Lab 06
Decorator Pattern and Java I/O

NetDB
CS, NTHU,
Fall, 2013

Outline

- Decorator Pattern
- Java I/O
- Today's Mission

Outline

- Decorator Pattern
- Java I/O
- Today's Mission

Design Pattern

- Good practice from the experts
- Good solution for common problems
- “Design Vocabulary” between developers

Decorator Pattern

Starbuzz Coffee

- an example from O'Reilly "Head First Design Pattern"

- Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around

- They want an ordering system to spread their coffee all over the world

Menu

綠茶

紅茶

烏龍茶

Menu

Menu

$20

+ $10

$20

+ $5

+ $5

$25

+ $5

「請幫我們設計一個訂單系統。」

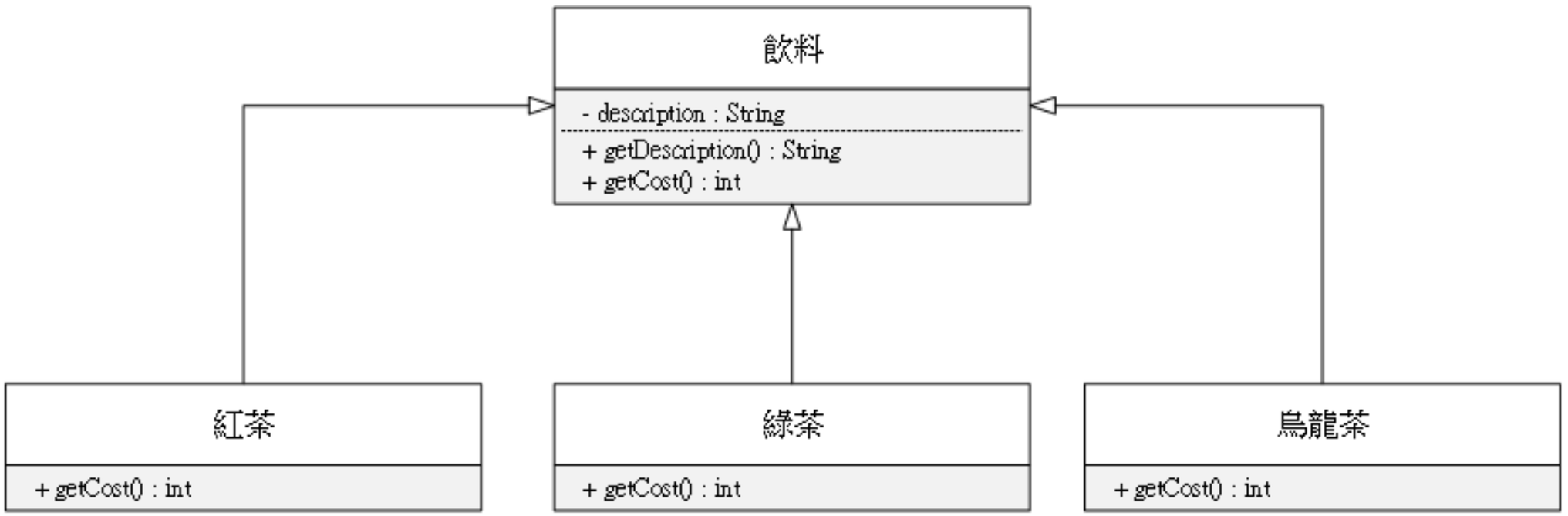
–Starbuzz Coffee

Requirement

- Any kind of beverage can add 4 kinds of condiment

- A method getCost() will return the total price of this beverage

Order System v.1



「你好，我想要一杯布丁紅茶。」

Order System v.1

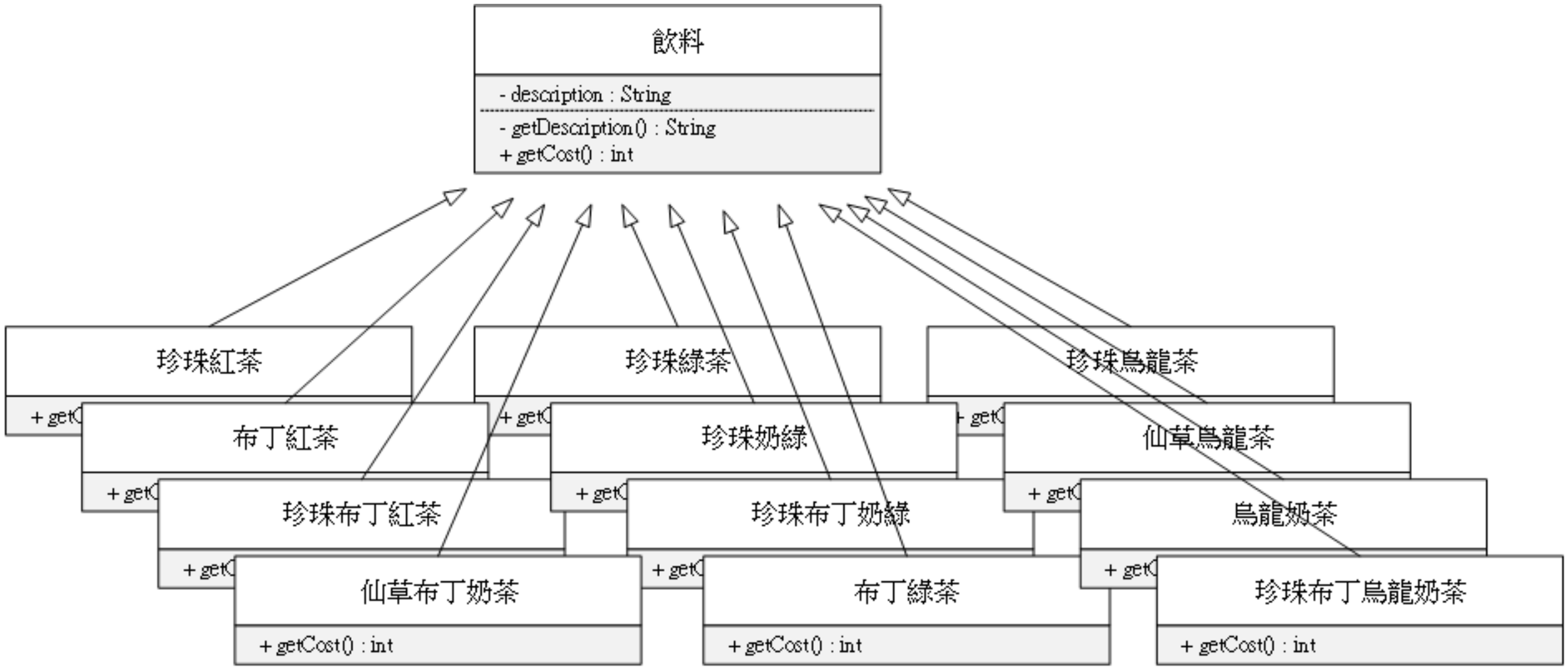
```
public class 布丁紅茶{
   public int getCost(){
      return 30;
   }
}
```

「對了，我要加珍珠。」

Order System v.1

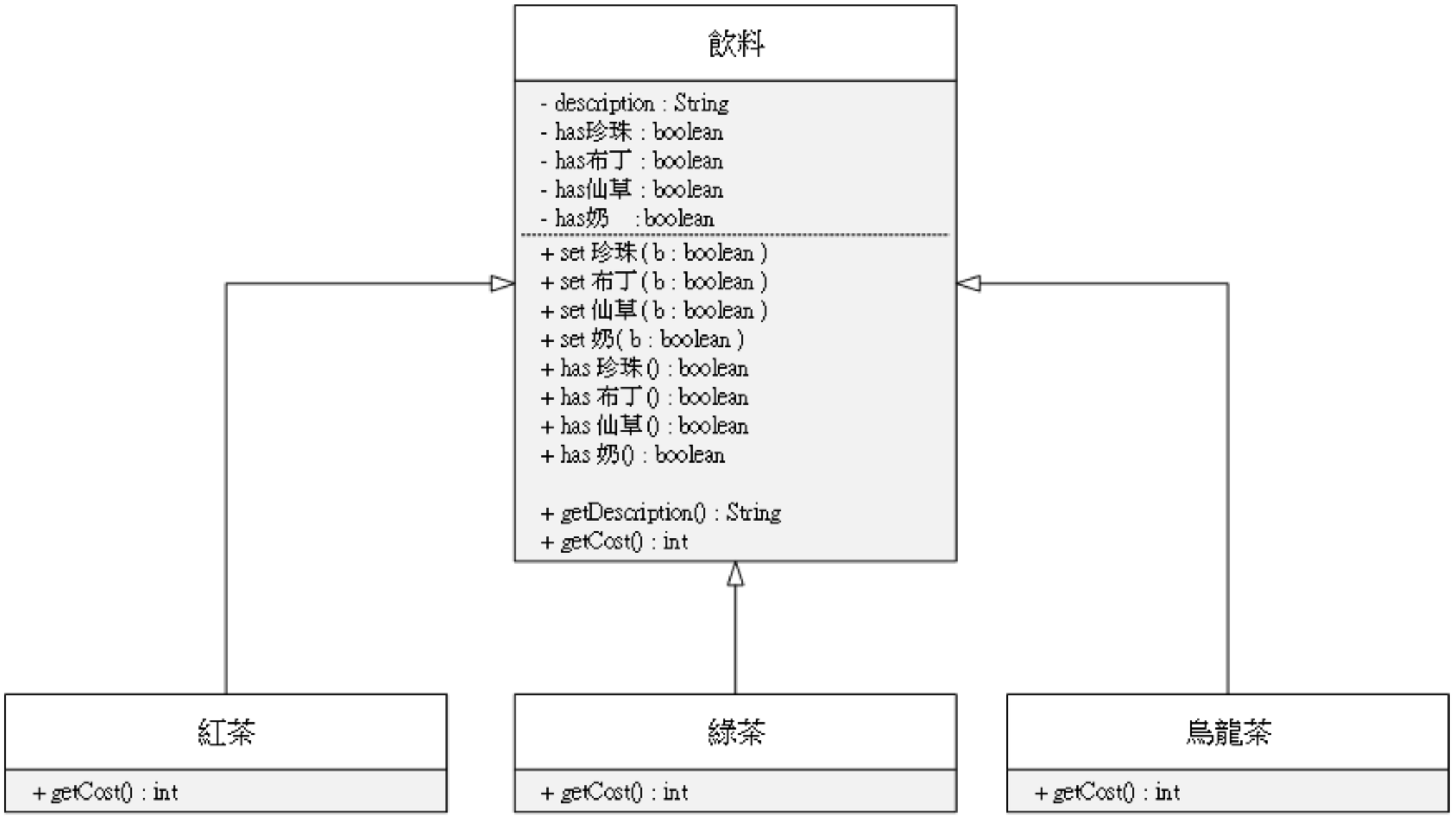
```
public class 布丁珍珠紅茶{
   public int getCost(){
      return 35;
   }
}
```

Order System v.1



Can't we just use instance variables and inheritance in the superclass to keep track of the condiments?

