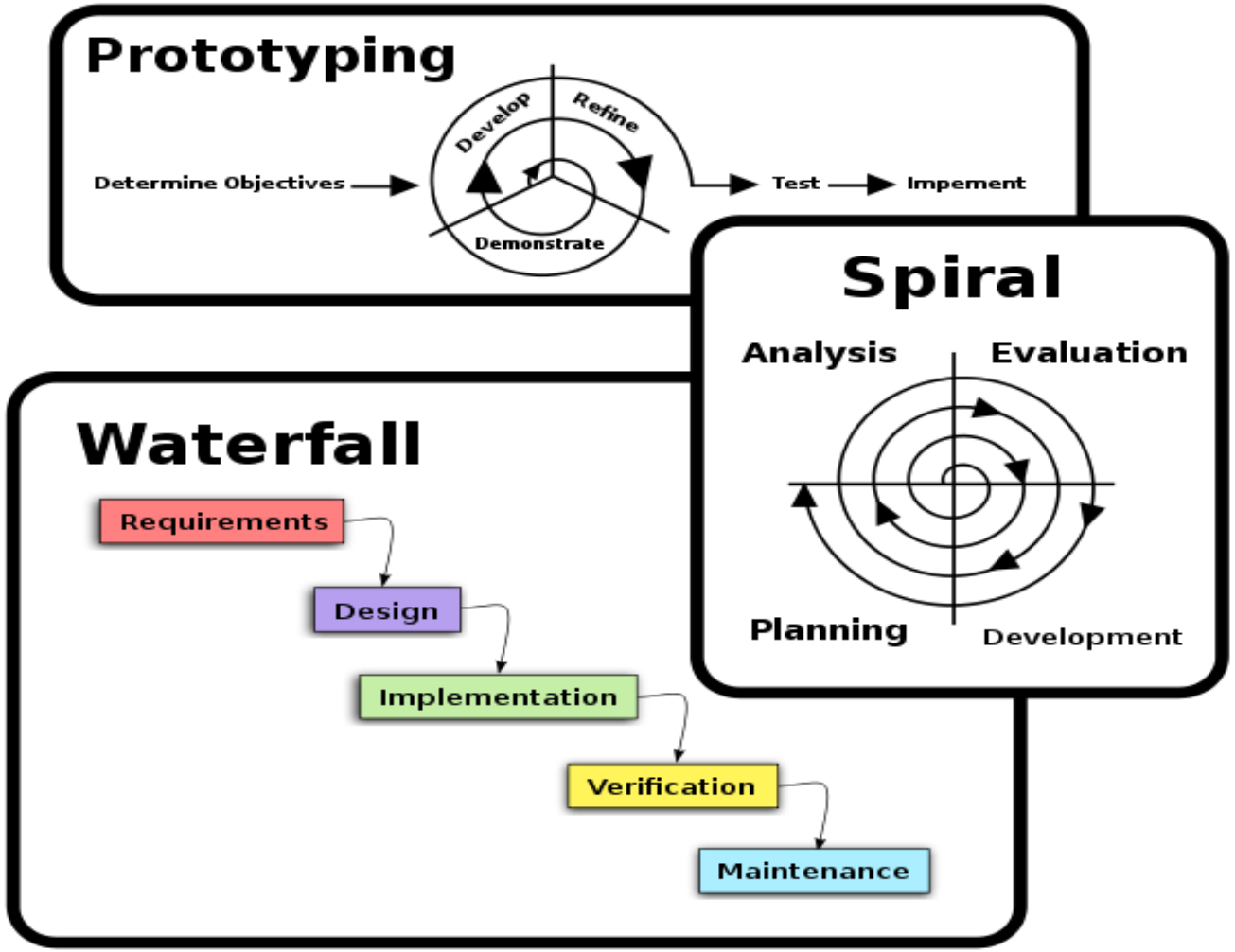
Object Orientation First Story

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/JavaSE

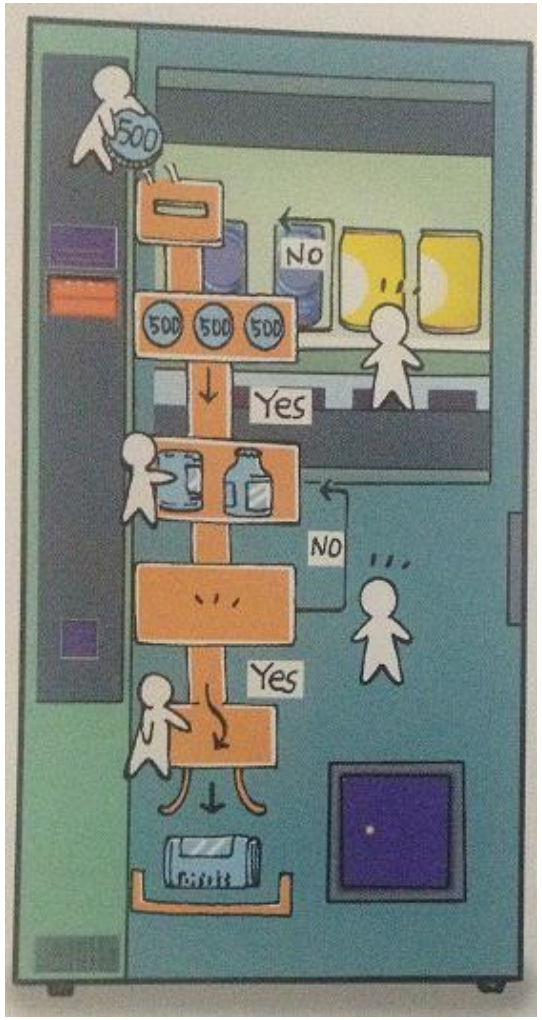
