** that is passed by specifying the parameter in the parenthesis of a method call.

```
System.out.println("Java is FUN!");  
int price = 100;  
System.out.println("The price is " + price);  
myCar.setColor("black");
```

3.2 Implicit Parameters

- **Implicit parameter:** the **object** on which the method is invoked (object reference before the name of a method).

```
Car myCar = new Car("gray", "2000");  
myCar.setColor("white");
```

```
Car momCar = new Car("red", "3000");  
momCar.setColor("black");
```

```
public class Car{  
    String color;  
    int price;  
    ...  
    public void setColor(String c){  
        color = c;  
    }  
}
```

```
public class Box{  
    double width, height;  
    ...  
    public double getArea(){  
        return width * height;  
    }  
}
```

```
Box myBox = new Box();  
myBox.getArea();
```


Implicit Parameters

```
class BankAccount{  
    double balance;  
    ...  
    public void withdraw(double amount) {  
        double newBalance = balance - amount;  
        balance = newBalance;  
    }  
}
```

`balance` is the balance of the object to the left of the dot

```
momsSavings.withdraw(500);
```

Compiler will translate the above statement to

```
double newBalance = momsSavings.balance - amount;  
momsSavings.balance = newBalance;
```

Implicit Parameters and **this**

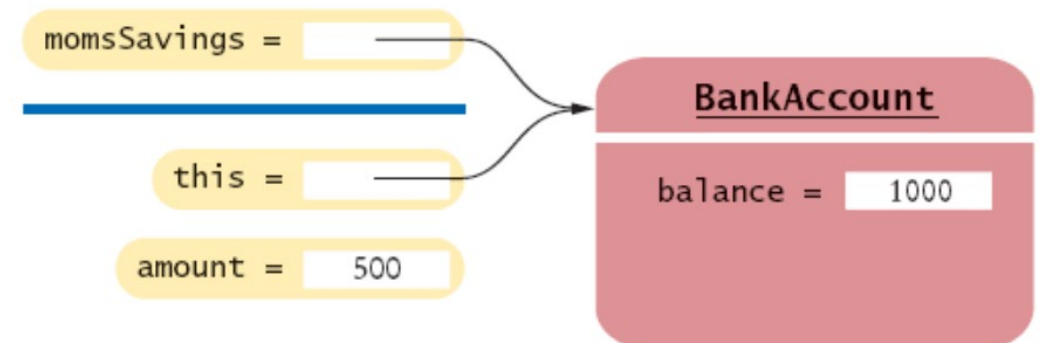
- The **this** reference denotes the implicit parameter. *One method can have only one implicit parameter.*
- Here is the deposit method in the BankAccount Class

```
public void deposit (double amount) {  
    balance = balance + amount;  
}
```

```
public void deposit (double amount) {  
    this.balance = this.balance + amount;  
}
```

momsSavings.deposit(500);

****Java assumes that a variable name that are not an explicit *parameter* or a *local* variable must refer to an instance variable. That's why you don't need to write `this` before the variable `balance`**



Implicit Parameters and **this** (cont.)

- To avoid confusion. You MUST use **this** to refer to an *instance variable* **when** you have the same variable name for *explicit parameter* or *local variable*.

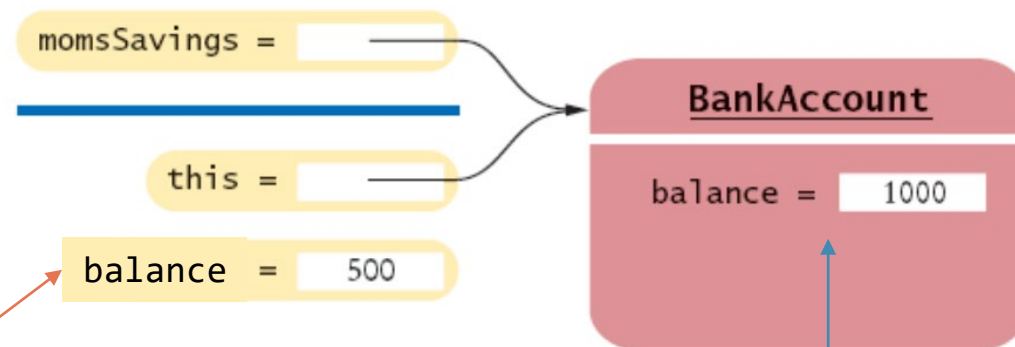
momsSavings.deposit(500);

```
public void setBalance(double balance){  
    balance = balance;  
}
```



NOT a syntax error but a logical error

```
public void setBalance(double balance){  
    this.balance = balance;  
}
```



momsSavings.setBalance(500);