Order System v.2



Beverage Class

```
getCost(){
    if (有珍珠)
        price += 5;
    if (有仙草)
        price += 5;
    if (有布丁)
        price += 5;
    ……
    ……
}
```

紅茶 Class

```
getCost(){
    price = super.getCost();
    price += 20;
}
```

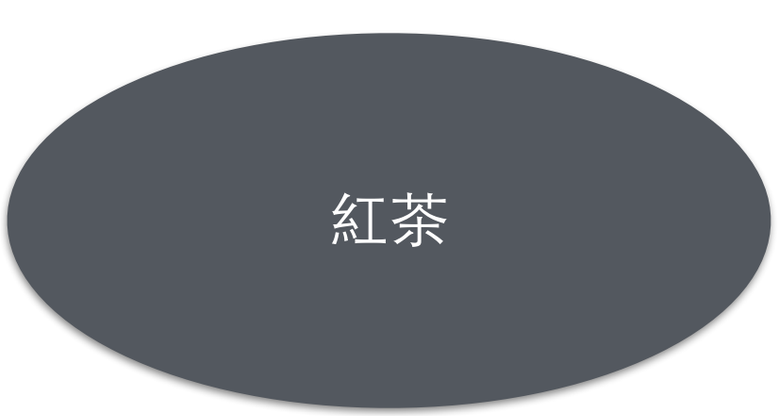
What if we have another condiment, for example, 蒟蒻？

You will have to modify your super class for any new condiment coming.

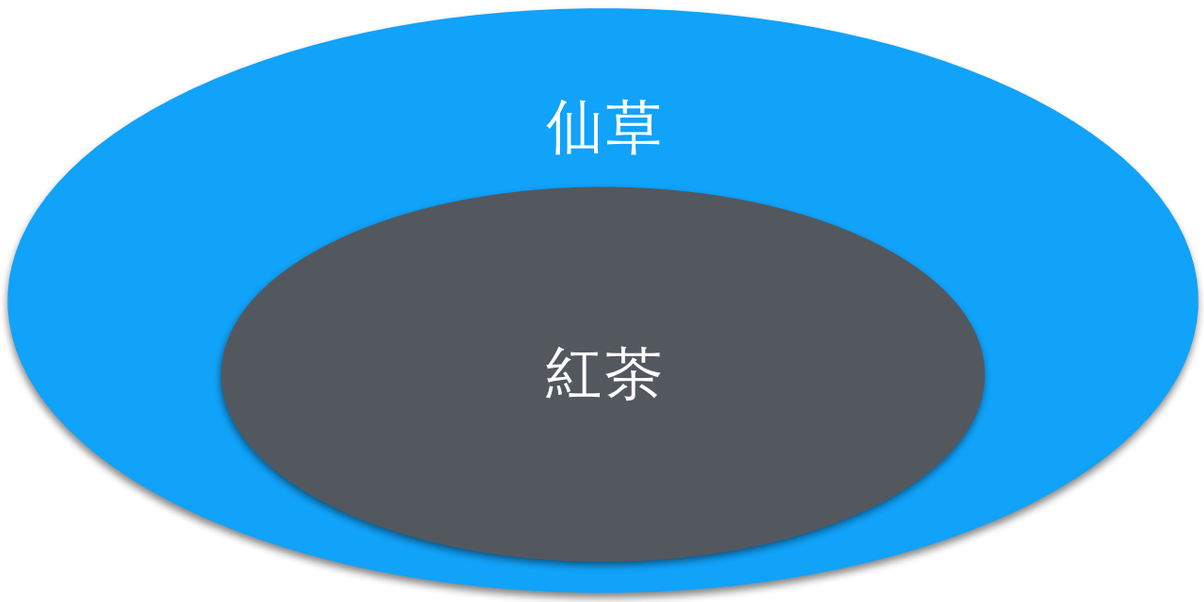
Design Principle

“Classes should be open for extension,
but closed for modification”