Methodologies (Programming Paradigm)

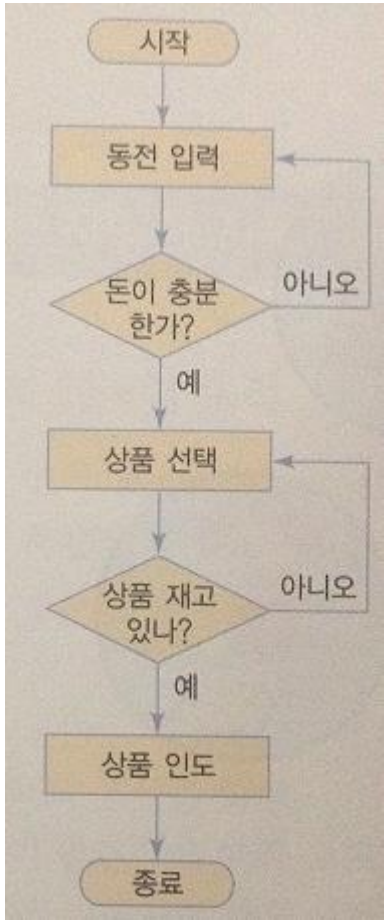
- Is a fundamental style of computer programming.



Methodologies (Programming Paradigm)