3.3 Primitive Type Variables as Parameters

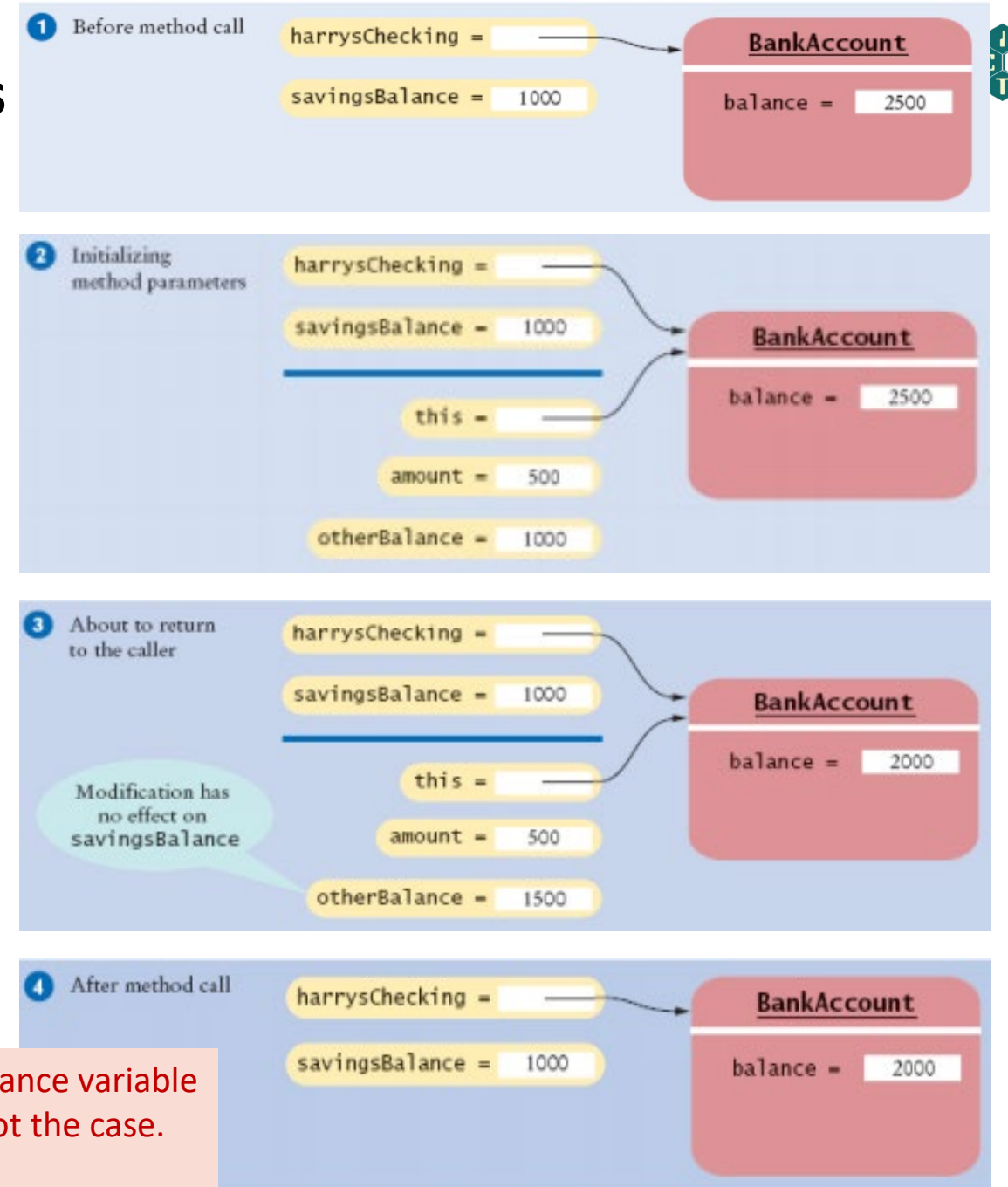
This code is in the `main` method

```
BankAccount harrysChecking = new BankAccount();  
harrysChecking.balance = 2500;  
double savingsBalance = 1000;  
harrysChecking.transfer(500, savingsBalance);  
System.out.println(savingsBalance);
```

This method is in the `BankAccount` Class

```
...  
void transfer(double amount, double otherBalance) {  
    balance = balance - amount;  
    otherBalance = otherBalance + amount;  
}
```

You might expect that after the call, the `savingsBalance` variable has been incremented to 1500. However, that is not the case.
-> **passed by value**



3.4 References Variables as Parameters

This code is in the `main` method

```
BankAccount harrysChecking = new BankAccount();  
harrysChecking.balance = 2500;
```

```
BankAccount harrysSavings = new BankAccount(1000);  
harrysChecking.transfer(500, harrysSavings);
```

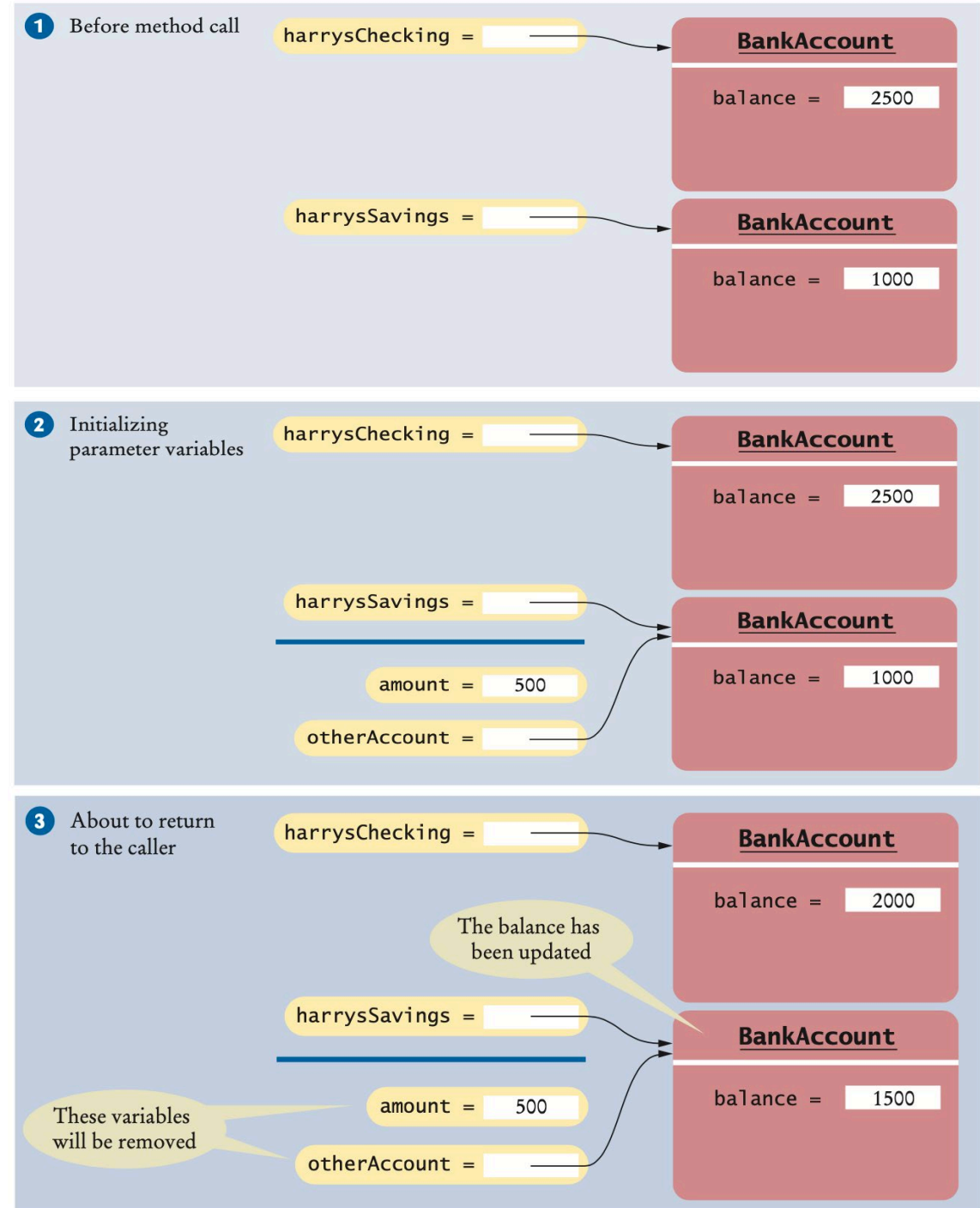
```
System.out.println(harrysSavings.getBalance());
```

