The Object-Oriented Thought Process

Chapter 05

Class Design Guidelines

Contents

- Modeling Real World Systems
- Identifying Public Interfaces
- Robust Constructors
- Error Handling Design
- Designing with Reuse in Mind
- Designing with Extensibility
- Maintainability
- Object Persistence

Modeling Real World Systems

One of the primary goals of object-oriented (OO) programming is to model real-world systems in ways similar to the ways in which people actually think.

- Rather than using a structured, or *top-down, approach,* where data and behavior are logically separate entities.
- The OO approach encapsulates the data and behavior into objects that interact with each other.

Modeling Real World Systems

- Design classes in a way that represents the true behavior of the object
  - Cab class, Cabbie class : model a real-world entity
    - encapsulate their data and behavior
    - interact through each other's public interfaces

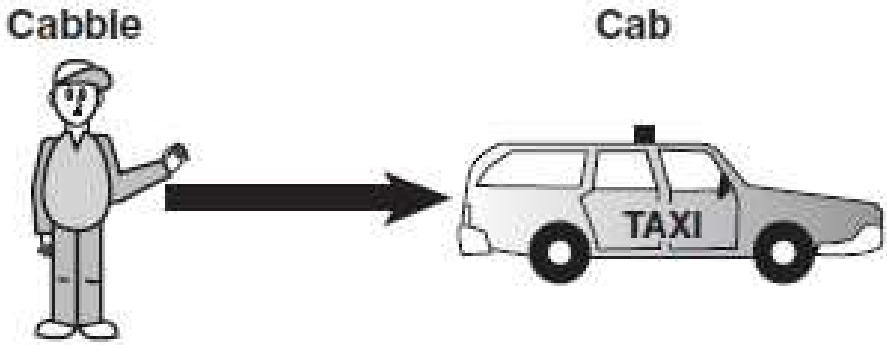


Figure 5.1 A cabbie and a cab are real-world objects.

Identifying Public Interfaces

Perhaps the most important issue when designing a class is to keep the public interface to a minimum.

- The entire purpose of building a class is to provide something useful and concise
  - "the interface of a well-designed object describes the services that the client wants accomplished."

The Minimum Public Interface(1/2)

If the public interface is not properly restricted, problems can result in the need for debugging, and even trouble with system integrity and security can surface.

- Creating a class is a business proposition, and it is very important that the users are involved with the design right from the start and throughout the testing phase.

The Minimum Public Interface(2/2)

- Example :
  - If other objects need to get the name of a cabbie,
    - Cabbie class must provide a public interface

      ➡ getName()

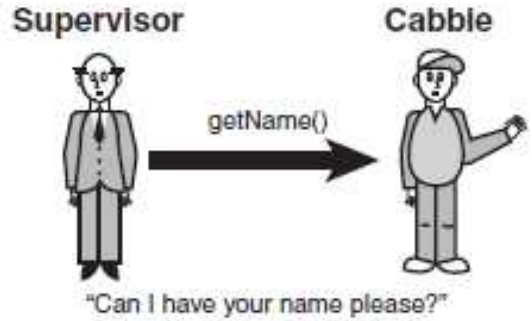


Figure 5.2 The public interface specifies how the objects interact.

  - All the users need to know,
    - how to instantiate and use the object.
    - nothing about its internal workings.

Hiding the Implementation(1/2)

The implementation should not involve the users at all.

- The implementation must provide the services that the user needs.
- But how these services are actually performed should not be made apparent to the user.

Hiding the Implementation(2/2)

- Cabbie example

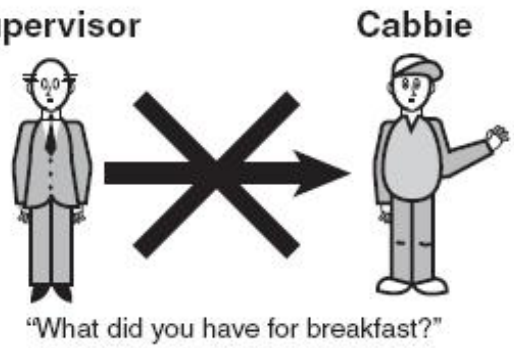


Figure 5.3 Objects don't need to know some implementation details.