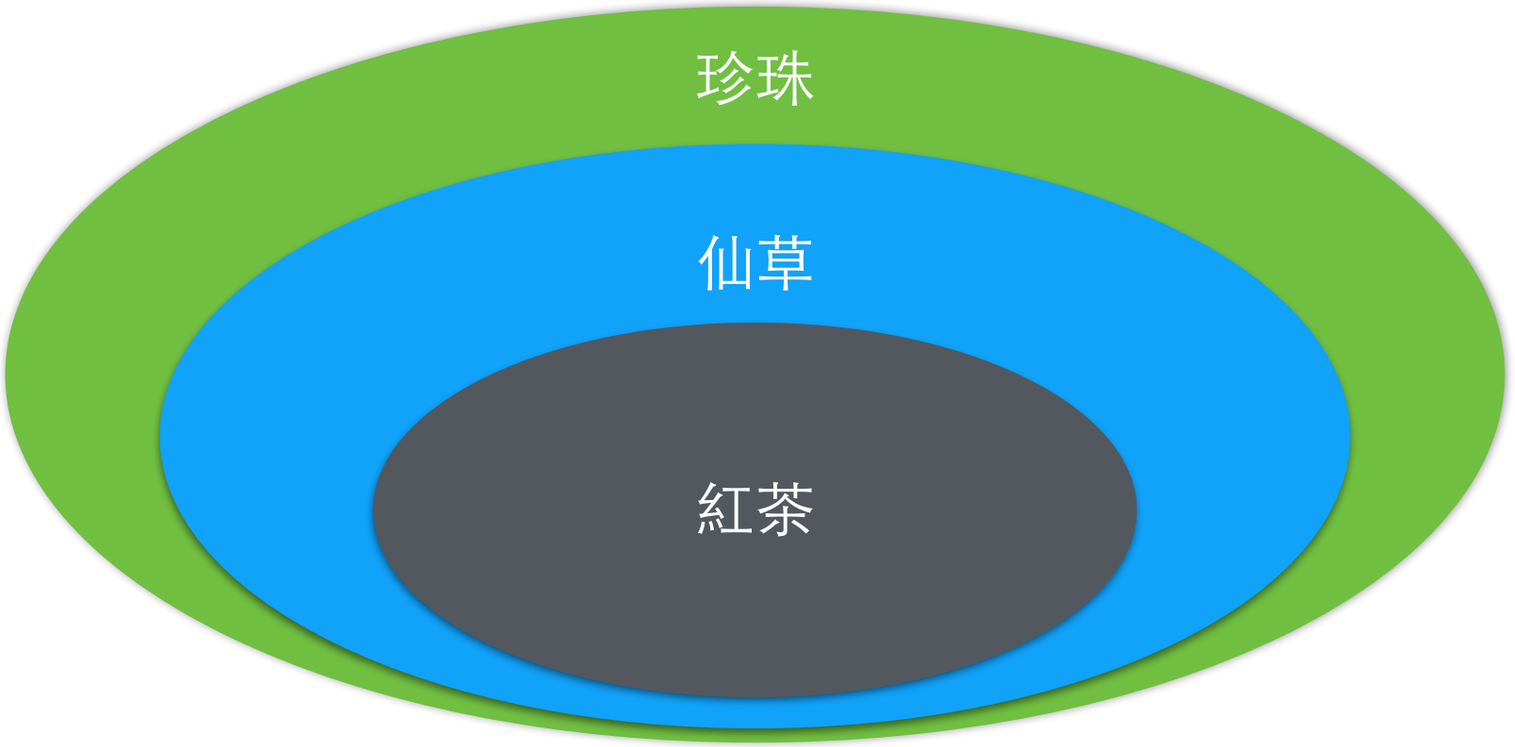
It’s All About Decoration



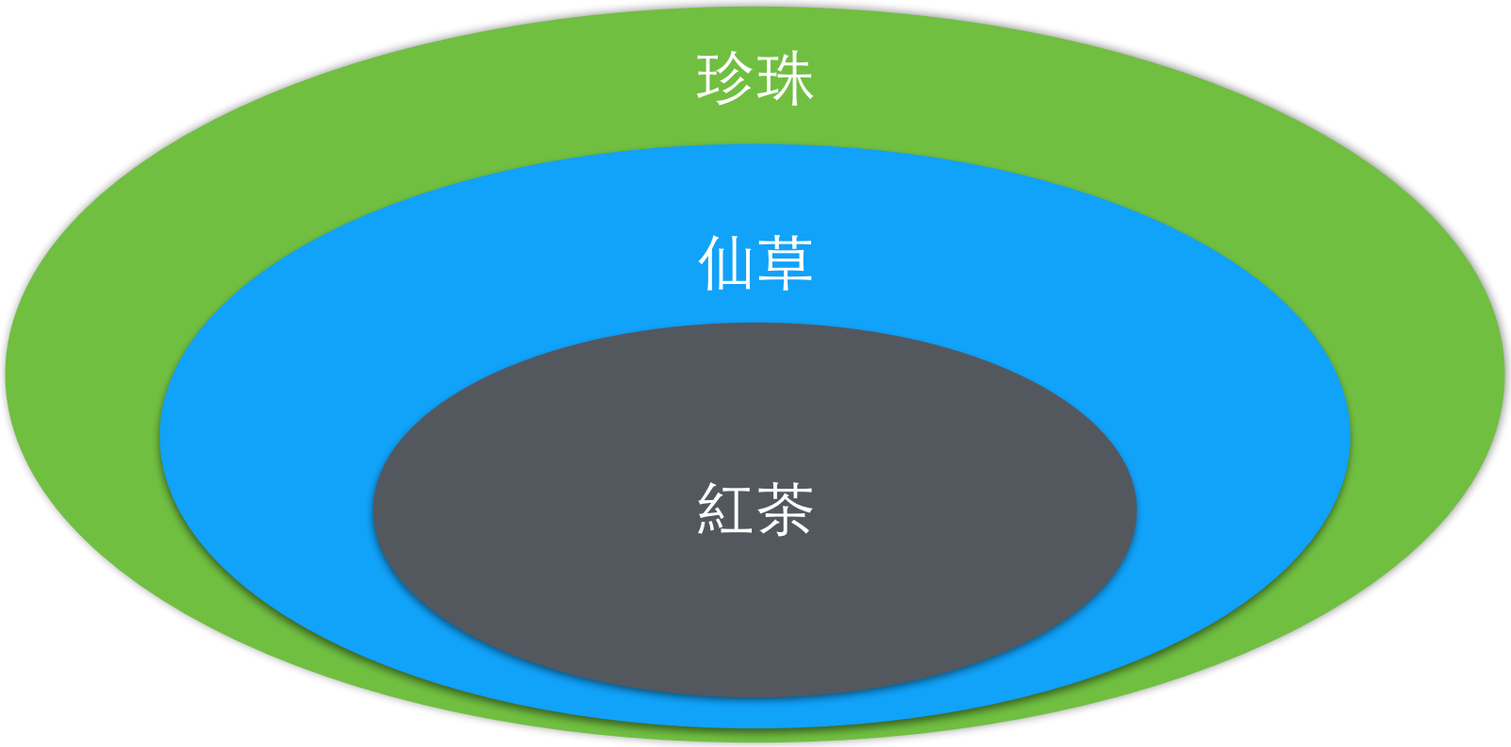
It’s All About Decoration



It’s All About Decoration

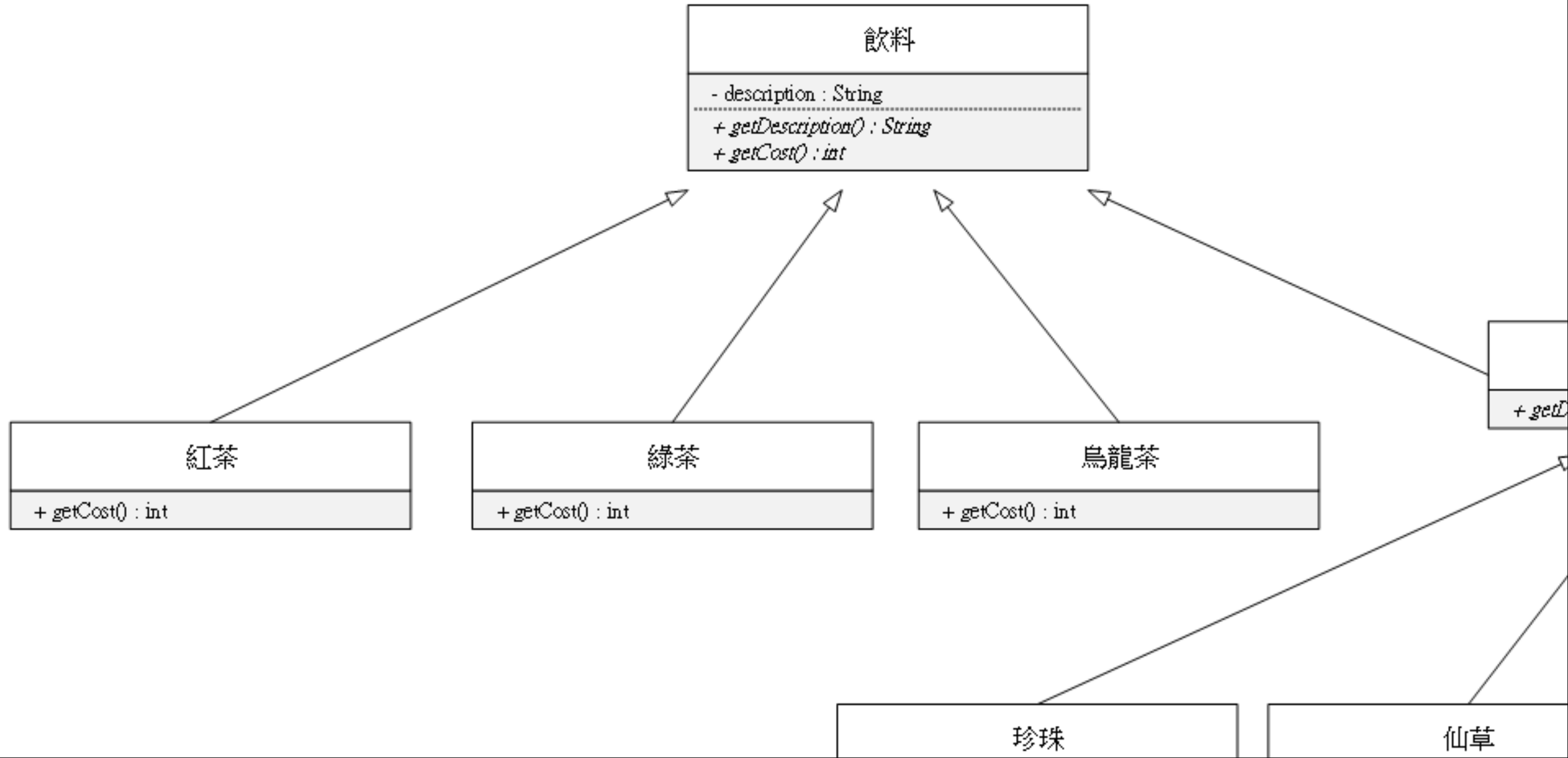


It’s All About Decoration

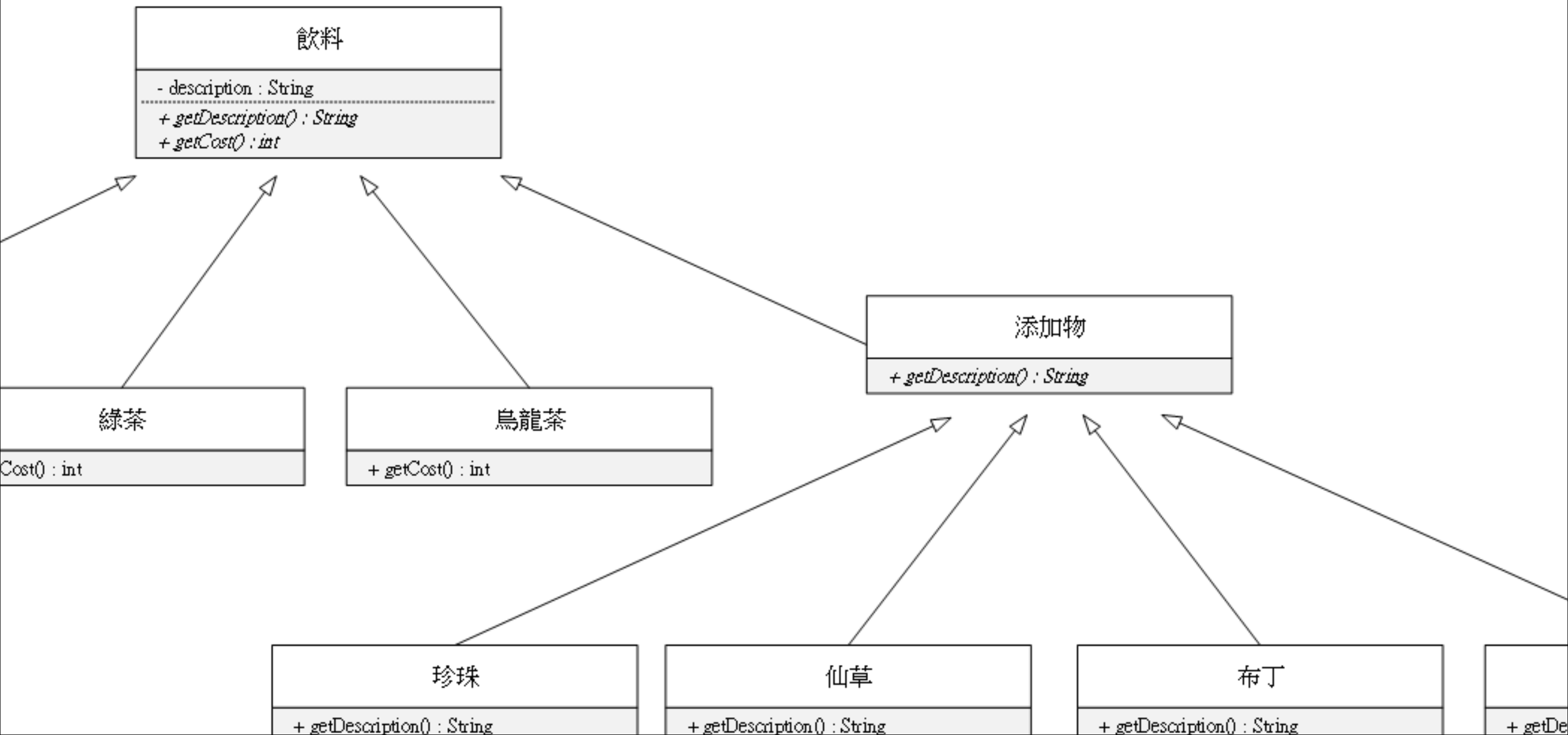


加了珍珠與仙草的飲料，還是飲料!

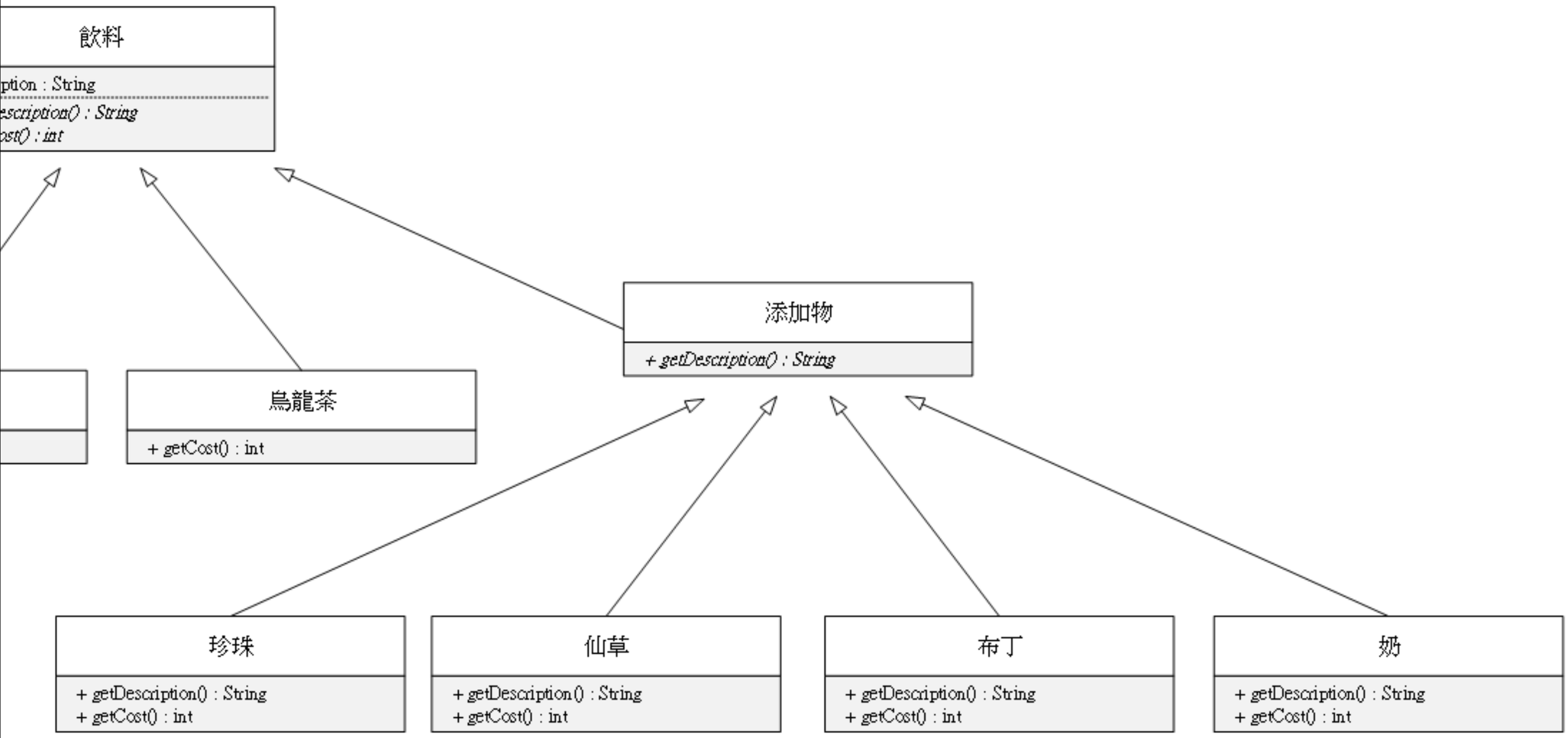
Order System v.3



Order System v.3



Order System v.3



```
public abstract class Beverage{
   description
   getDescription()
   getCost()
}

public class 紅茶 extends Beverage{
   getDescription(){
      return “紅茶”
   }
   getCost(){
      return 20;
   }
}
```

```
public abstract class 添加物 extends Beverage{
   getDescription()
   getCost()
}
```

```
public class 仙草 extends 添加物{
   Beverage b;

   public 仙草(Beverage b){
      this.b = b;
   }

   getDescription(){
      return “仙草” + b.getDescription()
   }

   getCost(){
      return b.getCost() + 5;
   }
}
```