This method is in the `BankAccount` Class

```
...  
void transfer(double amount, BankAccount otherAccount) {  
    balance = balance - amount;  
    otherAccount.deposit(amoung);  
}
```

Methods can mutate any objects to which they hold referenes
-> **passed by value (reference to the same object)**

Note that, methods cannot replace the object with another object
`otherAccount = new BankAccount(newBalance); // Won't work`



4. Pattern for Object Data

- When you implement the class, you need to come up with the instance variables for the class. It is not always obvious how to do this.
- Fortunately, there is a small set of **recurring patterns** that you can adapt when you design your own classes
 - 4.1 Keeping a Total
 - 4.2 Collecting values
 - 4.3 Managing Properties
 - 4.4 Tracking Distinct States

4.1 Keeping a total

Many classes need to keep track of a quantity:

- A bank account keeps a balance.
- A counter keeps a count value.
- A car keeps an amount of gas and distance traveled.

💡 When a class need to keep track of a quantity, there should be an **instance variable** for the total. **Methods** then update this variable such as add amount, clear (set to 0), and get amount.

```
public class Counter {  
    private int value;    // keeping the total  
    // . . .  
    public void click() {  
        this.value++;    // update (increase)  
    }  
    public void reset() {  
        this.value = 0;    // update (decrease)  
    }  
}
```

4.2 Collecting values

Some objects collect other objects:

- An order has many order items.
- A student has many enrolled courses.
- A vending machine has many items.

💡 When a class need to collect values, there should be an instance variable of type **array** or **array list**.

Methods then update this array or array list.

```
public class Student {  
    private String name;  
    private String studentId;  
    private ArrayList<Enrollment> enrollments; // collection variable  
  
    public Student(String studentId, String name) {  
        this.studentId = studentId;  
        this.name = name;  
        this.enrollments = new ArrayList<Enrollment>(); // initialized in constructor  
    }  
    // . . .  
    public void enroll(Enrollment e) {  
        this.enrollments.add(e); // update the array list  
    }  
    // . . .  
}
```


4.3 Managing Properties

Most classes have to store information in a form of instance variables, but the behaviors can be different from what being store:

- Student has firstName and lastName, but only getName() is available.
- BankAccount has accountNumber, but cannot be changed (no setter).

💡 Instance variables are accessed with a getter method and changed with a setter method. The getter and setter methods allow us to manage the properties of an object.

```
public class Student {  
    private String name;  
    private String studentId;  
    private ArrayList<Enrollment> enrollments;  
  
    // . . .  
    public void setName(String newName) {  
        if (newName.length() > 0) { this.name = newName; } // error checking  
    }  
    // . . .  
    public String getStudentId() { return this.studentId; }  
    // no setter, Student ID cannot be changed.  
}
```

4.4 Tracking Distinct States

Some object behavior differently depending on what happened in the past:

- Fish may look for food when it is hungry and ignore food after it ate.
- Student may not be able to enroll if he/she already graduated.
- A Ghost AI in Pacman will run away if it is in a "fear" state.

💡 If an object can have one of many states that affect the behavior, there should be an instance variable for the current state.

```
public class Fish {  
    private int hungry; // current state  
  
    // state values (constants)  
    public static final int NOT_HUNGRY = 0;  
    public static final int SOMEWHAT_HUNGRY = 1;  
    public static final int VERY_HUNGRY = 2;  
    // . . .  
    public void eat() {  
        this.hungry = NOT_HUNGRY; // state update  
        // . . .  
    }  
    // . . .  
    public void move() { // behavior depends on the current state.  
        // . . .  
        if (this.hungry < VERY_HUNGRY) { this.hungry++; } // state update  
        if (this.hungry == VERY_HUNGRY) { // look for food }  
    }  
}
```

5. Static Variable and Methods

- Sometimes, a value properly belongs to a class, not to any object of the class. For example, we want to assign bank account numbers sequentially.
 - Account number: 1001
 - Account number: 1002
 - Account number: 1003
- To solve this problem, we need to have a single value of `lastAssigned Number` that is a property of the *class*, not any object of the class. Such a variable is called a static variable because you declare it using the **static** keyword.
- In OOP, the use of **static** variable and methods are **not** very common.
- The **main** method is always static.