- Procedural programming

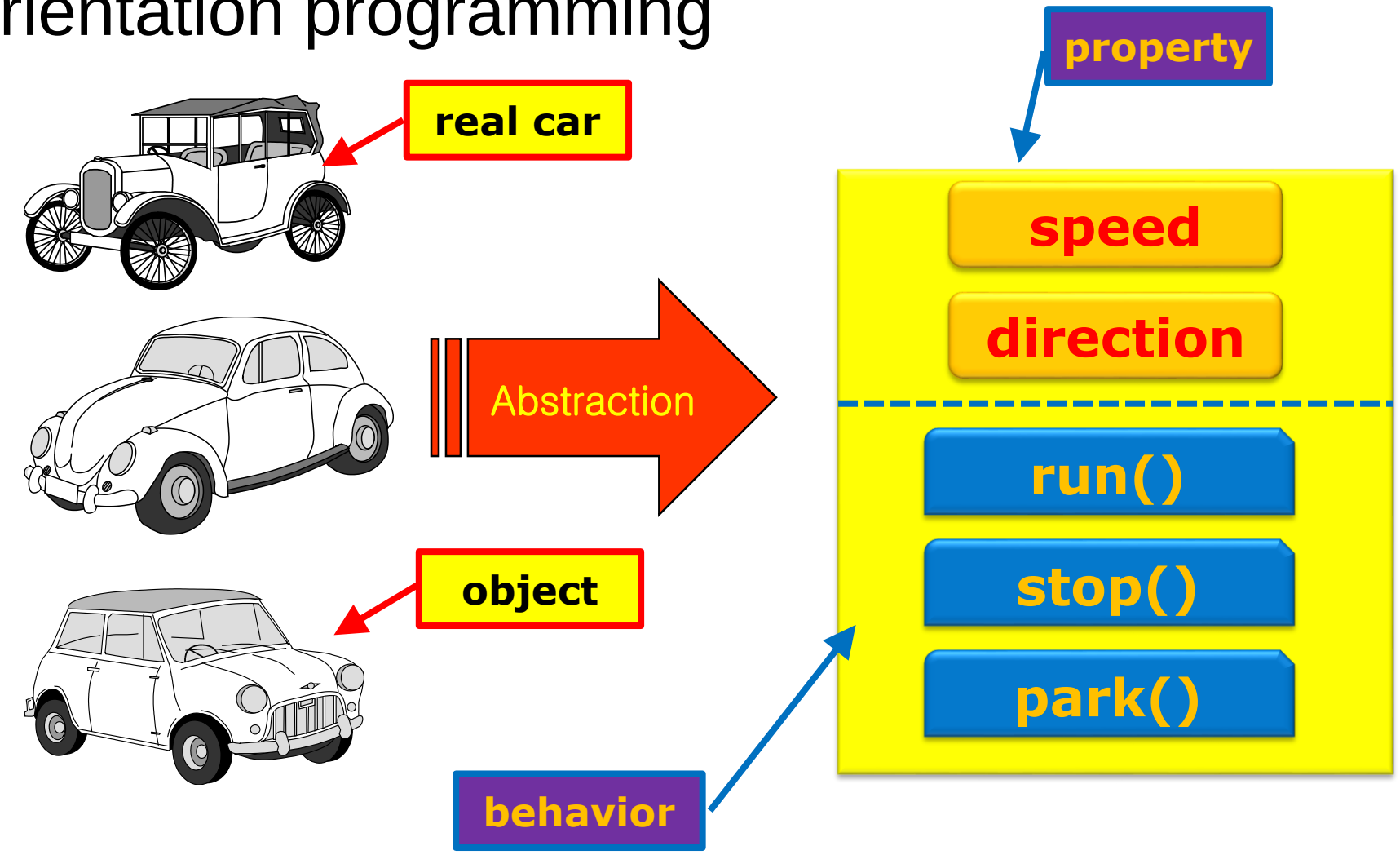




황기태/김효수, “명품 JAVA Programming”, (경기 : 생능출판사, 2011), p.166.

Methodologies (Programming Paradigm)

- Object-orientation programming



Object Orientation

- Classes : Source-code templates for objects
- Objects : Runtime instances of classes
- Members : Items put into a class to define the data content (data members) and functionality (member methods) of the objects
  - Data Members - Variables and constants which store the values that model the real-world concept the objects represent
  - Member Methods – Statements grouped into a standalone “module” for accessing or modifying data members within a class

Object Orientation (Cont.)

- Object Orientation Methodology's 3R
  - Readability
  - Reusability
  - Reliability

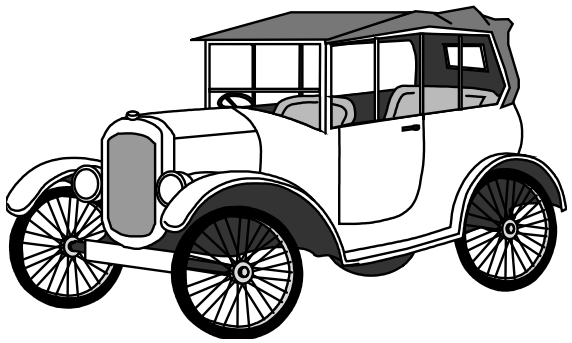
임동혁, “임동혁의 JAVA programming”, (서울 : 다올미디어, 2011), p.183.

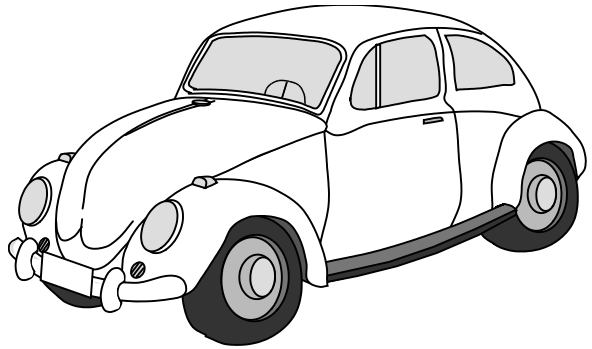
What Is a Class?