I Want a 珍珠仙草紅茶

```
Beverage b =
    new 紅茶();
```

I Want a 珍珠仙草紅茶

```
Beverage b =
    new 仙草(new 紅茶());
```

I Want a 珍珠仙草紅茶

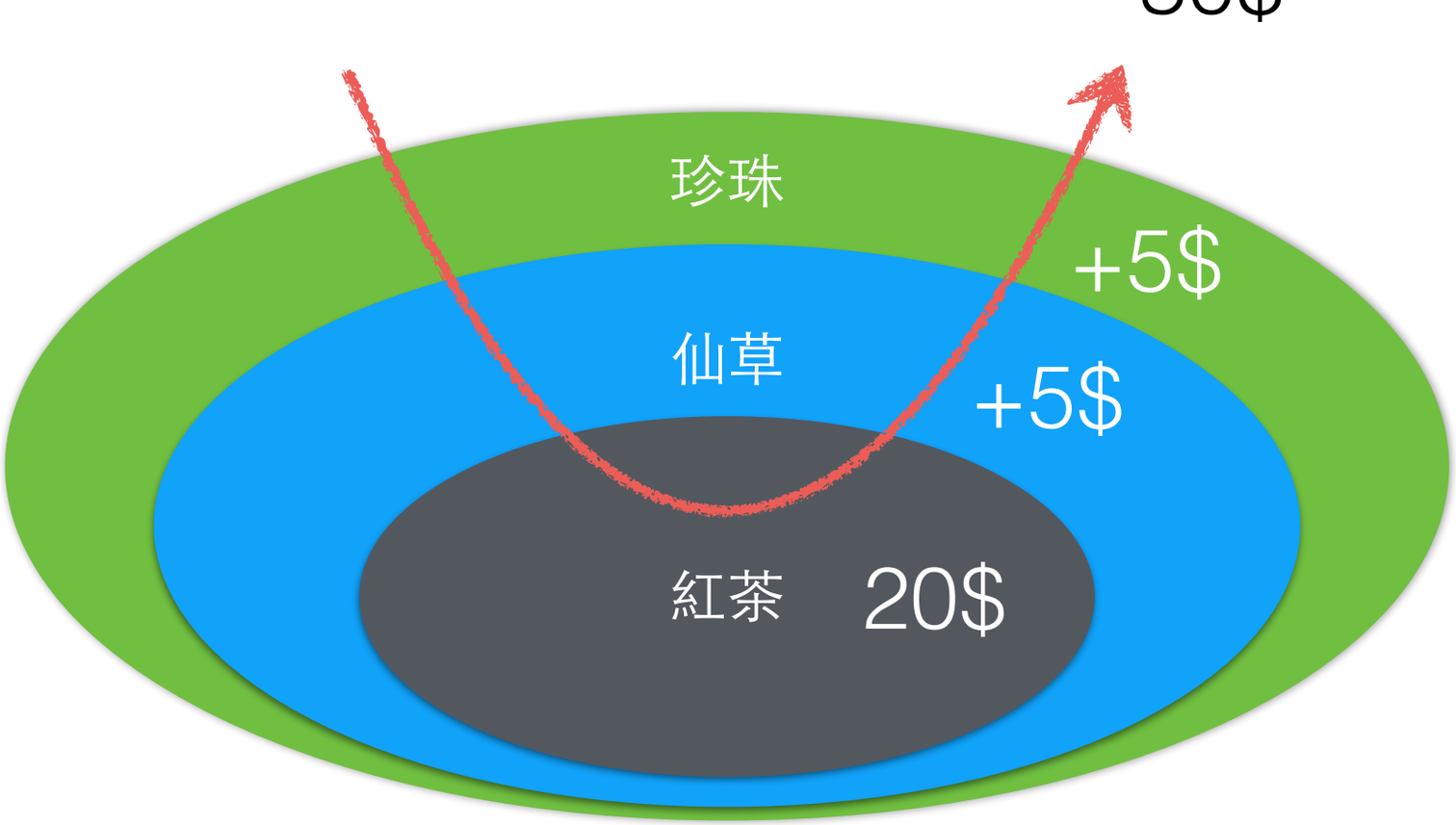
```
Beverage b =
    new 珍珠((new 仙草(new 紅茶())));
```

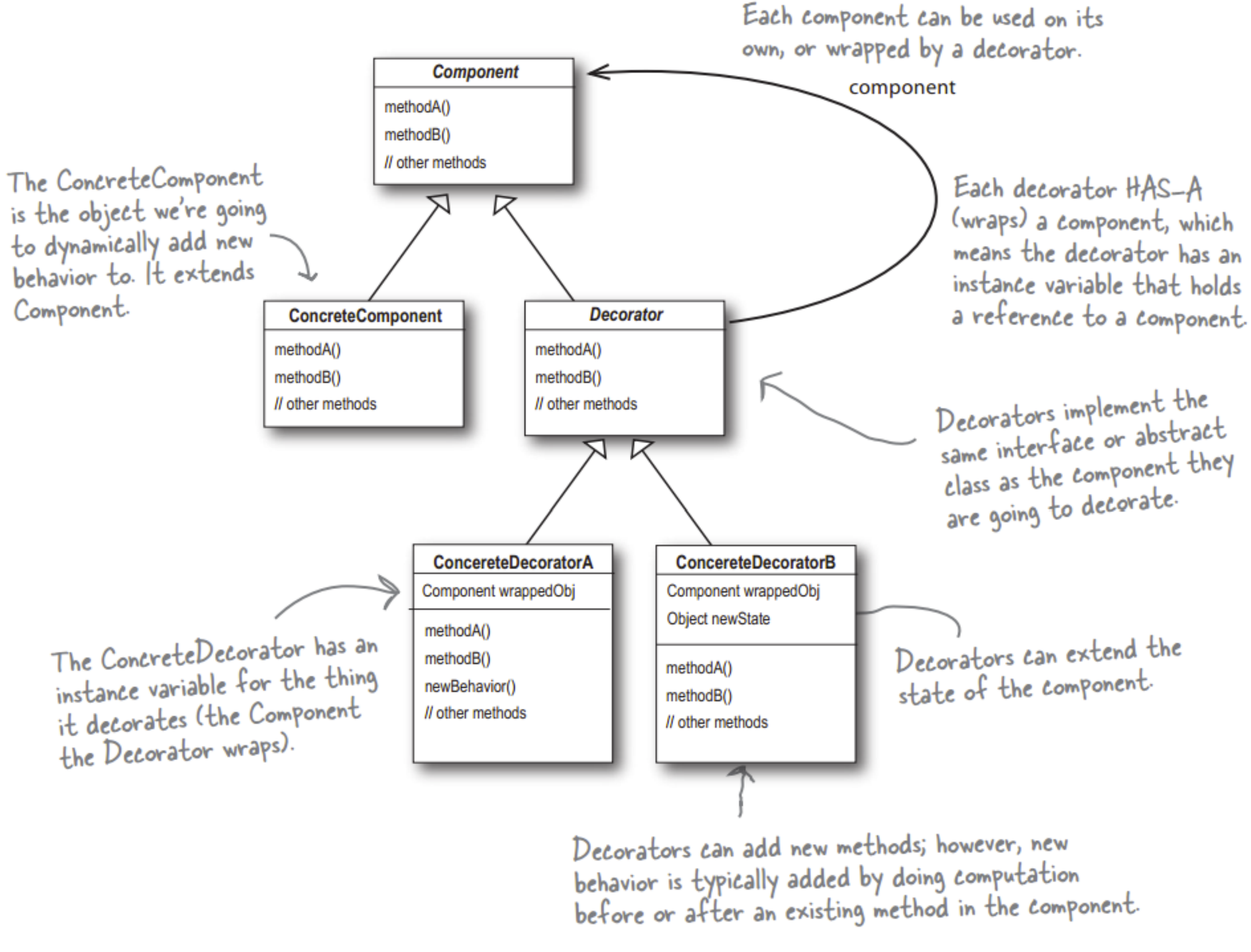
I Want a 珍珠仙草紅茶

```
Beverage b =
    new 珍珠((new 仙草(new 紅茶())));

b.getCost();
```

I Want a 珍珠仙草紅茶

```
Beverage b =
    new 珍珠((new 仙草(new 紅茶())));

b.getCost();
```





This chapter is free sample. You can get it here.

Outline

- Decorator Pattern
- Java I/O
- Today's Mission

Outline

- Decorator Pattern
- Java I/O
  - Paths and Files
  - Stream
  - New I/O
- Today's Mission

Java I/O

- I/O is one of the most common tasks for every programmer

Main Packages

- java.io: traditional API that provides high-level abstractions for paths, random access files, and streams

- java.nio: complement java.io with low- level abstractions and asynchronous access

Paths

- java.io.File provides a system-independent view of hierarchical paths
- Represents either files or directories:
  - File parent = new File("/bin"); // a directory
  - File child = new File(parent, "java.exe"); // a file

Random Access Files

- java.io.RandomAccessFile provides an abstraction to random accessible files (usually on local disks)

```
File path = ...
RandomAccessFile rf = new
RandomAccessFile(path, "rws");
rf.seek(10); // set pointer at which next
read/write occurs
int b = rf.read(); // pointer is advanced
automatically
rf.seek(10);
rf.write(++b);
```

Random Access Files

- Mode: "r", "rw", "rws"
  - 's' ensures that every write is synchronized to disk
- Acts like a large array of bytes
  - Has a file pointer that can be read and moved: getFilePointer(), seek()

RandomAccessCopier

```
public void copy(File src, File dest) throws IOException {
    RandomAccessFile srcRa = null, destRa = null;
    try {
        srcRa = new RandomAccessFile(src, "r");
        destRa = new RandomAccessFile(dest, "rws");
        int b = -1;
        while ((b = srcRa.read()) != -1) {
            destRa.write(b);
        }
    } finally {
        if (srcRa != null) srcRa.close();
        if (destRa != null) destRa.close();
    }
}
```

Performance Issue

- destRa (with mode "rws") ensures that each call to write() is reflected to disk
- However, disks are very slow
- The performance of RandomAccessCopier is poor
- Can you make it faster?

BufferedRandomAccessCopier

```
public void copy(File src, File dest) throws IOException {
    RandomAccessFile srcRa = null, destRa = null;
    byte[] buffer = new byte[BUFFER_SIZE];
    try {
        srcRa = new RandomAccessFile(src, "r");
        destRa = new RandomAccessFile(dest, "rws"); int num = -1;
        while ((num = srcRa.read(buffer)) != -1) {
            destRa.write(buffer, 0, num);
        }
    } finally {
        if (srcRa != null) srcRa.close();
        if (destRa != null) destRa.close();
    }
}
```