5.1 Static Variable

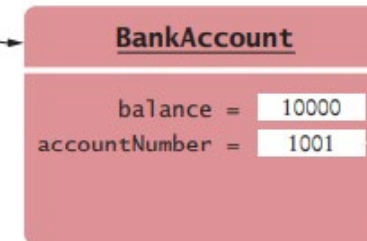
Use case: Using constructor method to increase the account number by one and automatically assign account number

```
public class BankAccount {
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber = 1000; // here!
    public static final double OVERDRAFT_FEE = 29.95;
    public BankAccount(double balance) {
        lastAssignedNumber++;
        this.accountNumber = lastAssignedNumber;
        this.balance = balance;
    }
    // . . .
}
```

```
BankAccount collegeFund = new BankAccount(10000);
BankAccount momsSavings = new BankAccount(8000);
BankAccount harrysChecking = new BankAccount(0);
System.out.println(BankAccount.OVERDRAFT_FEE); // prints 29.95
System.out.println(momsSavings.OVERDRAFT_FEE); // prints 29.95, but not recommended
```

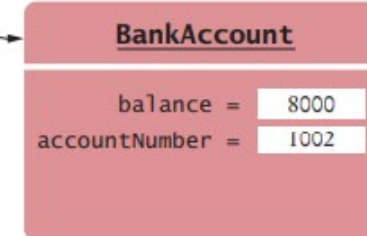
Note: Static fields should always be declared as **private**
Except constant value (final), which may be either private or public.

collegeFund =

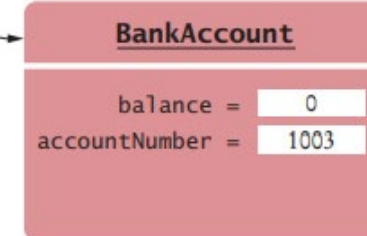


Each BankAccount object has its own accountNumber instance variable.

momsSavings =



harrysChecking =



There is a single lastAssignedNumber static variable for the BankAccount class.

BankAccount.lastAssignedNumber = 1003

Calling this static variable with the name of the class containing it.

5.2 Static Method

- Sometimes a class defines methods that are not invoked on an object. Such a method is called a static method.
- The static method is commonly used of instructions **that result does not depend on the attributes of an object**.
- A typical example of a static method is the sqrt method in the **Math** class.
 - Number, like integer, is not an object, so we cannot invoke methods of it
 - Math class allow you to work with number, such as `Math.sqrt(5)`; No object of the Math class is constructed.
- You can make your static method for use in other classes as well.

```
public class BankAccount {  
    // . . .  
    public static boolean isRicher(BankAccount a, BankAccount b) {  
        return a.getBalance() > b.getBalance();  
    }  
    // . . .  
}
```

```
BankAccount x = new BankAccount(10000);  
BankAccount y = new BankAccount(8000);  
boolean r1 = BankAccount.isRicher(x, y); // true
```

```
public class Financial {  
    public static double percentOf(double p, double a) {  
        return (p / 100) * a;  
    }  
    // More financial methods can be added here.  
}
```

```
double tax = Financial.percentOf(taxRate, total);
```

Calling this static method with the name
of the class containing it.

5.3 Common Error

- Static methods belong to its class, not any object. There is **no implicit this parameter** here. So you cannot directly access any instance variables or methods that are not static.

```
public class SavingsAccount {  
    private double balance;  
    private double interestRate;  
  
    public static double interest(double amount) {  
        return (interestRate / 100) * amount;  
        // ERROR: Static method accesses instance variable (interestRate)  
    }  
}
```

6. Designing class using UML

- The **UML** Class diagram is a graphical notation used to construct and visualize object-oriented systems.
 - Programming languages are not abstract enough for OO design
 - UML is an open standard; lots of companies use it
- **Union of all Modeling Languages**
 - Use case diagrams
 - **Class diagrams**
 - Object diagrams
 - Sequence diagrams
 - Collaboration diagrams
 - Statechart diagrams
 - Activity diagrams
 - Component diagrams
 - Deployment diagrams