- Gilbert and Mc-Carty state that
  - prime directive of encapsulation is that
    - "all fields shall be private."
    - none of the fields in a class is accessible from other objects.

Robust Constructors

A constructor should put an object into an initial, safe state.

- This includes issues such as attribute initialization and memory management.
- You also need to make sure the object is constructed properly in the default condition.
- It is normally a good idea to provide a constructor to handle this default situation.

Destructors

- Destructors include proper clean-up functions.
  - releasing system memory that the object acquired at some point.
- Java & .NET
  - reclaim memory automatically via a garbage collection mechanism.
- C++
  - include code in the destructor to properly free up the memory.
  - If ignored, a memory leak will result.

참고 : C++ Destructor 의 이해(1)

```
class AAA
{
    // empty class
};
```

```
~AAA() { . . . . }
```

AAA 클래스의 소멸자! 객체 소멸 시 자동으로 호출된다.


```
class AAA
{
public:
    AAA() { }
    ~AAA() { }
};
```

- 클래스 이름 앞에 '~' 가 붙은 형태
- 반환형이 선언되어 있지 않음
- 매개변수가 없음 - 오버로딩도 디폴트값 설정도 불가능

생성자와 마찬가지로 소멸자도 정의하지 않으면 디폴트 소멸자가 삽입된다.

참고 : C++ Destructor 의 이해(2)

```
class Person
{
private:
    char * name;
    int age;
public:
    Person(char * myname, int myage)
    {
        int len=strlen(myname)+1;
        name=new char[len];
        strcpy(name, myname);
        age=myage;
    }
    void ShowPersonInfo() const
    {
        cout<<"이름: "<<name<<endl;
        cout<<"나이: "<<age<<endl;
    }
    ~Person()
    {
        delete []name;
        cout<<"called destructor!"<<endl;
    }
};
```

```
void func1()
{
    Person p1("Lee Dong Su", 29);
    Person p2("Hyeun Bin", 30);
    p1.ShowPersonInfo();
    p2.ShowPersonInfo();
}
void main()
{
    func1();
    ....
}
```

생성자에서 할당한 메모리 공간을 소멸시키기 좋은 위치가 소멸자이다.

Designing Error Handling into a Class

The general rule is that the application should never crash.

- It is not a good idea to ignore potential errors.
- When an error is encountered, the system should either fix itself and continue, or exit gracefully without losing any data that's important to the user.

Documenting a Class

One of the most crucial aspects of a good design, whether it's a design for a class or something else, is to carefully document the process.

- Too much documentation and/or commenting can become background noise and may defeat the purpose of the documentation in the first place.
- Make the documentation and comments straightforward and to the point.

Cooperating Objects

A class will service other classes; it will request the services of other classes, or both.

- When designing a class, make sure you are aware of how other objects will interact with it.

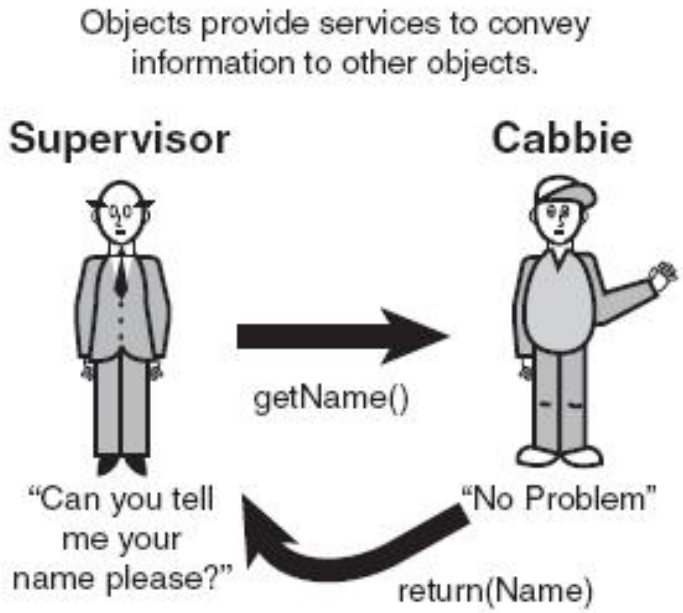


Figure 5.4 Objects should request information.