Outline

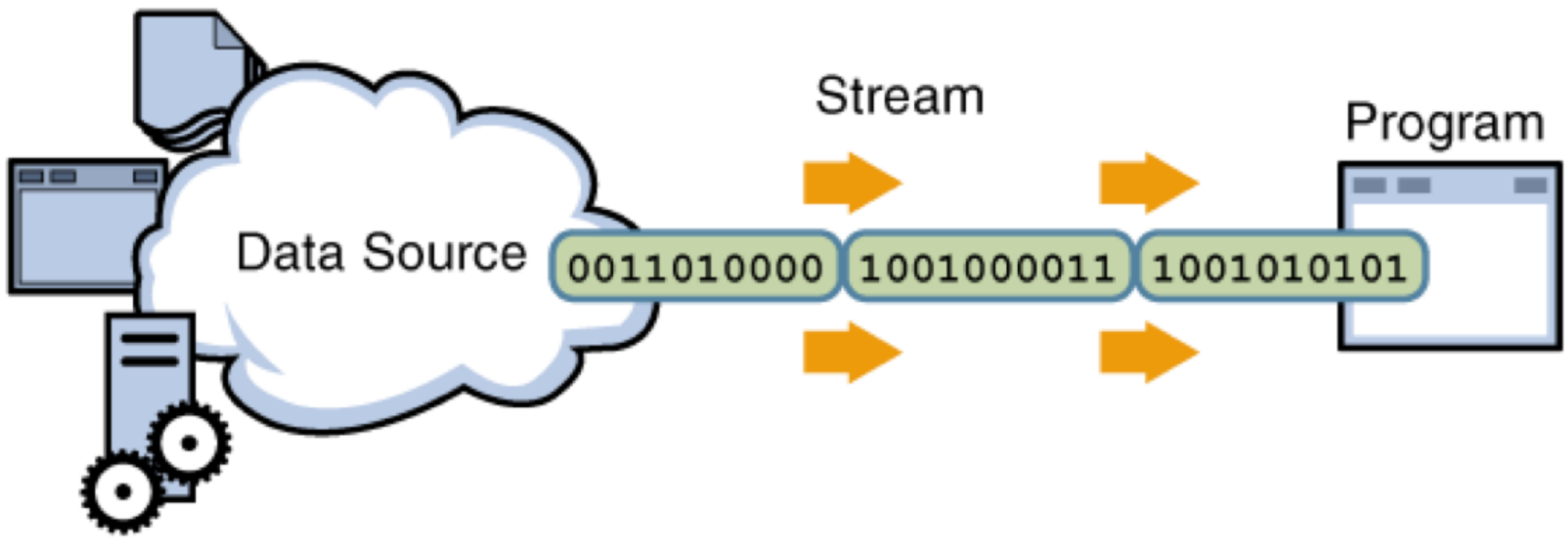
- Decorator Pattern
- Java I/O
  - Paths and Files
  - Stream
  - New I/O
- Today's Mission

Streams

- File and RandomAccessFile corresponding to paths and files in your file system
  - They are API specific to file I/O
- General I/O can read/write bytes from/to any device other than disks
  - E.g., network interface card, console, etc.
- Random access may not be possible!

Streams

- java.io.InputStream/OutputStream provides an abstraction of streams



Streams

- Streams are independent of the nature of data sources

- Bytes can be read/written sequentially by calling read() and write() repeatedly

ByteStreamCopier

```
public void copy(File src, File dest) throws IOException {
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new FileInputStream(src);
        out = new FileOutputStream(dest);
        int b;
        while ((b = in.read()) != -1) {
            out.write(b);
        }
    } finally {
        if (in != null) in.close();
        if (out != null) out.close();
    }
}
```

Types of Streams

- Two main categories in Java:
  - Byte streams: InputStream/OutputStream
  - Character streams: Reader/Writer

Endpoint Streams

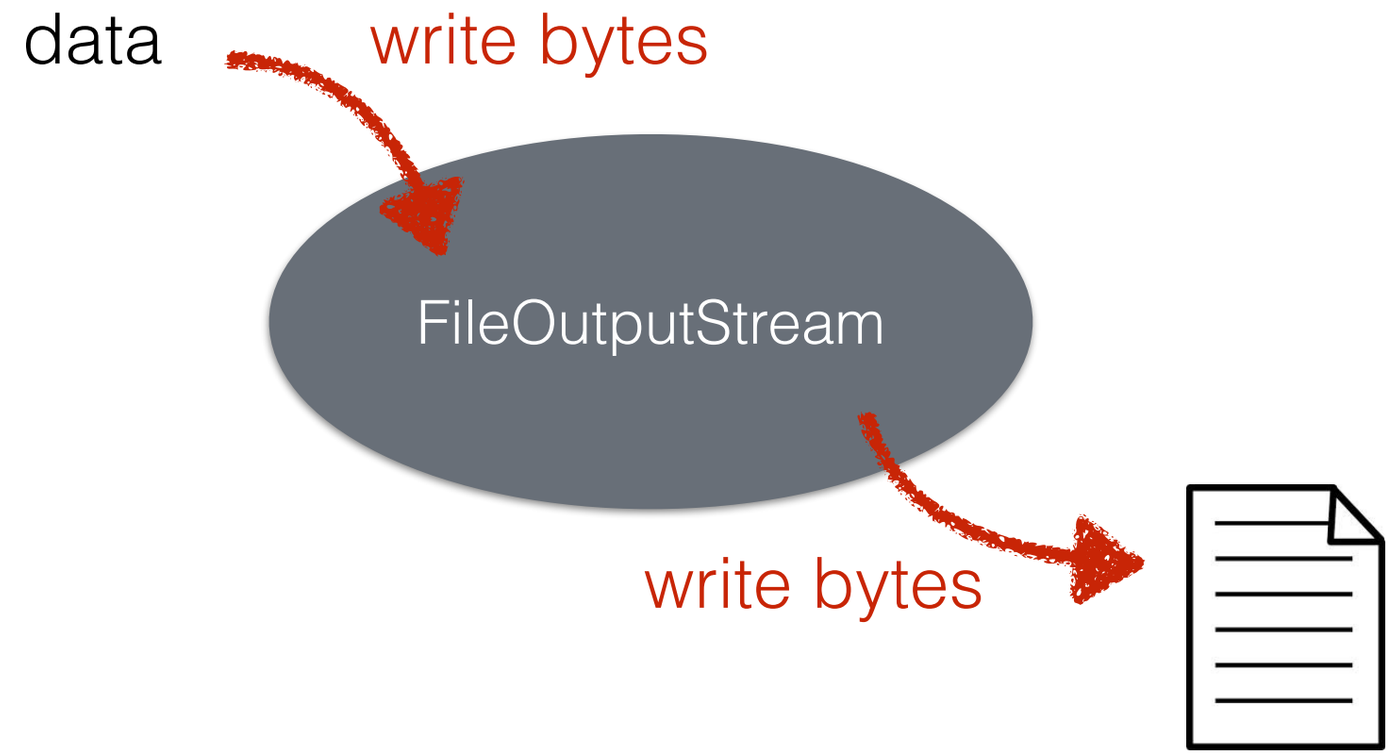
- Endpoint streams that read/write directly from/to devices:
  - E.g., FileInputStream, FileReader, ByteArrayInputStream, StringReader, etc.

Wrapper Streams

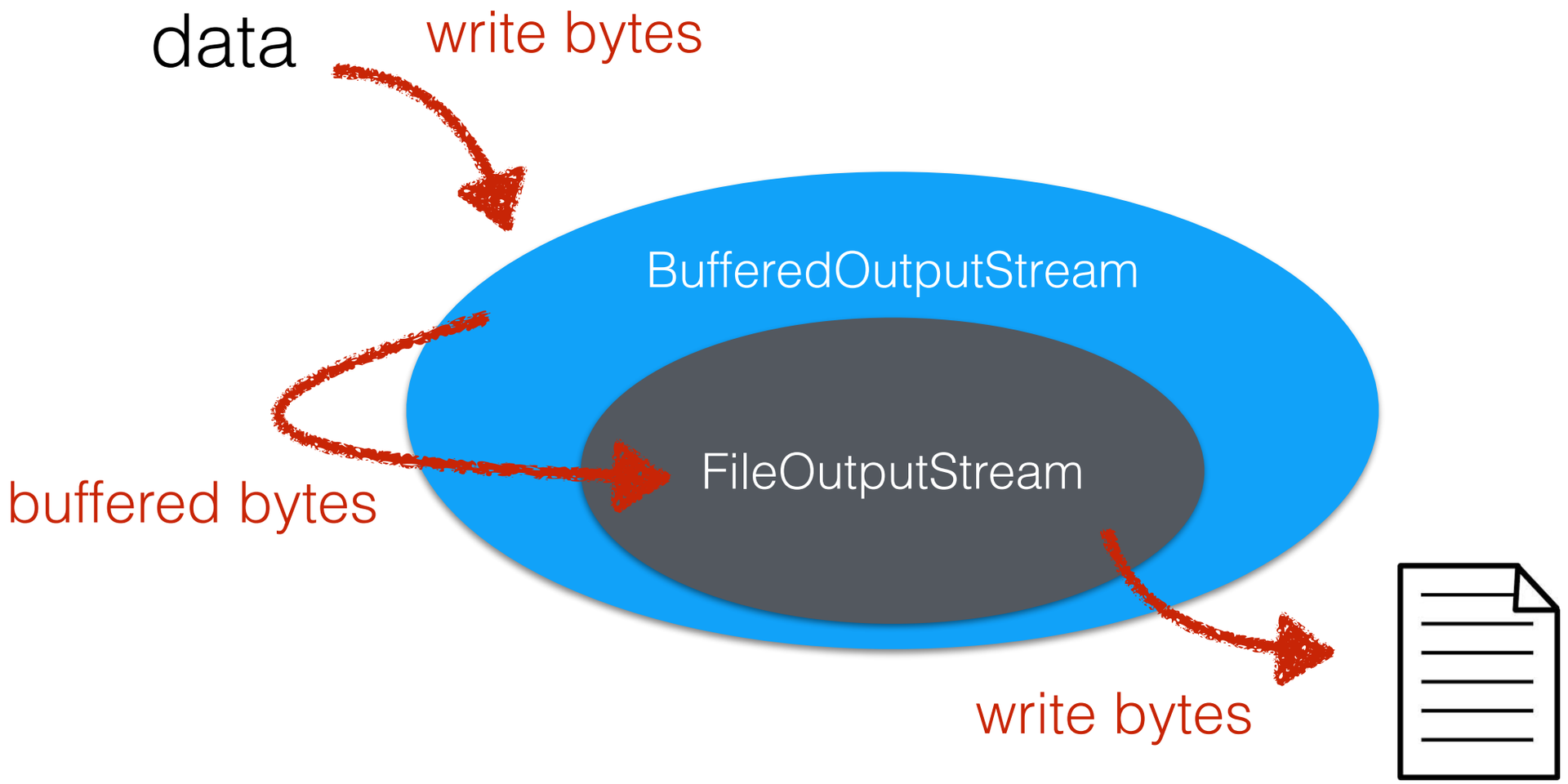
- Wrapper streams that read/write from/to endpoint streams:

- E.g., BufferedOutputStream, BufferedWriter, DataOutputStream, ObjectOutputStream, PrintWriter etc.

Decorator in Java I/O



Decorator in Java I/O



BufferedByteStreamCopier

```
public void copy(File src, File dest) throws IOException {
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new BufferedInputStream(new FileInputStream(src));
        out = new BufferedOutputStream(new FileOutputStream(dest));
        int b;

        while ((b = in.read()) != -1) {
            out.write(b);
        }
        //(optional) make sure data are reflected to disk out.
        out.flush();
        ...
    } finally {
        if (in != null) in.close();
        if (out != null) out.close(); // flush first
    }
}
```

ObjectStream

- java.io.DataInputStream/DataOutput Stream allows reading/writing primitives and objects from/to streams

ObjectStream

```
ObjectOutputStream oos = new ObjectOutputStream(...);
oos.writeObject("Today");
oos.writeObject(new Date());
ClassA a1 = new ClassA(...);
oos.writeObject(a1);
oos.close();
...
ObjectInputStream ois = new ObjectInputStream(…);
String s = (String) oos.readObject();
Date date = (Date) oos.readObject();
ClassA a2 = (ClassA) oos.readObject();
ois.close();
....
```

Object Serialization

- We say objects are deserialized/serialized upon reading/writing from streams

- Serialization is deep; all fields will be serialized recursively

- The only exceptions are those fields declared with the `transient` modifier

Serializable Interface

- An object is not serializable unless its class implements java.io.Serializable

```
public ClassA implements Serializable {
    private static final long serialVersionUID = 1L;
    ...
}
```

serialVersionUID

- What happens if you serialize an object of ClassA, modify ClassA, and then deserialize the object?