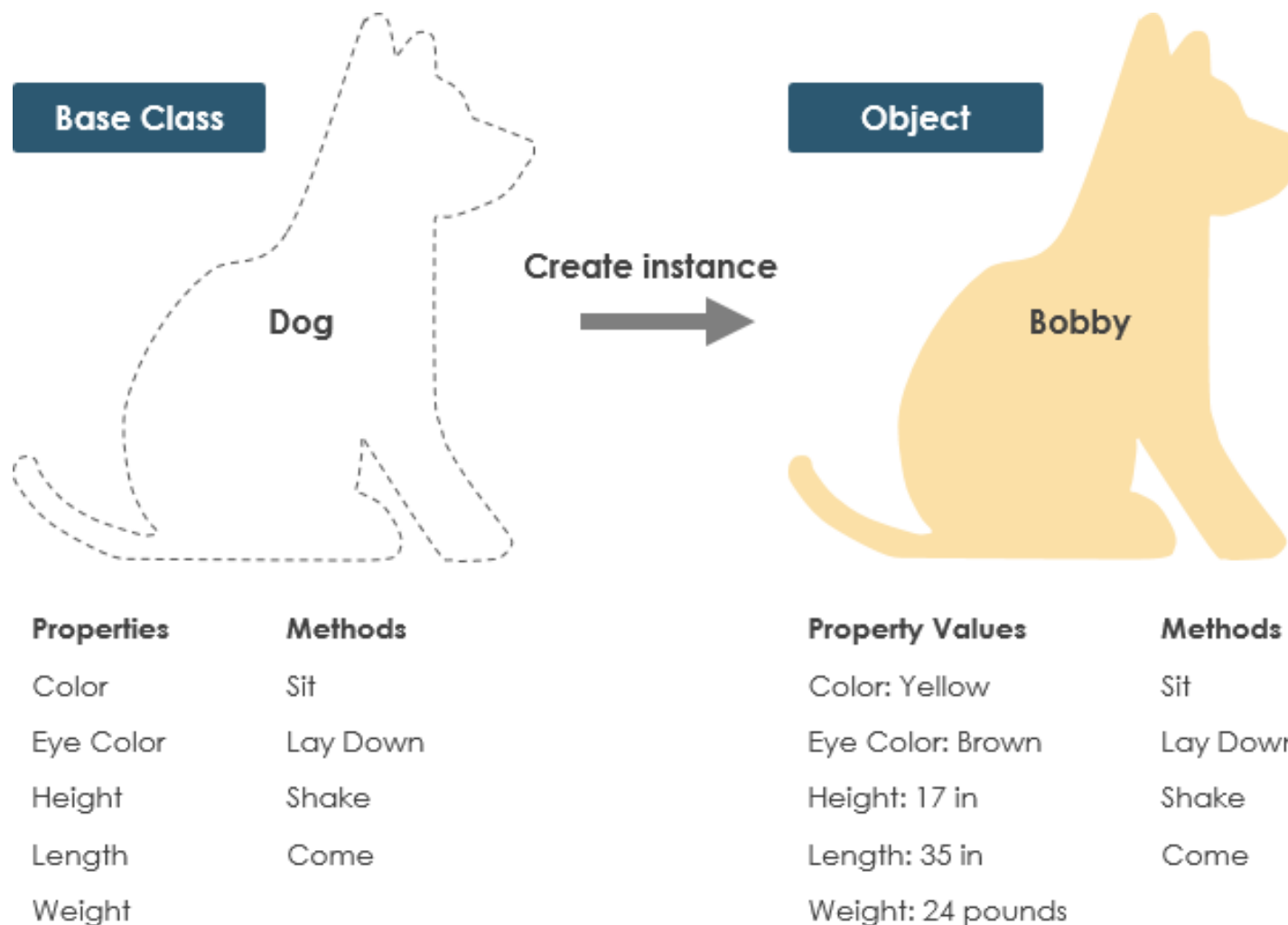
6.1 UML class diagrams

- **UML class diagram** is a picture of:
 - the **classes** in an OO system
 - their **fields** and **methods**
 - **Connections/Relationships** between the classes

Recall: Class vs Object

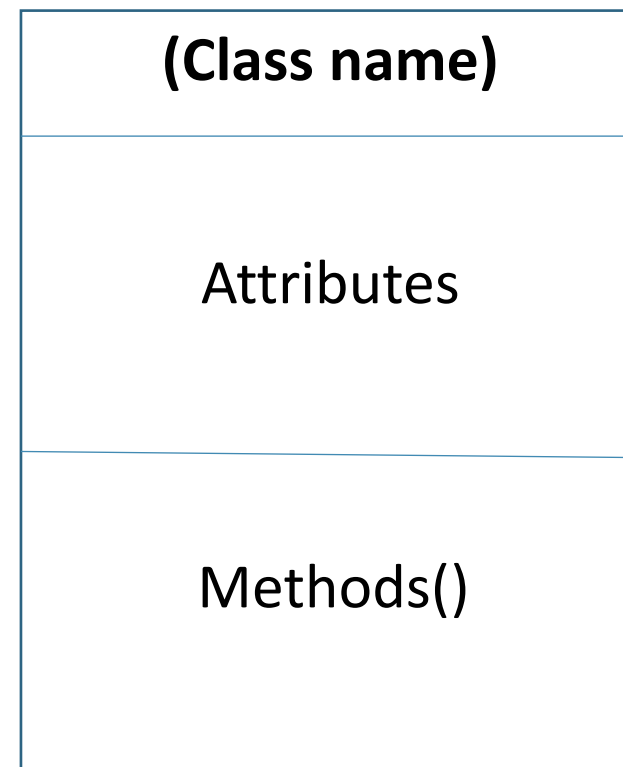
A dog has **states** – color, name, breed as well as **behaviors** – wagging, barking, eating.

“An object is an instance of a class.”



6.2 Basic Diagram of one class

- **Class** name in top of box
- **Attributes** (optional)
 - should include all fields of the object
- **Methods** (optional)



Class Notation – Attributes and Methods

- Each **attribute** has a type
 - **Format** -> name : type [count] = default_value (if any)
 - underline static attributes

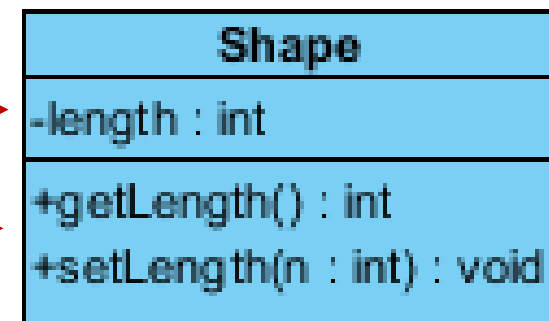
- attribute example:

balance : double = 0.00

bookList : Book [0...*]

Attribute(s) →

Method(s) →

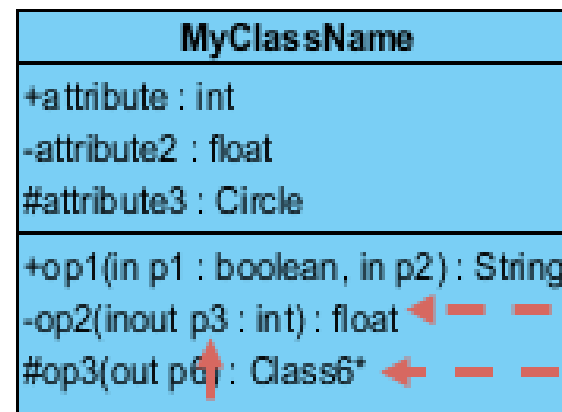


- Each **method** has a signature
 - Signature is the method parameters and return types
 - **Format** -> name (parameters) : return_type

Class Notation – Attributes and Methods (cont.)