Cabbie and the Supervisor are not standalone entities; they interact with each other at various levels

Designing with Reuse in Mind

Objects can be reused in different systems, and code should be written with reuse in mind.

- This is where much of the thought is required in the design process. Attempting to predict all the possible scenarios in which an object must operate is not a trivial task.
- In fact, it is virtually impossible.
  - e.g) Attempting to figure out all the possible scenarios in which a Cabbie object must operate is not a trivial task

Designing with Extensibility in Mind

Adding new features to a class might be as simple as extending an existing class, adding a few new methods, and modifying the behavior of others.

- It is not necessary to rewrite everything.
- Consider the future use of a class when designing it.

Designing with Extensibility in Mind

- Inheritance
  - If you have just written a **Person** class, you must consider the fact that you might later want to write an **Employee** class, or a **Vendor** class
    - **Person** should contain only the data and behaviors that are specific to a person.
    - Other classes can then subclass it and inherit appropriate data and behaviors.
  - In this case, the **Person** class is said to be *extensible*

Descriptive Names

Make sure that a naming convention makes sense.

- People often go overboard and create conventions that are incomprehensible to others.
- Take care when forcing others to conform to a convention.
- Make sure that conventions are sensible.

Abstracting Nonportable Code(1/2)

If you are designing a system that must use nonportable (native) code (that is, the code will run only on a specific hardware platform), you should abstract this code out of the class.

- By abstracting out, we mean isolating the non-portable code in its own class or at least its own method (a method that can be overridden).

Abstracting Nonportable Code(2/2)

- Example
  - When writing code to access a serial port of particular hardware
    - create a wrapper class
    - send a message to the wrapper class to get services.
  - If the class moves to another hardware system,
    - code in your primary class does not have to change
    - The only place the code needs to change is in the wrapper class.

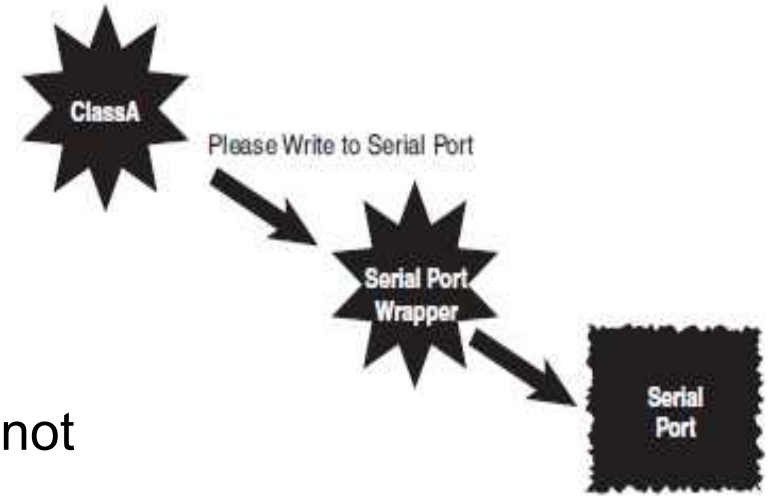


Figure 5.5 A serial port wrapper.

Copying and Comparing Objects

It is important to understand how objects are copied and compared.

- You must make sure that your class behaves as expected, and this means you have to spend some time designing how objects are copied and compared.

Minimizing Scope(1/2)

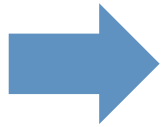
Keeping the scope as small as possible goes hand-in-hand with abstraction and hiding the implementation.

- The idea is to localize attributes and behaviors as much as possible.
- In this way, maintaining, testing, and extending a class are much easier.

Minimizing Scope(2/2)

- Example

```
public class Math {

    int temp=0;

    public int swap (int a, int b) {

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```



Localize a temporary attribute

```
public class Math {

    public int swap (int a, int b) {

        int temp=0;

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```

Class Responsibility(1/3)

Classes should be responsible for their own behavior whenever possible.

- For example, if you have several objects of various shapes, each individual shape should be responsible for drawing itself.
- This design practice localizes functionality and make it easier to add new shapes.