  - A serializable class should define a field named **`serialVersionUID`**

  - Increment **`serialVersionUID`** if you think the modification of **`CalssA`** makes the previously serialized objects incompatible with the new ones

Object Serialization

- ObjectInput/OutputStream provides a convenient way to perform deep cloning:

```
ObjectOutputStream oos = new ObjectOutputStream(
new ByteArrayOutputStream()); // endpoint is memory
ClassA obj = new ClassA(...);
oos.writeObject(obj);

ObjectInputStream ois = new ObjectInputStream(
new ByteArrayInputStream(oos.toByteArray()));
ClassA deepCloneObj = (ClassA) ois.readObject();
...
oos.close();
ois.close();
```

Outline

- Decorator Pattern
- Java I/O
  - Paths and Files
  - Stream
  - New I/O
- Today's Mission

Channels

- java.nio offers a new way for I/O: Channels and Buffers

- Channels are like streams, but they

  - Are bi-directional

  - Read/write data from/into Buffers only

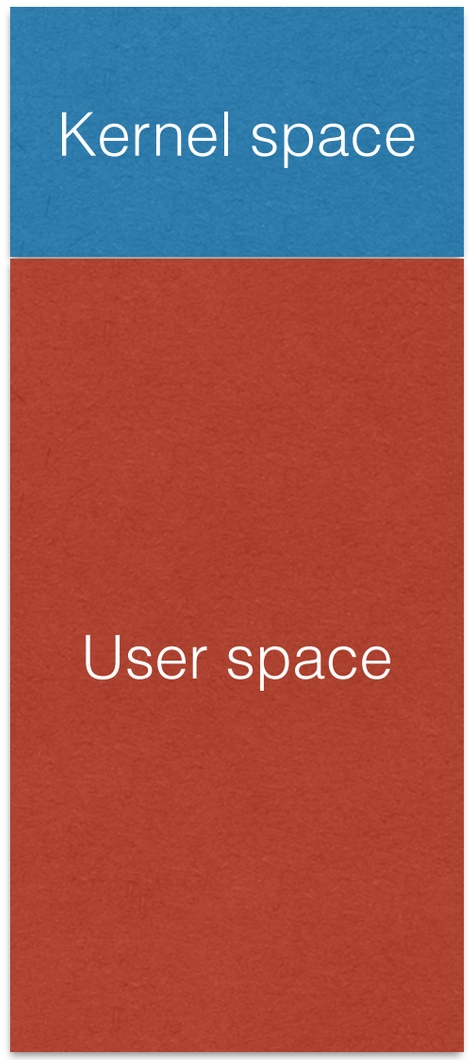
Buffers

- All data operations are performed on Buffers (data blocks in memory)

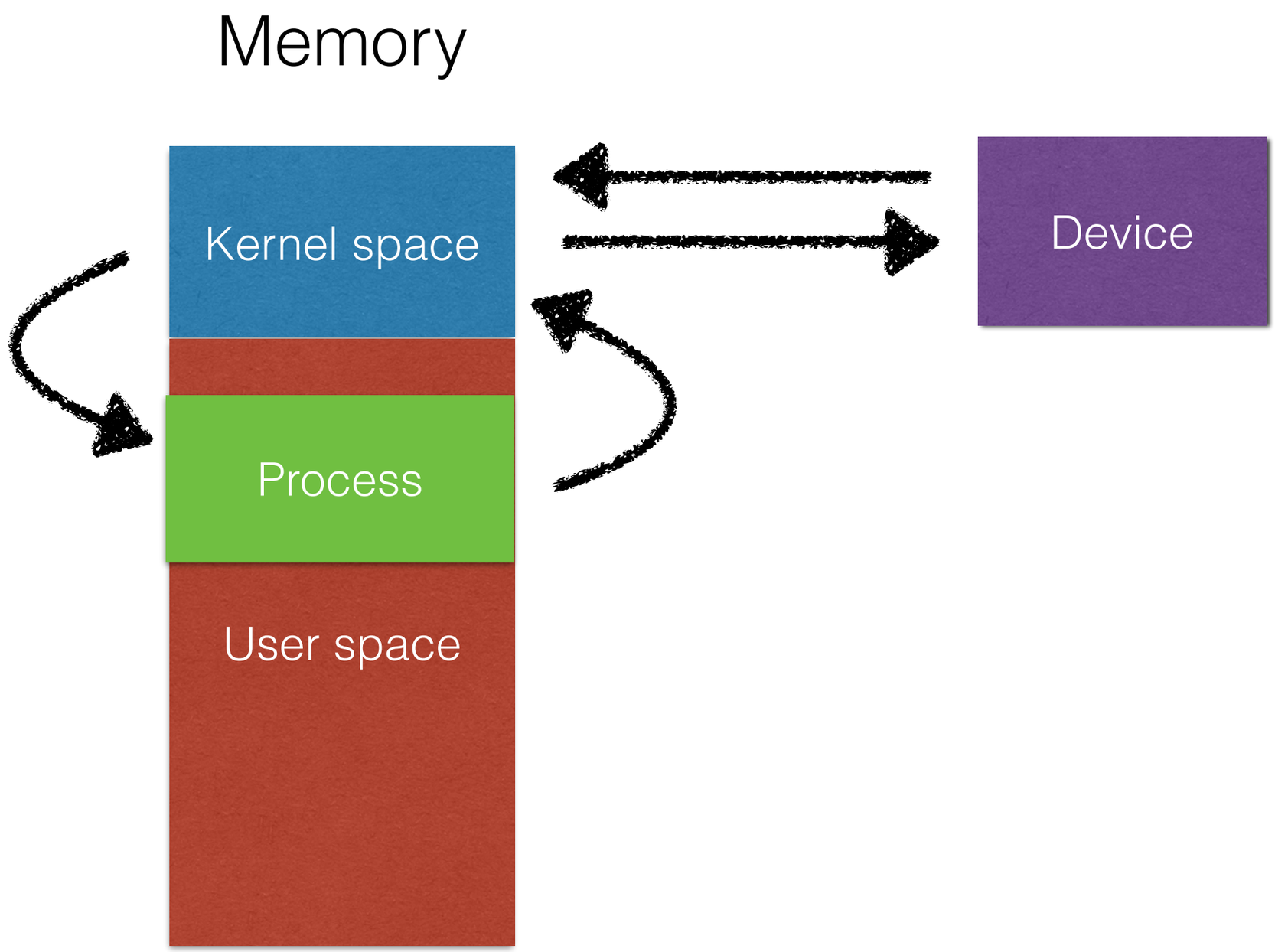
- There are buffers for all primitive types, like ByteBuffer, LongBuffer, FloatBuffer, etc.

- Buffers can be allocated directly by OS, saving memory copying from user space to kernel space before each I/O operation