- Each **operation/method** has a signature
 - Signature is the method parameters and return types
 - **Format** -> name (**parameters**) : return_type
 - underline static methods
 - parameter types listed as (**name**: **type**)
 - omit return_type on *constructors* and when return type is *void*
 - method example:

```
distance(p1: Point, p2: Point): double  
printInfo()
```



Parameter p3 of op2 is of type int

Class Visibility (Access Specifiers)

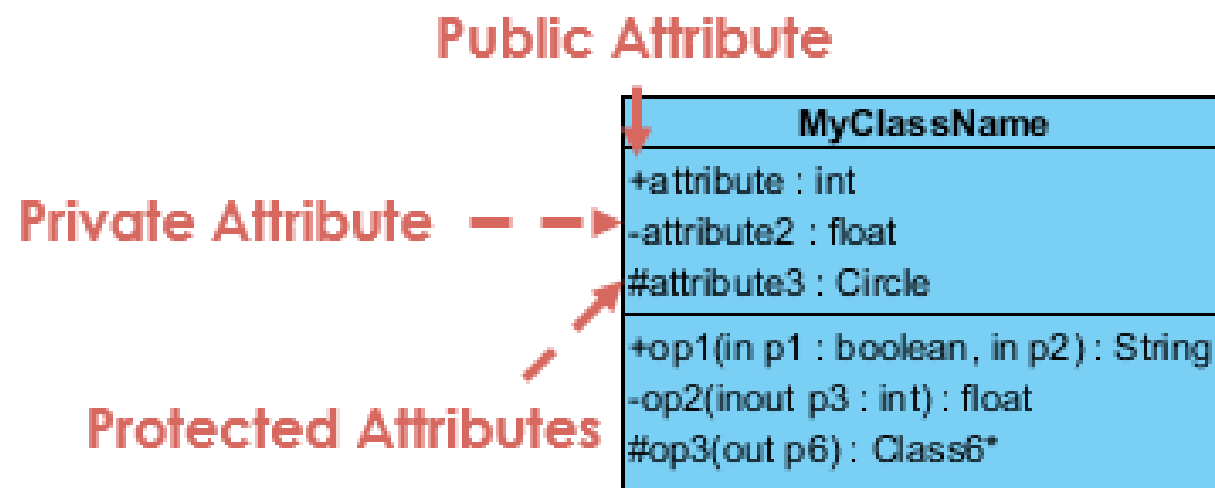
- The +, -, # and ~ symbols before an attribute and operation name in a class denote the visibility of the attribute and method.

visibility: + public (everywhere)

 # protected (package + subclass)

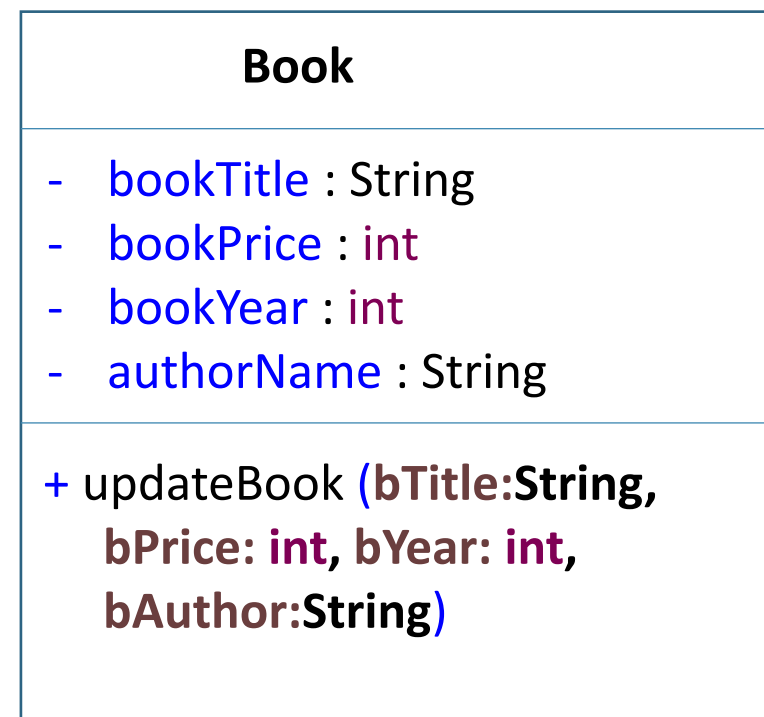
 ~ default (package)

 - private (own class)



UML for bookstore (Example)

- **Class**
 - Book
- **Attributes**
 - Book's title
 - Book's price
 - Year of publication
 - Author's name
- **Method**
 - Update



Exercise (1)

... Implement a class Product. A product has a name and a price. Supply methods getName, getPrice, and reducePrice ...

For testing (later in the lab)

Implement a program ProductPrinter that makes two products, prints the name and price, reduces their prices by 5.00%, and then prints the prices again.

Exercise (2)

... Implement a class Student. For the purpose of this exercise, a student has a name and a total quiz score.

Supply an appropriate constructor and methods
getName(), addQuiz(int score), getTotalScore(), and
getAverageScore() ...

Notice

To compute the average score, you also need to store the number of quizzes that the student took.