Class Responsibility(2/3)

- A non-OO Example

print(circle);

```
switch (shape) {
    case 1:  printCircle(circle); break;
    case 2:  printSquare(square); break;
    case 3:  printTriangle(triangle);  break;
    default: System.out.println("Invalid shape.");break;
}
```

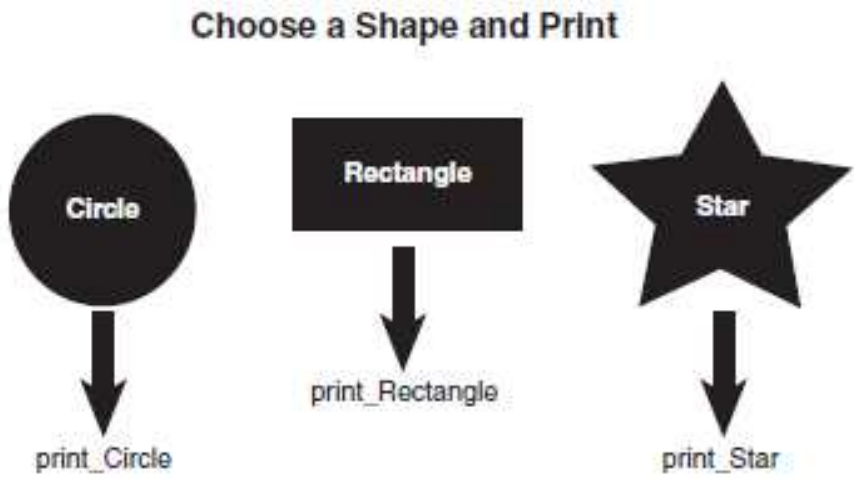


Figure 5.6 A non-OO example of a print scenario.

Class Responsibility(3/3)

- An OO Example
  - By using **polymorphism** and grouping the **Circle** into a **Shape** category, **Shape** figures out that it is a **Circle** and knows how to print itself

```
Shape.print(); // Shape is actually a Circle
Shape.print(); // Shape is actually a Square
```

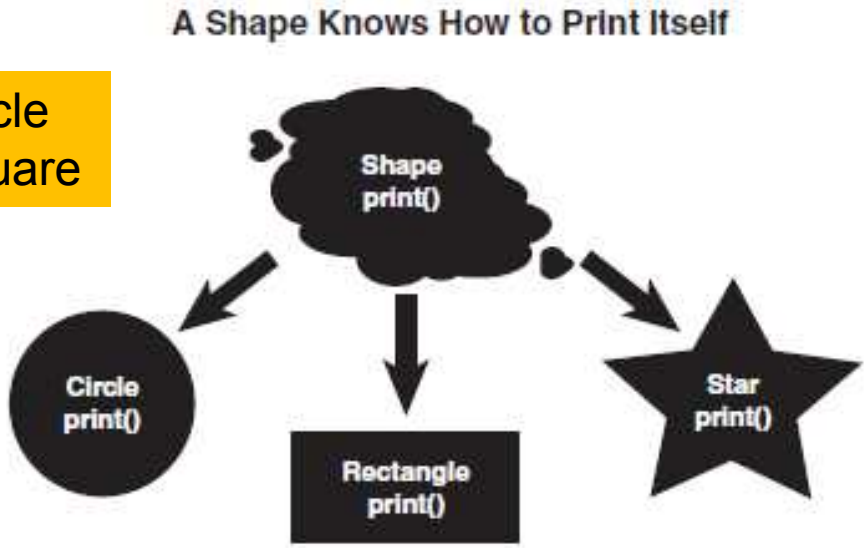


Figure 5.7 An OO example of a print scenario.

Maintainability

Designing useful and concise classes promotes a high level of maintainability.

- Just as you design a class with extensibility in mind, you should also design with future maintenance in mind.
- One of the best ways to promote maintainability is to reduce interdependent code.
  - changes in one class have no impact or minimal impact on other classes.
- If the classes are designed properly in the first place, any changes to the system should only be made to the implementation of an object.

Using Iteration

Using an iterative process is recommended.

- dovetails well into the concept of providing minimal interfaces
- A good testing plan quickly uncovers any areas where insufficient interfaces are provided.
  - Testing the design with walkthroughs and other design review techniques is very helpful
- In this way, the process can iterate until the class has the appropriate interfaces.
  - Iterate through all phases of the software life cycle.