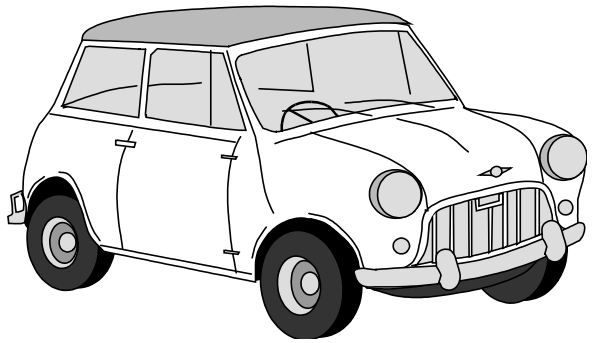
- For the philosopher…
  - An artifact of human **`class`**ification!
  - **`Class`**ify based on common behaviour or attributes
  - Agree on descriptions and names of useful **`class`**es
  - Create vocabulary; we communicate; we think!
- For the object-oriented programmer…
  - A named syntactic construct that describes common behaviour and attributes
  - A data structure that includes both data and functions

What Is an Object?

- An object is an instance of a class
- Objects exhibit:
  - Identity: Objects are distinguishable from one another
  - Behaviour(operation, function) : Objects can perform tasks
  - State(property, attribute: Objects store information