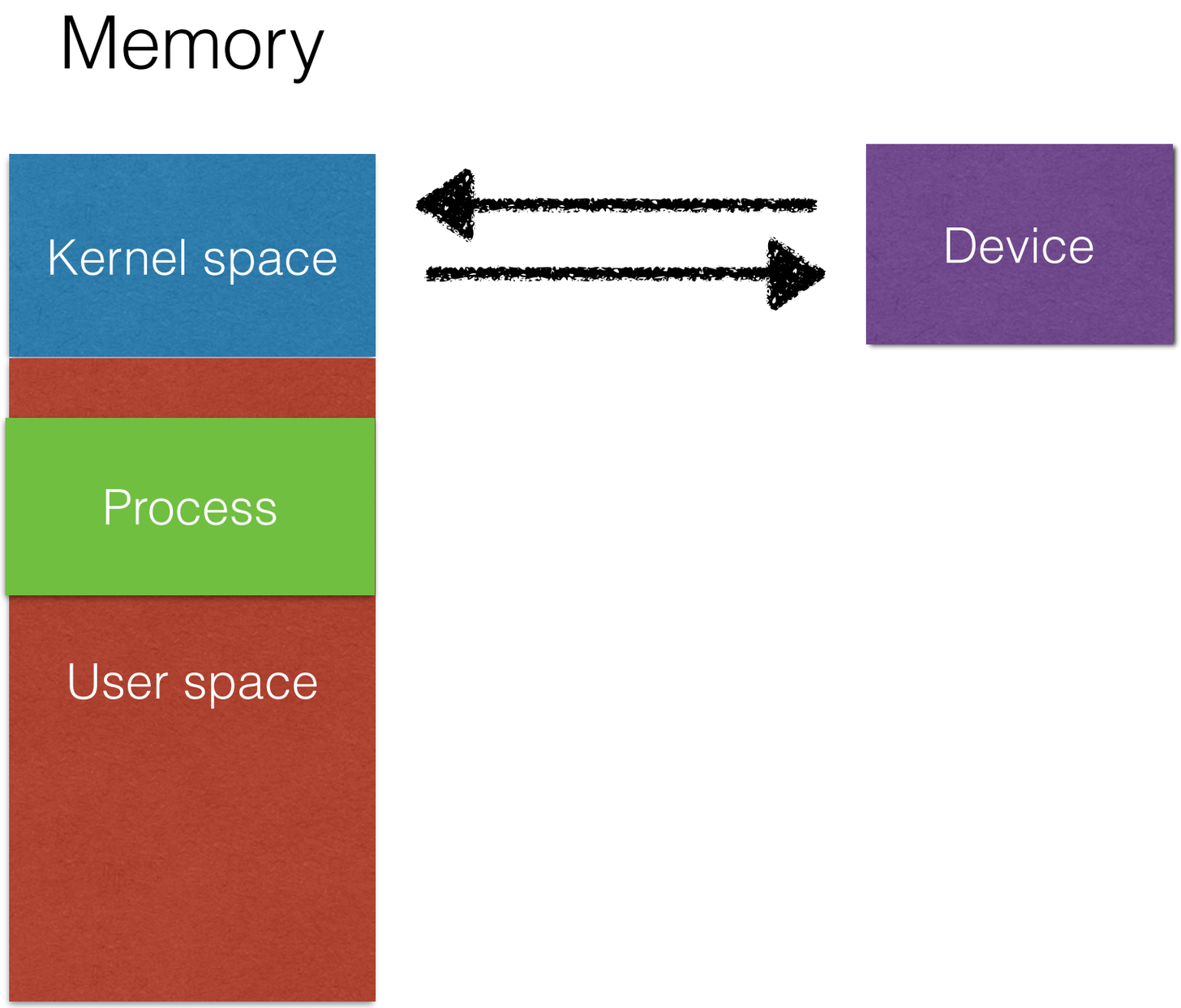
Traditional Approach



Traditional Approach

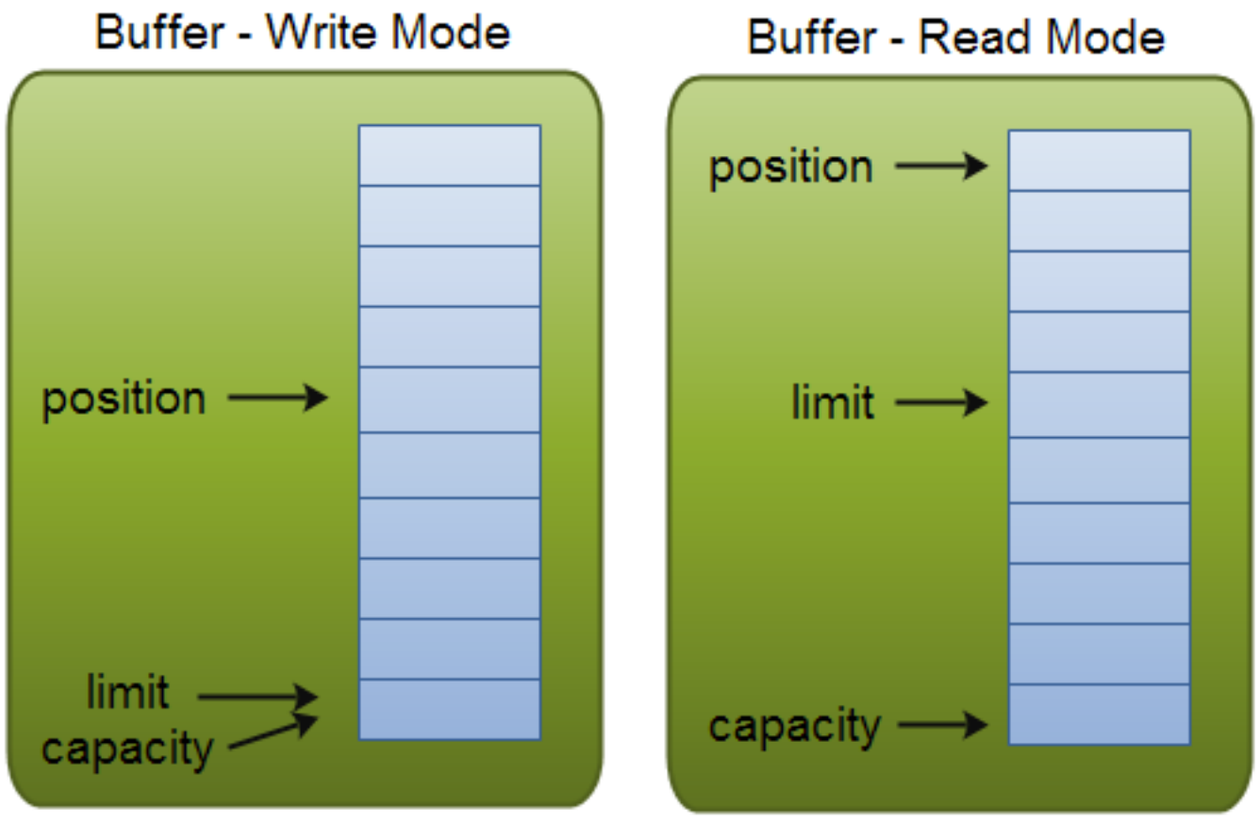


New I/O Approach



Buffers

- Buffers maintain their capacity, limit, and position, thus simplifying your code



Buffer capacity, position and limit in write and read mode.

ChannelCopier

```
public class ChannelCopier implements FileCopier {
    private static final int BUFFER_SIZE = 8192; // in bytes
    private ByteBuffer buffer = ByteBuffer.allocateDirect(BUFFER_SIZE);

    public void copy(File src, File dest) throws IOException {

        FileChannel ic = null, oc = null;
        try {
            ic = new FileInputStream(src).getChannel();
            oc = new FileOutputStream(dest).getChannel();
            while (ic.read(buffer) != -1) {
                buffer.flip();
                oc.write(buffer);
                buffer.clear();
            }
        } finally {
            if (ic != null) ic.close();
            if (oc != null) oc.close();
        }
    }
}
```

Outline

- Decorator Pattern
- Java I/O
- Today's Mission

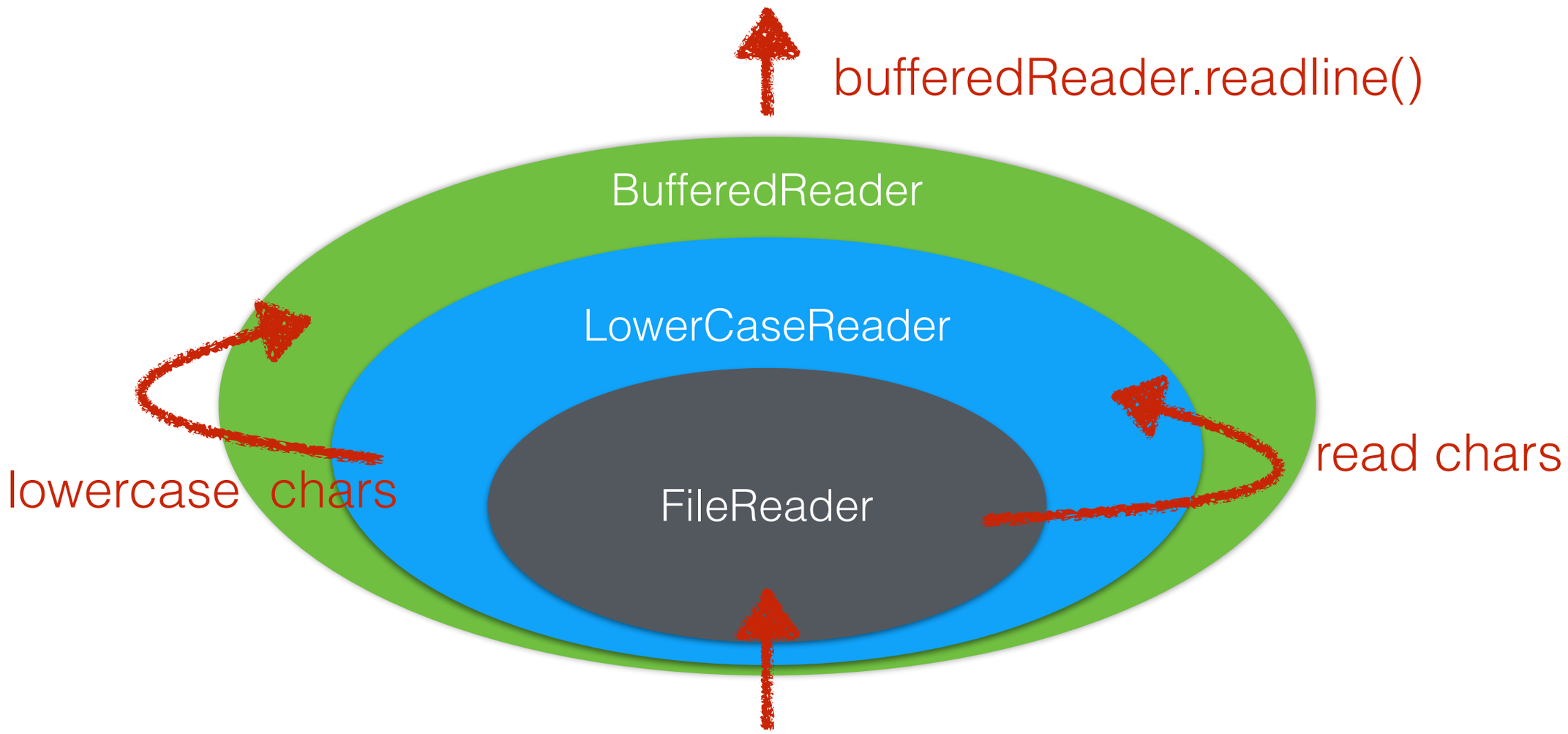
Writing Your Own Java I/O Decorator

- How about: write a decorator that converts all uppercase characters into lowercase in the input stream?