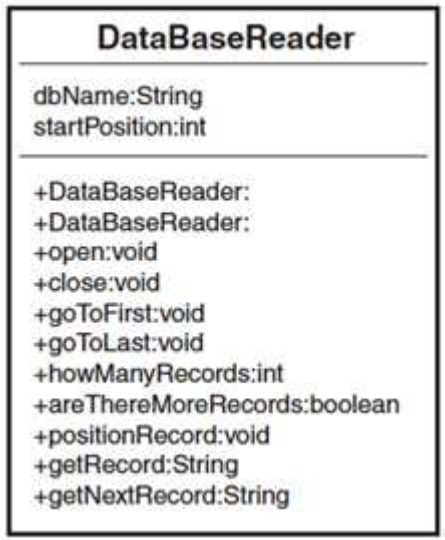
Testing the Interface(1/3)

The minimal implementations of the interface are often called *stubs.*

- By using stubs, you can test the interfaces without writing any *real code.* - from the user's perspective
- Stubs also allow testing without having the entire system in place.

Testing the Interface(2/3)

- Example



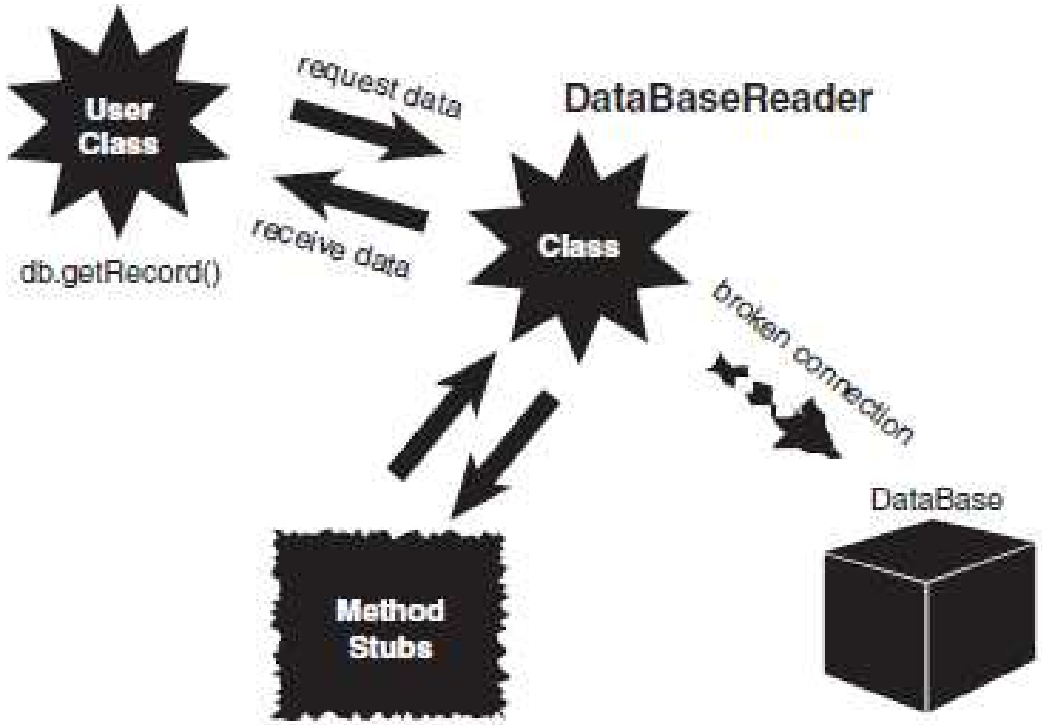


Figure 5.8 Using stubs.

Testing the Interface(3/3)

```
private String db[] = { "Record1",
    "Record2",
    "Record3",
    "Record4",
    "Record5"};

private boolean DBOpen = false;
private int pos;

public void open(String Name){
    DBOpen = true;
}
public void close(){
    DBOpen = false;
}
public void goToFirst(){
    pos = 0;
}
public void goToLast(){
    pos = 4;
}
public int howManyRecords(){
    int numOfRecords = 5;

    return numOfRecords;
}
public String getRecord(int key){

    /* DB Specific Implementation */
    return db[key];
}
public String getNextRecord(){

    /* DB Specific Implementation */
    return db[pos++];
}

}
```

Object Persistence(1/2)

Object persistence is the concept of maintaining the state of an object.