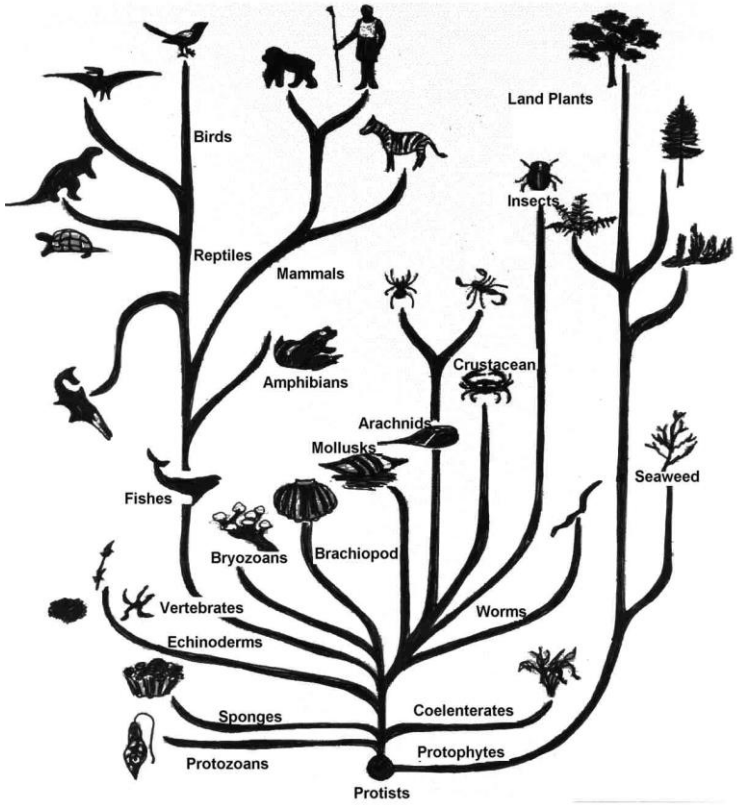






Object-Oriented Key Features

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



Abstraction

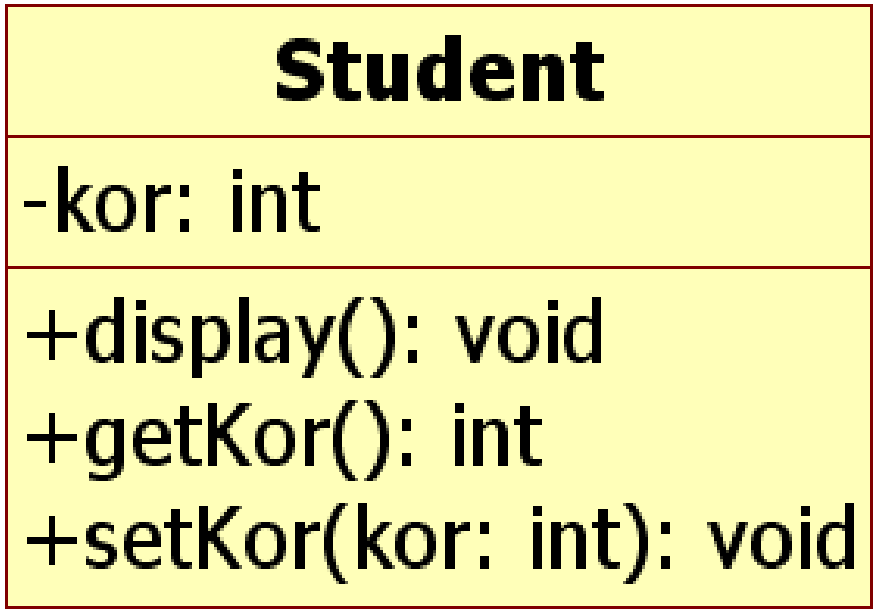
- Abstraction is selective ignorance
  - Decide what is important and what is not
  - Focus and depend on what is important
  - Ignore and do not depend on what is unimportant
  - Use encapsulation to enforce an abstraction

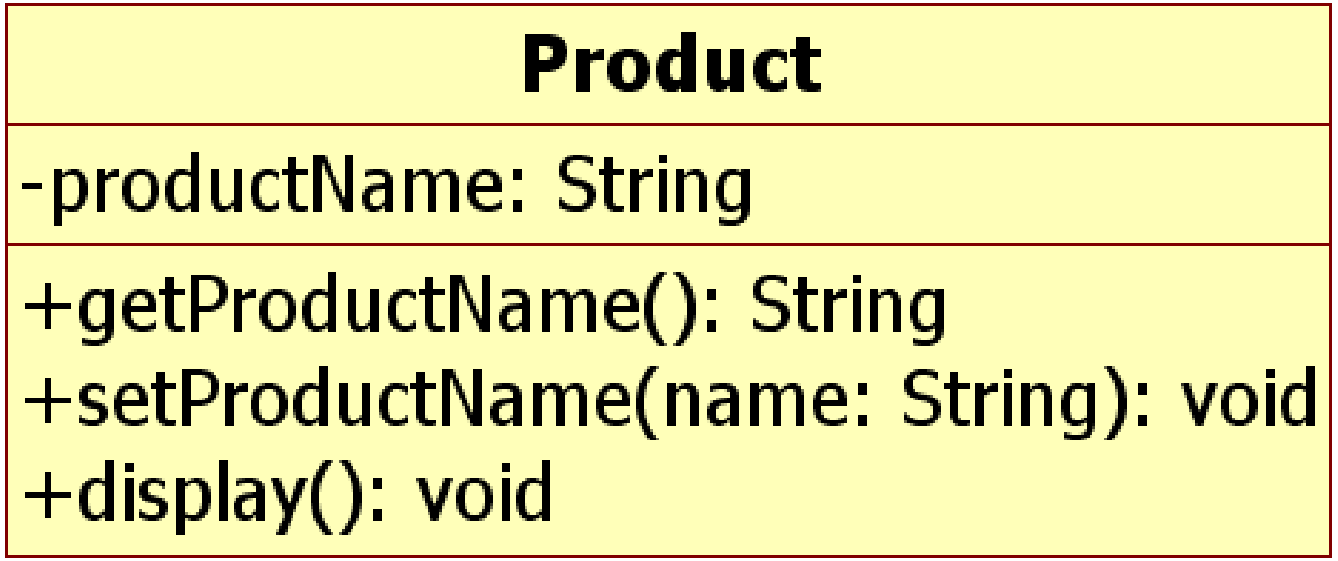
The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
Edsger Dijkstra

Encapsulation

- The principle of protecting sensitive parts of your objects from external manipulation.
- Operations and attributes are its *members*.
- Members can be **`public`** or **`private`**.
- All variables should be kept **`private`**.
- Variables are modified by methods of their *own* class.
- Hides the implementation details of a class.
- Forces the user to use an interface to access data.
- Makes the code more maintainable.