I Am Input Stream  i am input stream

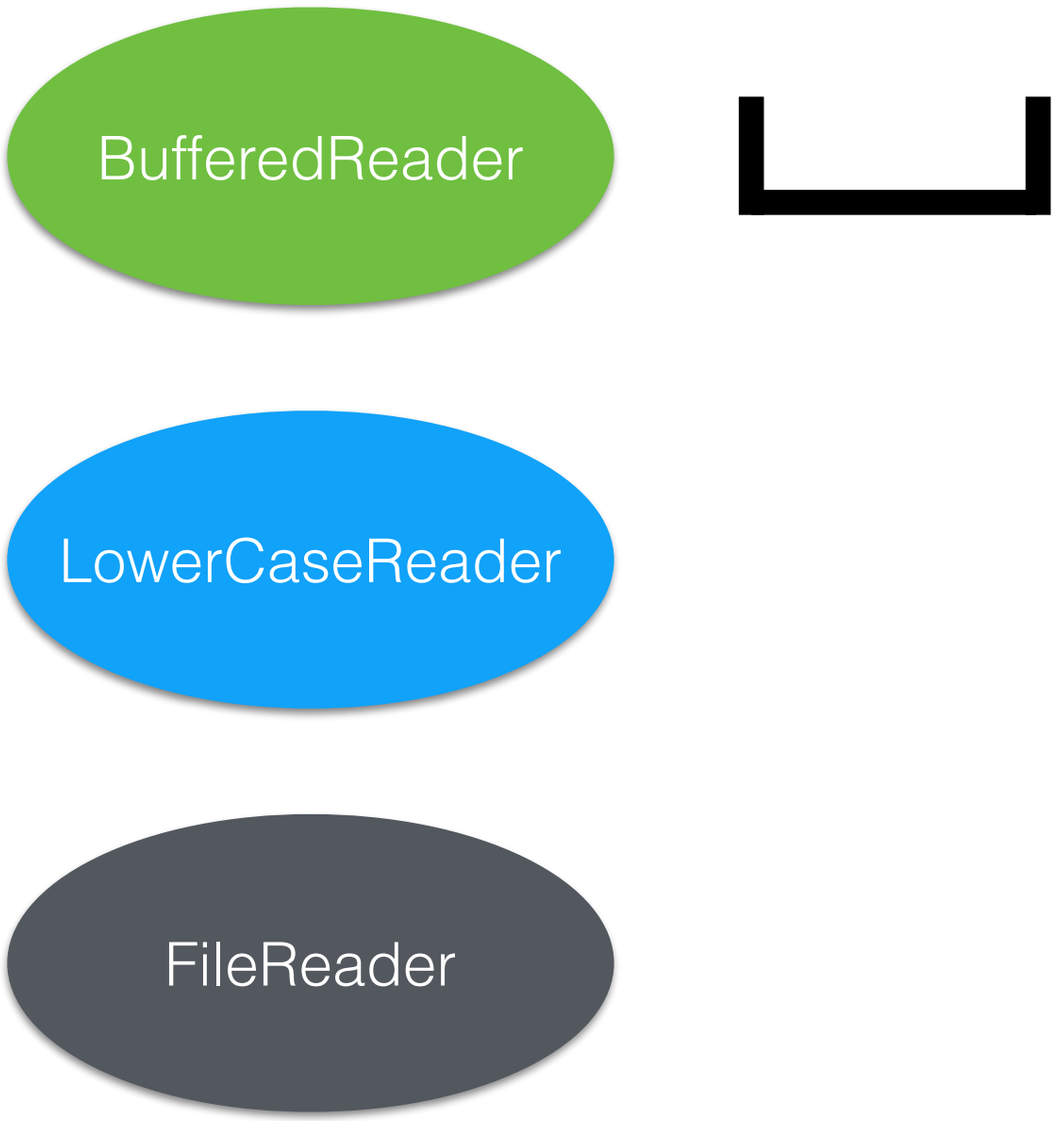
LowerCase Decorator

i am input stream



I Am Input Stream

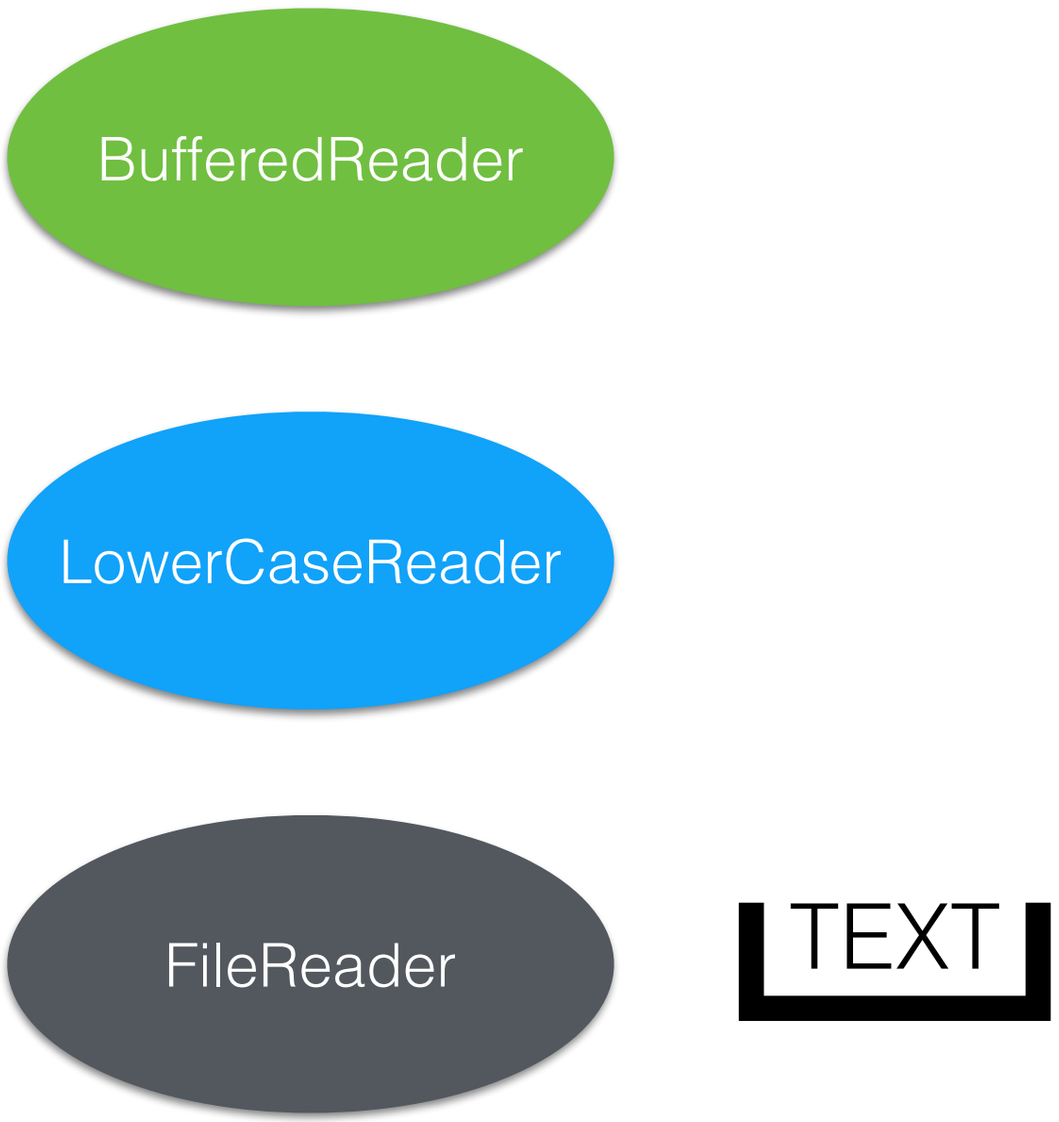
The Char Buffer



The Char Buffer

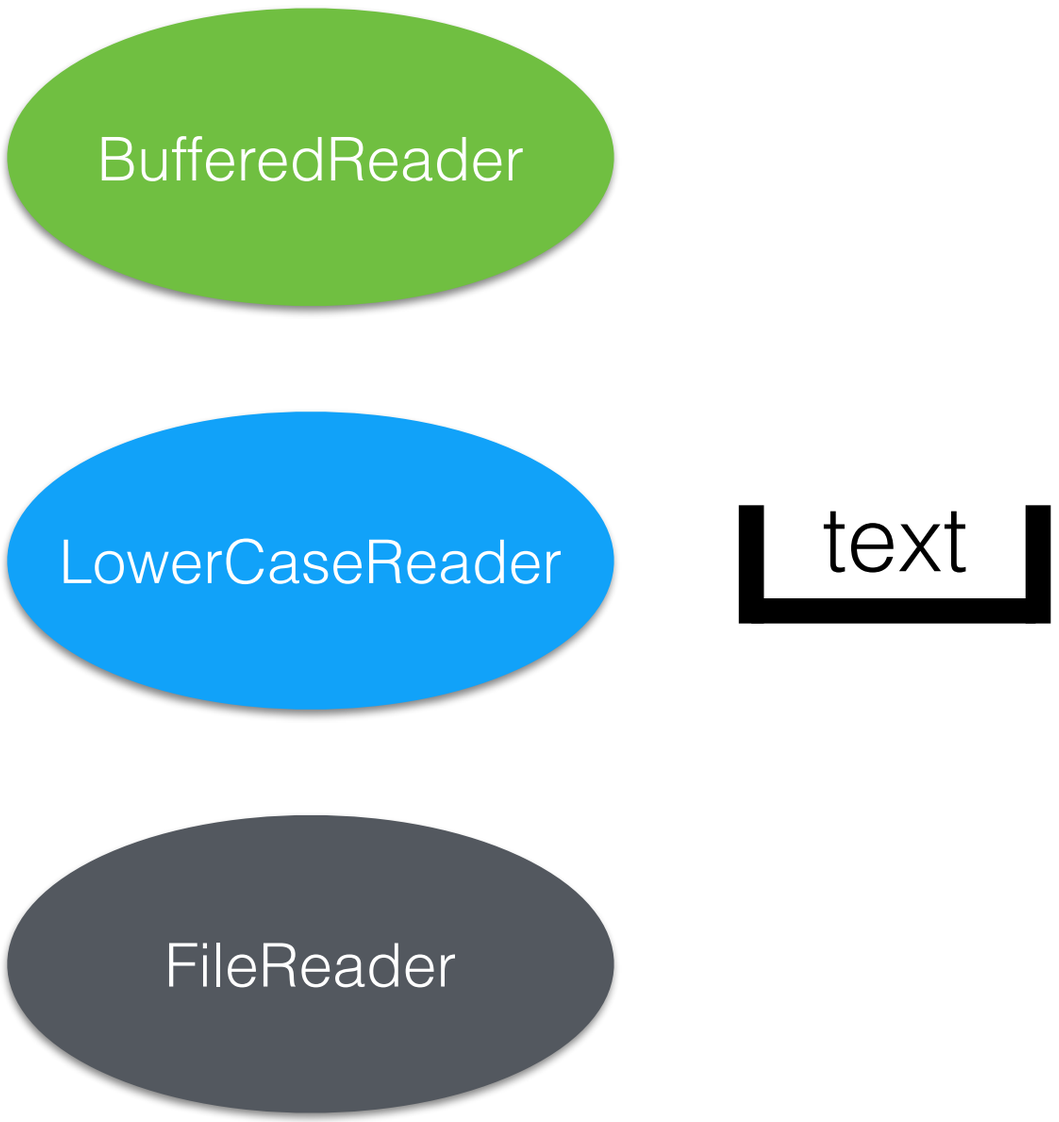
The Char Buffer

The Char Buffer

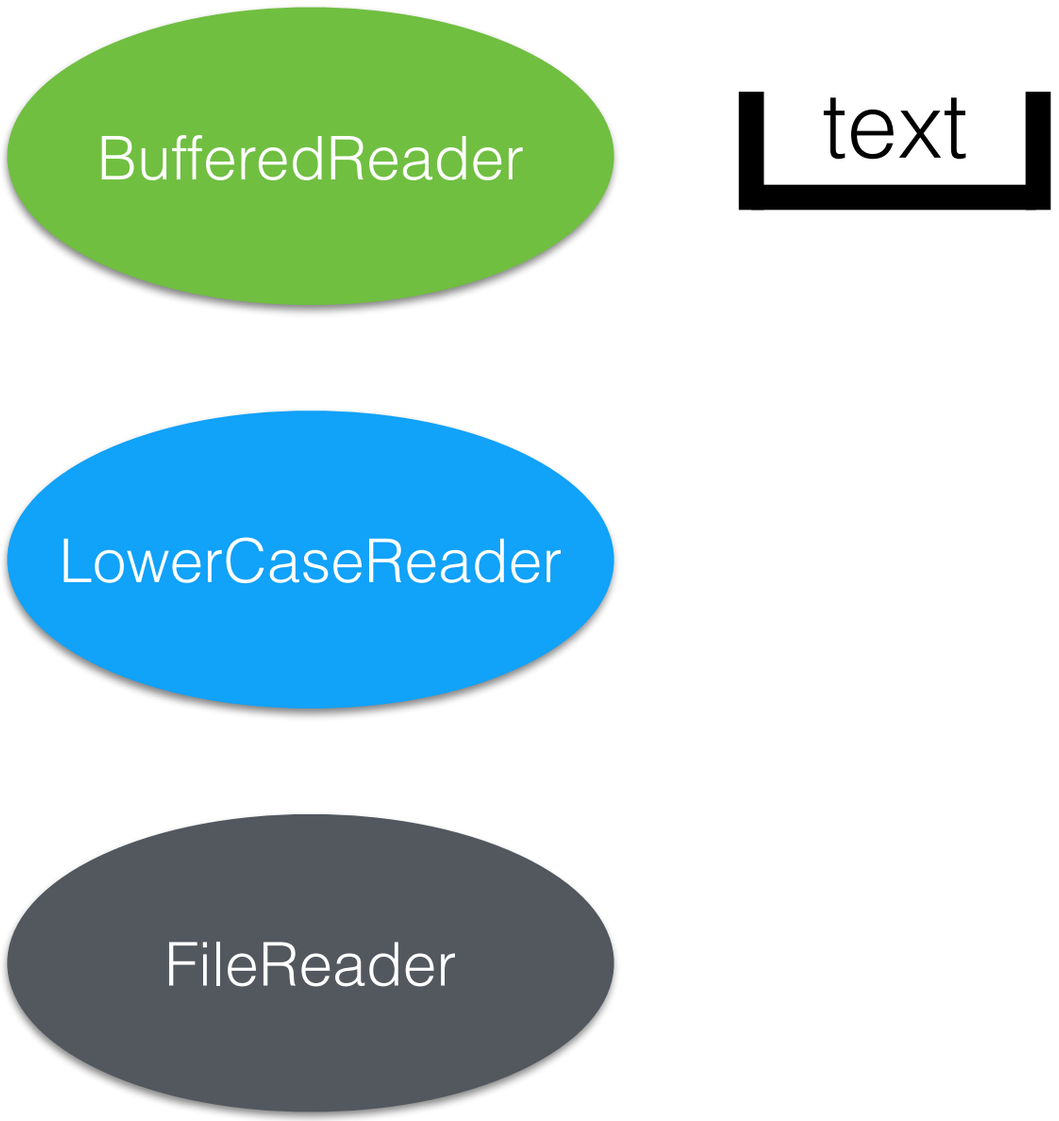


The Char Buffer

The Char Buffer



The Char Buffer



Hints

- The classes might be used:
  - BufferedReader
  - BufferedWriter
  - FileReader
  - FileWriter

Hints

- The methods might be used:
  - readline() of BufferedReader
  - write(…) of BufferedWriter