- When you run a program, if you don't save the object in some manner, the object dies, never to be recovered.
- In most business systems, the state of the object must be saved for later use.

Object Persistence(2/2)

Primary storage devices for object persistence

- Flat file system
  - store an object in a flat file by serializing the object. This has very limited use.
- Relational database
  - Some sort of middleware is necessary to convert an object to a relational model.
- OO database
  - logical way to make objects persistent
  - Most companies have all their data in legacy systems and are just starting to explore object databases.

Serializing and Marshalling Objects

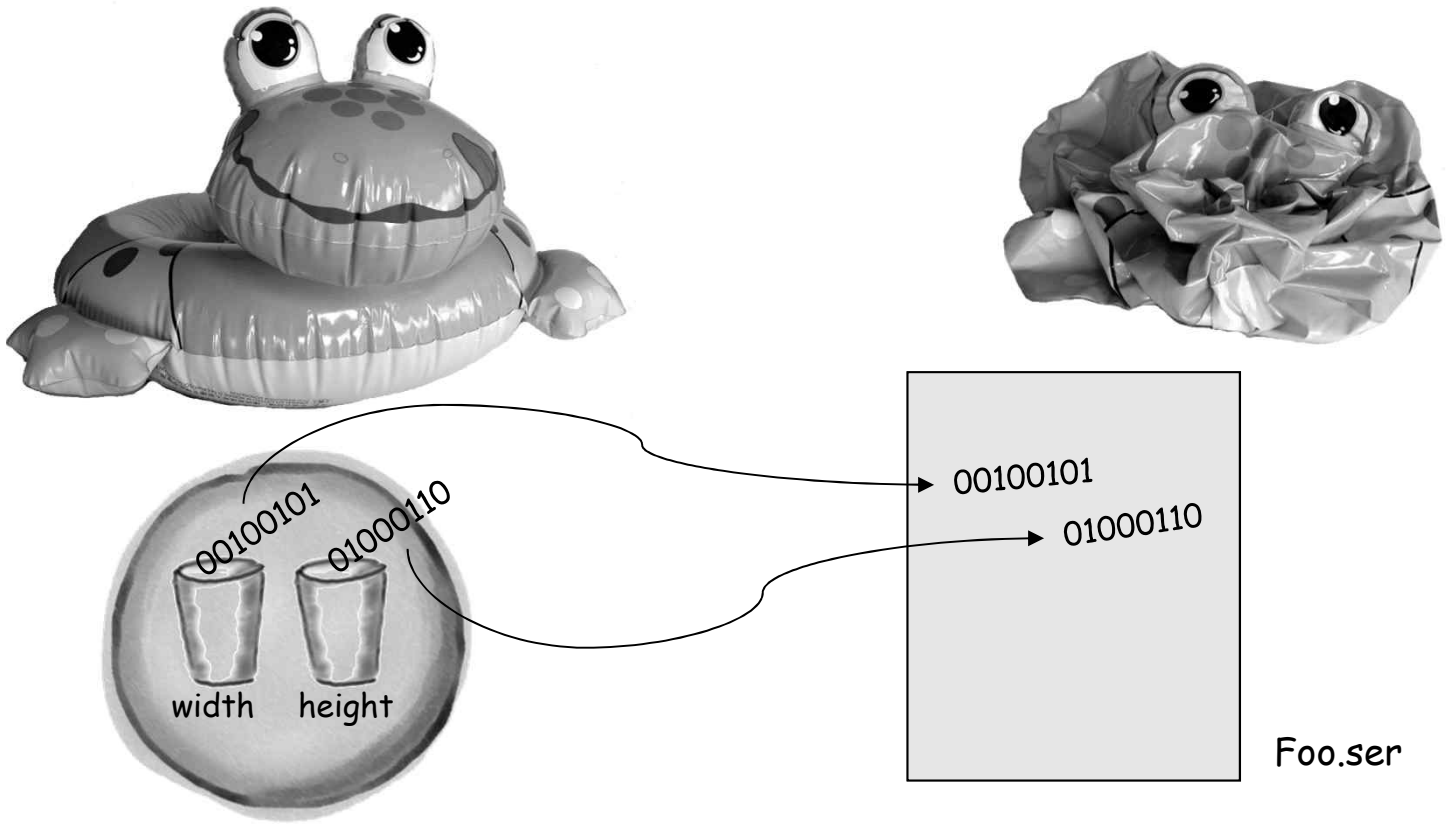
To send an object over a wire (for example, to a file, over a network), the system must deconstruct the object (flatten it out), send it over the wire, and then reconstruct it on the other end of the wire.