Encapsulation Examples



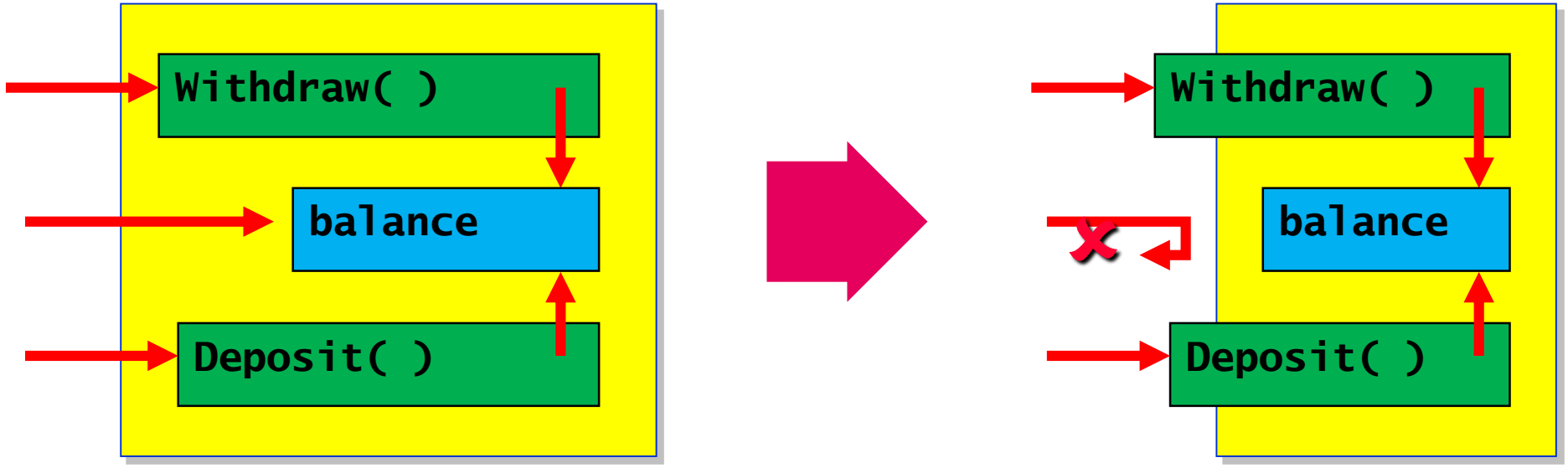


Restricting Data Access

- Use modifiers to restrict data access:
  - **`public`**
    - Everyone can use that part of the object.
  - **`private`**
    - No one outside the object's class can use that part of the object.

Implementing Encapsulation

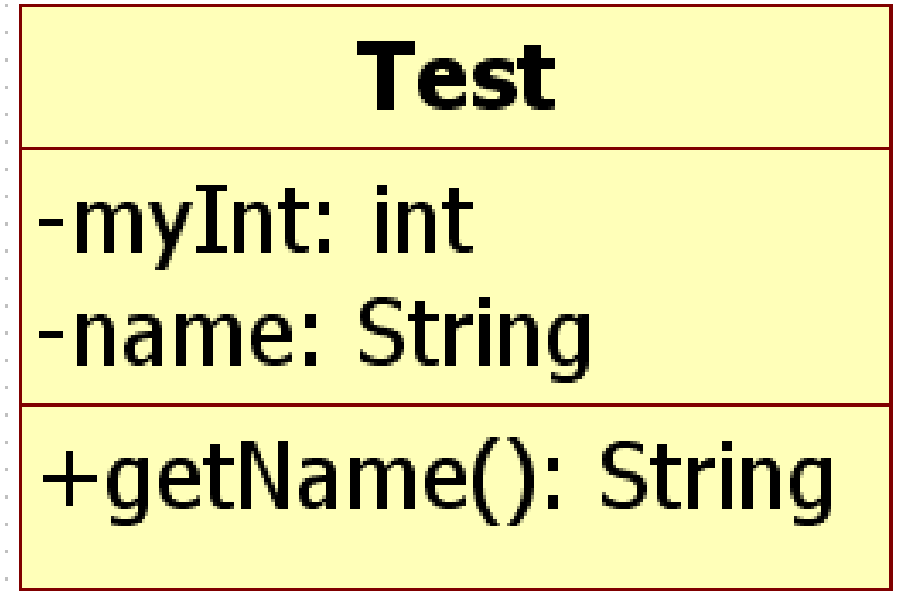
- Methods are **`public`**, accessible from the outside.
- Data is **`private`**, accessible only from the inside.