6.3 Multiple classes usage

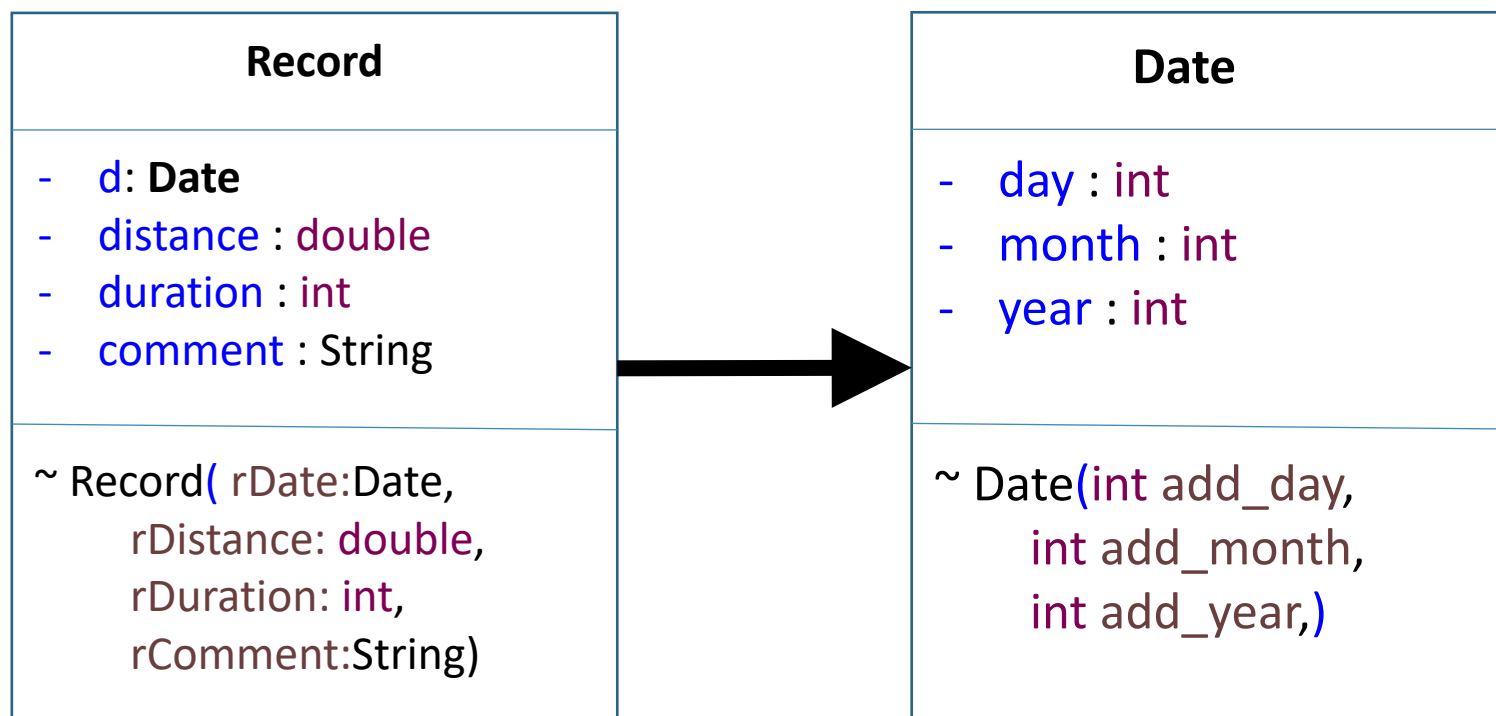
- Sometime you need to deal with a problem that need more than one class.

For example:

... Develop a program that manages a runner's training log. Every day the runner enters one record about the day's run. Each record includes the day's date, the distance of the day's run, the duration of the run, and a comment describing the runner's post-run ...

Multiple classes usage

... Develop a program that manages a runner's training log. Every day the runner enters one record about the day's run. Each record includes the day's date, the distance of the day's run, the duration of the run, and a comment describing the runner's post-run ...



Record.java

```
class Record {  
  
    Date d;  
    double distance;  
    int duration;  
    String comment;  
  
    Record(Date rDate, double rDistance,  
           int rDuration, String rComment)  
    {  
        d = rDate;  
        distance = rDistance;  
        duration = rDuration;  
        comment = rComment;  
    }  
  
}
```

Date.java

```
class Date {  
  
    int day;  
    int month;  
    int year;  
  
    Date(int add_day,  
         int add_month,  
         int add_year)  
    {  
        day = add_day;  
        month = add_month;  
        year = add_year;  
    }  
  
}
```

RecordTester.java



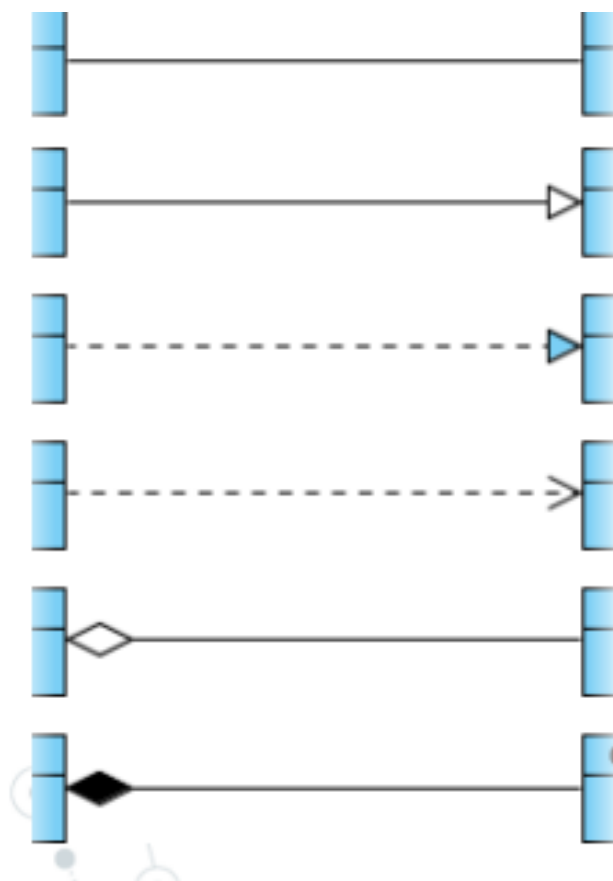
```
public class RecordTester {  
    public static void main(String[] args) {  
        //Code here//  
    }  
  
}
```

Exercise (3)

... Develop a program that helps visitor navigate restaurant in Mahidol Salaya. The program must be able to provide four pieces of information for each restaurant: its name, the type of food it serves, its price range, and the address (street, district, postcode, phone number)...

- Example of data: (1) Mai-tok mai-tak, a Thai restaurant, inexpensive, on Phutthamonthon Sai 4 Salaya 73170 (0977797989).

Relationships Notation – More on this in OOD Lecture



Association

- A structural link between two peer classes.

Inheritance

- Represents an "is-a" relationship.