- This process is called *serializing an object.*
- *ISerializable* interface in C# .Net and Visual Basic .NET - allows an object to control its own serialization and deserialization
- *The act of sending the object across a* wire is called *marshaling an object.*

[참고] Object Serialization

힙 안에 들어있는 객체

직렬화된 객체




```
Foo myFoo = new Foo();
myFoo.setWidth(37);
myFoo.setHeight(70);
```

```
FileOutputStream fs = new FileOutputStream("Foo.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(myFoo);
```

JAVA ObjectInputStream & ObjectOutputStream

- 직렬화를 통해 객체의 입출력을 지원하는 클래스
- 직렬화(serialization):
  - 객체가 가진 데이터들을 순차적인 데이터로 변환하는 것
  - 대부분의 표준적인 클래스는 이미 직렬화를 지원
  - 어떤 클래스가 직렬화를 지원하려면 Serializable 인터페이스 구현 필요
- 역직렬화(deserialization)
  - 직렬화된 데이터를 읽어서 자신의 상태를 복구하는 것

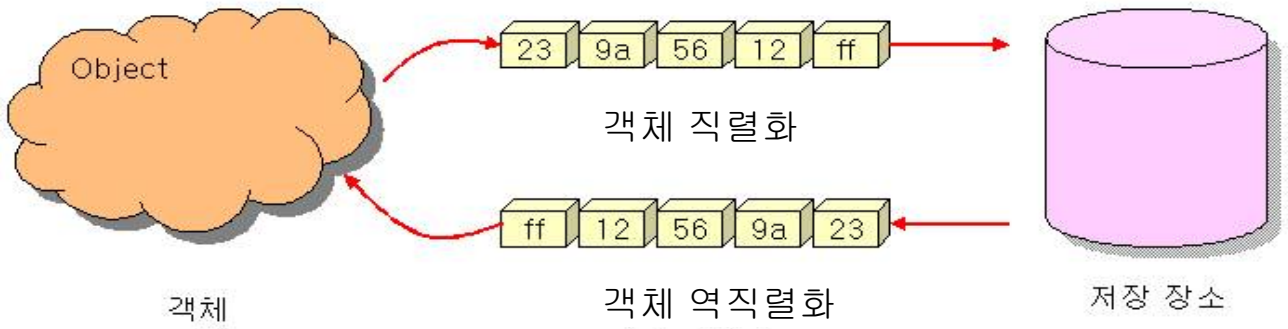
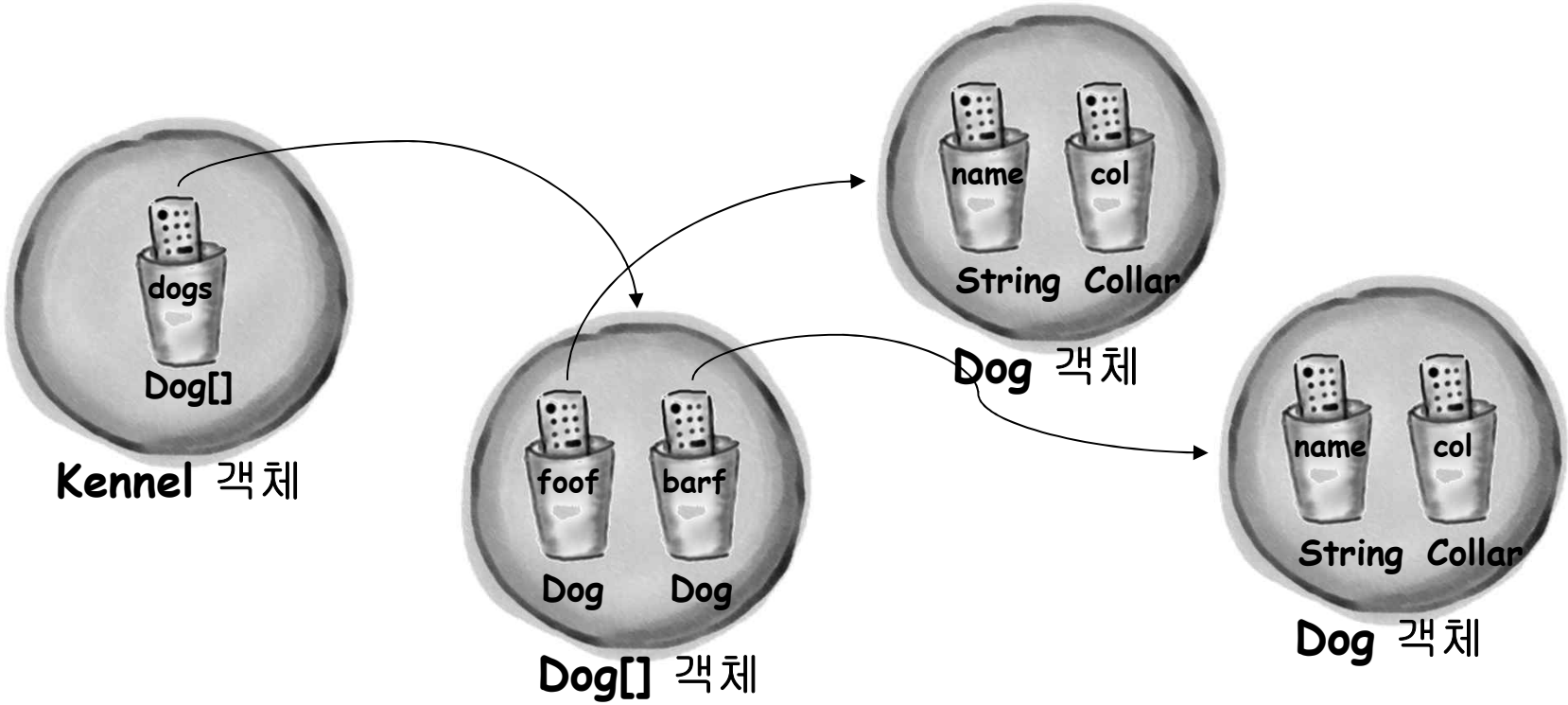