Implementing Encapsulation (Cont.)

- Put `public` or `private` in front of members

```
private int myInt;
private String name;
public String getName(){
    return name;
}
```



`get` Methods and `set` Methods

- When variables are **`private`**, they must be accessed by member methods.
- **`get`** and **`set`** methods obtain and assign values.

```
class EncapsulatedEmployee{
    private int employeeNumber;
    public void setEmployeeNumber(int newValue){
      employeeNumber = newValue;
    }
    public int getEmployeeNumber(){
      return employeeNumber;
    }
}
```

The `this` Reference

- The `this` keyword means "*reference to the same object*".

```
class Example {
    void method1() {
        this.method2();
    }
    void method2(){
     //whatever method2 does
     }
}
```

The `this` Reference (Cont.)

```
class Test {
    private int kor;
    public void setKor(int kor){
        this.kor = kor;
     }
}
```

The `this` Reference (Cont.)

```
class A {
   int a;
   B  b;
   public A() {
      b = new B ();
   }
   void a(int a){
      this.a = a;
      b.doJob(this);
   }
}
```

```
class B {
    public B() { }
    void doJob(A a) {
          System.out.println(a.a);
    }
}
```

```
public class This{
     public static void main(String [] args){
           A a = new A();
           a.a(10);
     }
}
```

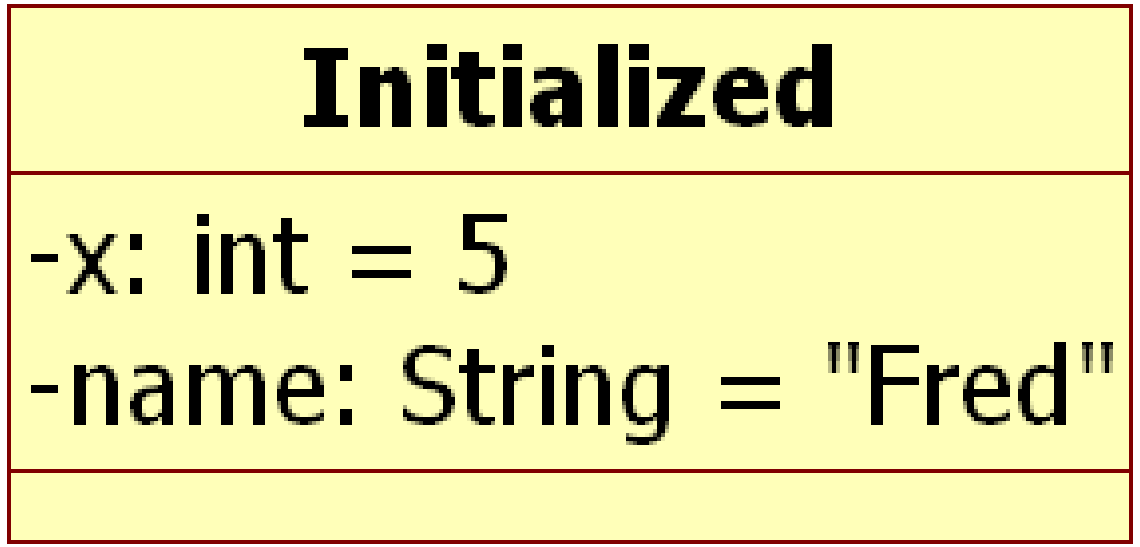
Creating Objects

- Step 1: Allocating memory
  - Use **`new`** keyword to allocate memory from the heap
- Step 2: Initializing the object by using a ***constructor***
  - Use the name of the class followed by parentheses

```
Date when = new Date( );
```

Explicit Member Initialization

```
public class Initialized {
        private int x = 5;
        private String name = “Fred”;
 }
```



Constructors

- Are special methods.
- Are called each time you create an object.
- Have no return type.
- Have the same name as the class name.
- Constructors allow you to specify values for objects when you create them.
- You have been using a ***default constructor*** throughout most of the course.