Realization

- Relationship between interface and Class

Dependency

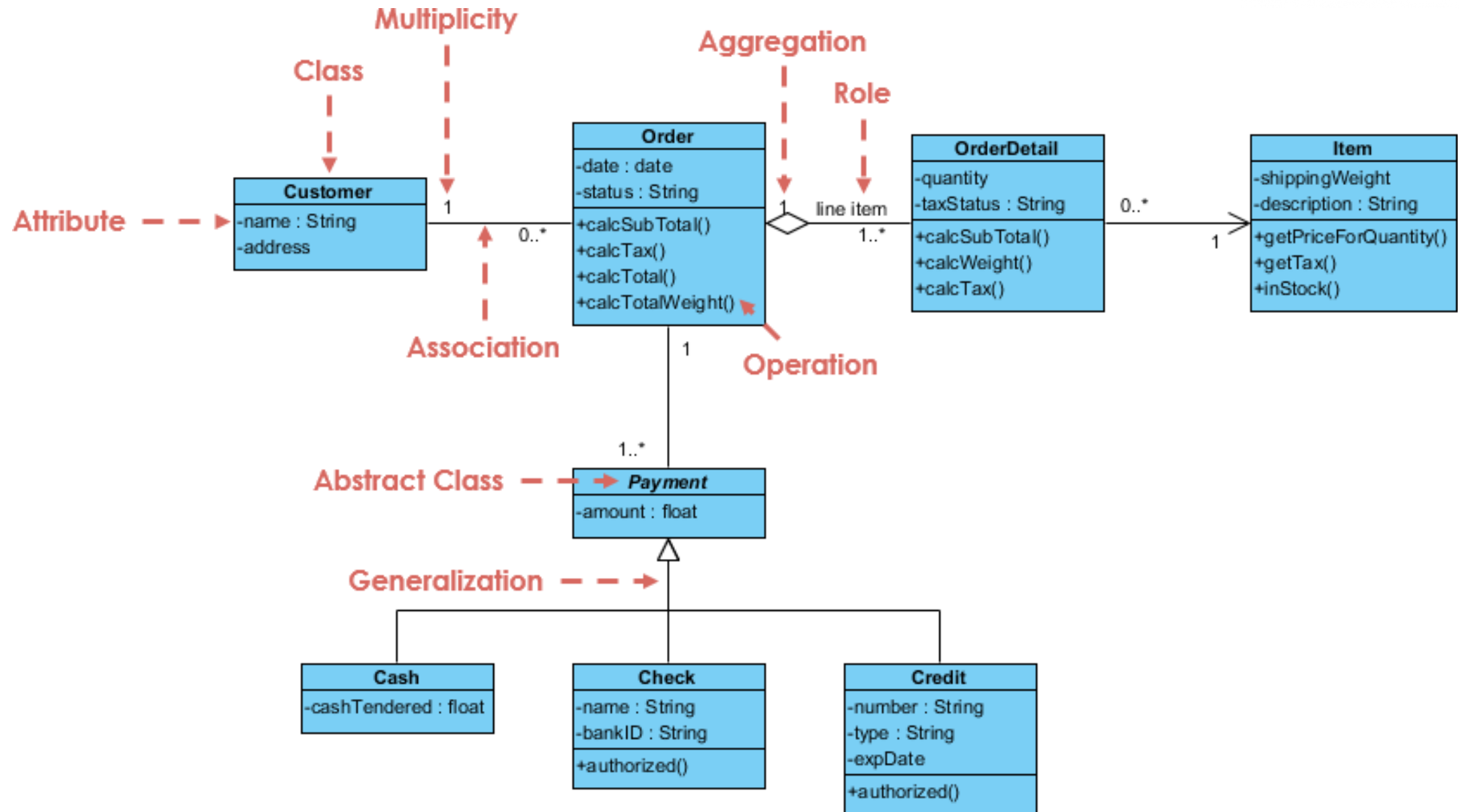
- An object of one class might use an object of another class in the code of a method.

Aggregation

- It represents a "part of" relationship.

Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.





Additional Topic

Pre-condition and Post-Condition

Preconditions

- Precondition is a requirement that the caller of a method must meet
- Publish preconditions so the caller won't call methods with bad parameters

```
/**  
    Deposits money into this account.  
    @param amount the amount of money to deposit  
    (Precondition: amount >= 0)  
*/
```

- Typical use:
 - To restrict the parameters of a method
 - To require that a method is only called when the object is in an appropriate state
- If precondition is violated, method is not responsible for computing the correct result. It is free to do anything .

Preconditions (cont.)

- Method may throw exception if precondition violated – more in Chapter 11

```
if (amount < 0) throw new IllegalArgumentException();  
balance = balance + amount;
```

- Method doesn't have to test for precondition. (Test may be costly)

```
// if this makes the balance negative, it's the caller's fault  
balance = balance + amount;
```

Preconditions (cont.)

- An **assertion** is a technique for checking whether an assertion statement is true.
- If the assertion fails and `assertion` checking is enabled, then the program terminates with an **AssertionError**
- Method can do an assertion check:

```
public double deposit (double amount){  
    assert amount >= 0;  
    balance = balance + amount;  
}
```

- To enable assertion checking:
`java -enableassertions MyProg` or `java -ea MyProg`

To turn assert-statement execution on or off for one run in Eclipse:

1. Select menu item Run -> Run Configurations.
2. In the window that opens, click tab Arguments. The arguments pane will open.
3. In the VM Arguments field, type `-ea` to have the assert statement enabled.
4. Click button Run to run the program.

- You can turn assertions off after you have tested your program, so that it runs at maximum speed
- Many beginning programmers silently return to the caller

```
if (amount < 0) return; // Not recommended; hard to debug  
balance = balance + amount;
```

Postconditions

- Postcondition is a condition that is true **after a method has completed**
- If method call is in accordance with preconditions, it must ensure that postconditions are valid
- There are two kinds of postconditions:
 - The **return value** is computed correctly
 - The object is in a **certain state** after the method call is completed
- Example

```
/**  
    Deposits money into this account.  
    (Postcondition: getBalance() >= 0)  
    @param amount the amount of money to deposit  
    (Precondition: amount >= 0)  
*/
```

Postconditions (cont.)

- Formulate pre- and post-conditions only in terms of the interface of the class.

```
amount <= getBalance() // this is the way to state a postcondition  
                        // for withdraw
```

- The caller, which needs to check the precondition, has access only to the public interface, not the private implementation
- Contract: If caller fulfills precondition, method must fulfill postcondition