그림 24-9 객체 직렬화의 개념

연결된 객체의 직렬화

- 객체를 직렬화할 때는 그 객체와 관련된 모든 것이 저장됨
- 인스턴스 변수로 참조된 모든 객체가 줄줄이 엮여서 저장됨



클래스객체를 직렬화 가능하게 하려면?

- 클래스를 직렬화할 수 있도록 하고 싶다면 Serializable 인터페이스를 구현해야 함 - 구현해야 하는 메소드는 없음
- 어떤 객체를 저장할 때 그 객체와 관련된 것들이 모두 제대로 직렬화되지 않으면 그 직렬화 작업은 제대로 완료되지 않음.
- 직렬화하고자 하는 모든 객체가 직렬화할 수 있는(즉 Serializable을 구현하는) 클래스에 속해야 함.
- 직렬화가 제대로 되지 않는 경우에는NotSerializableException 예외가 발생.

```
objectOutputStream.writeObject(myBox);
```

```
import java.io.*;

public class Box implements Serializable {
…
}
```

예제

```
import java.io.*;
import java.util.Date;
public class ObjectStreamTest {
    public static void main(String[] args) throws IOException {
        ObjectInputStream in = null;
        ObjectOutputStream out = null;
        try {
            int c;
            out = new ObjectOutputStream(new FileOutputStream("object.dat"));
            out.writeObject(new Date());

            out.flush();
            in = new ObjectInputStream(new FileInputStream("object.dat"));
            Date d = (Date) in.readObject();
            System.out.println(d);
```

객체를 직렬화하여서 쓴다.

예제

```
        } catch (ClassNotFoundException e) {

    } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

```
Fri May 01 15:46:56 KST 2009
```