Default Constructors

- All classes must have at least one constructor.
- The compiler provides a default constructor to any class which does not have an explicit constructor.
- Features of a default constructor
  - Public accessibility
  - Same name as the class
  - No return type—not even void
  - Expects no arguments
  - Initializes all fields to zero, false or null

Default Constructors (Cont.)

- If a class has no constructor, a default constructor is inserted.
- When you use **`new`** to instantiate an object, **`new`** automatically calls the class's default constructor.
- The compiler will insert the default constructor.

Overriding the Default Constructor

- The default constructor might be inappropriate
  - If so, do not use it; write your own!

```
class Date {

    public Date( ) {
        ccyy = 1970;
        mm = 1;
        dd = 1;
    }
    private int ccyy, mm, dd;
}
```

Overloading Method Revisited

- It can be used as follows:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

- Argument lists ***must*** differ.
- Return types ***can*** be different.

Overloading Constructors

- Constructors are methods and can be overloaded
  - Same scope, same name, different parameters.
  - Allows objects to be initialized in different ways.
- WARNING
  - If you write a constructor for a class, the compiler does not create a default constructor.

```
class Date {

    public Date( ) { ... }
    public Date(int year, int month, int day) { ... }
    ...
}
```

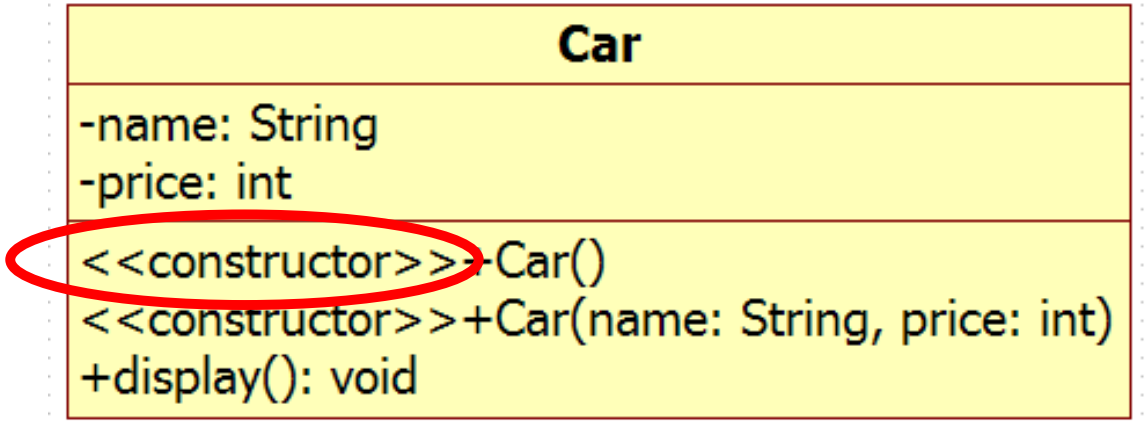
Overloading Constructors (Cont.)

- You can use the **`this`** reference at the first line of a constructor to call another constructor.

```
public class Employee {
        private String name;
        private int salary;
        public Employee (String n, int s) {
            name = n;
            salary = s;
        }
        public Employee (String n) {
            this(n,0);
        }
        public Employee () {
             this("Unknown");
        }
}
```

Overloading Constructors (Cont.)

```
2  public class Car {
3      private String name;    //member variable
4      private int price;      //member variable
5
6      public Car(){  //default constructor
7          System.out.println("Default Constructor");
8      }
9      public Car(String name, int price){   //constructor
10         System.out.println("Constructor");
11         this.name = name;
12         this.price = price;
13     }
14     public void display(){
15         System.out.printf("name = %s, price = %,d\n", this.name, this.price);
16     }
17 }
```



Instance Initialization Block

- Are called each time you create an object such ***constructor***.
- Is performed prior to ***constructor***.

```
2  public class Car {
3      private String name;    //member variable
4      private int price;      //member variable
5
6      {   //Instance Initialization Block
7          System.out.println("Instance Initialization Block");
8          this.name = "Matiz";
9          this.price = 10_000_000;
10         display();
11     }
12
13     public Car(String name, int price){   //Constructor
14         System.out.println("Constructor");
15         this.name = name;
16         this.price = price;
17     }
18     public void display(){
19         System.out.printf("name = %s, price = %,d\n", this.name, this.price);
20     }
```