CS340100
JAVA Programming Language

Ch.6 Exception, Immutable and Serialization

Outline

- Exception
- Wrapper Class and Immutable
- Adapter Pattern
- File Operation
- Decorator Pattern
- Serializable
- Scanf in Java

Exception

- 什麼是例外(exception)?
  - JVM無法執行的指令
    - Divide by zero

```
int a = 3/0;
```

Output:
Exception in thread "main" **java.lang.ArithmeticException**: / by zero at nthu.cs.ch6.HelloWorld.main(HelloWorld.java:16)

    - Null pointer

```
Rectangle rec = null;
System.out.println(rec.area);
```

Output:
Exception in thread "main" **java.lang.NullPointerException** at nthu.cs.ch3.HelloWorld.main(HelloWorld.java:20)

Exception

- Exception的種類
  - java.lang.**RuntimeException**
    - 如divide by zero, null pointer
    - Programmer可以不處理,最後若是都沒人處理,則由JVM處理
  - java.lang.**Exception**
    - 如file not found, thread sleep
    - Programmer一定要處理

Exception Handling

- 例外的偵測與處理
  - **try-catch:** 嘗試執行某段可能發生例外的程式碼並在例外發生時做出補救
  - **throws:** 暫時不處理例外,而是讓caller來偵測與處理(逃避責任)

Try-Catch

- 如何偵測例外?
  - Try and catch

Output:

```
java.lang.ArithmeticException: / by zero
at nthu.cs.ch3.HelloWorld.devide(HelloWorld.java:22)
at nthu.cs.ch3.HelloWorld.main(HelloWorld.java:16)
c = 0
```

```
public static void main(String[ ] args){
    int c = divide(3,0);
    System.out.println("c = "+c);
}

static int divide(int a, int b){
    try{
        return a/b;
    } catch(ArithmeticException e){
        e.printStackTrace();
        return 0;
    }
}
```

即使發生了divide by zero,
程式依然能繼續往下執行

嘗試執行try block中的程式,

若在try block中發生了Arithmetic exception,將會**放棄剩下的程式碼並自動跳轉至catch block中執行**,並將所發生的exception物件傳入

Try-Catch

- 多重例外處理:當try block中可能發生不只一種exception時
  - 當try block中發生exception時
  - 依序檢查發生的exception是否為ArithmeticException或NullPointerException
  - 執行相對應的catch block

```
try{
    // ...
}
catch(ArithmeticException e1){
    // ...
}
catch(NullPointerException e2){
    // ...
}
```

Try-Catch-Finally

- **finally** block無論是否發生exception都會執行
  - try -> finally
  - try -> catch -> finally

```
try{
    // ...
}
catch(ArithmeticException e1){
    // ...
}
catch(NullPointerException e2){
    // ...
}
finally{
// ...
}
```

Throws

- throws做為method的修飾句,代表此method將不處理某例外
  - 所有**呼叫此method的caller都必須處理此例外**
  - 若是在main中透過throws來拋出例外,則此例外將由JVM來處理
  - 只要JVM收到任何一個例外,將會強制結束程式並且印出例外的資訊(printStackTrace)

Throws

```
public static void main(String[ ] args){
    try{
        writeToFile("hello");
    }
    catch(FileNotFoundException e){
        e.printStackTrace();
    }
}

static void writeToFile(String data) throws
FileNotFoundException{
    PrintWriter writer = new PrintWriter("myFile.txt");
    writer.write(data);
    writer.close();
}
```

由於呼叫的method可能會丟出FileNotFound例外,因此caller需要處理此例外

此處我們不採用try-catch來處理,而是**把例外拋出給caller處理**

由於FileNotFound不是runtime exception,因此若是不處理例外,此行會產生compile error

Exception Generating

- 利用**throw**來強制投出一個exception
  - 只要執行了throw exception指令,後面的程式將不會被執行(unreachable)
  - 通常搭配自訂的例外類別來使用

	throw	throws
功能	指令	修飾method
效果	主動丟出例外	被動丟出例外

Exception Generating

- throw: 產生exception並立刻結束當前的scope(在throw後面的程式無法被執行到)

```
public static void main(String[ ] args){
    int[] array = createArray(-1);
}

static int[] createArray(int length){
    if(length < 0){
        throw new NegativeArraySizeException();
        // unreachable codes
    }else
        return new int[length];
}
```

因NegativeArraySizeException是RuntimeException,我們可不用try-catch來執行這行程式,因此JVM將接收到例外並結束程式

由programmer手動丟出一個exception,此時會自動脫離當前的method

Exception Generating

- 繼承Exception之自定義類別
  - 依據需求可繼承自RuntimeException或Exception

```
class MyException extends Exception {

    public MyException(String msg){
        super(msg);
    }
}
```

繼承java.lang.Exception的類別,並實作了一個與父類別相同的建構子

補充: try-catch注意事項

- 一次catch所有的exception
  - 利用Exception類別來接丟出的例外物件(隱式轉換)

當用多重catch時,需要注意父類別的例外要最後再catch(why?)

```
try{
    //...
}
catch(NullPointerException e){
    System.out.println("NullPointerException occurs");
}
catch(Exception e){
    System.out.println("Another exception occurs");
}
```

補充: try-catch注意事項

- finally的使用時機: **關閉資源**

```
FileOutputStream fos = null;

try {
    fos = new FileOutputStream("output.txt");
} catch (Exception e) {
// complain to user
} finally {
    if (fos != null){
        try{
            fos.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

確定物件實體存在才進行close

避免在close過程中出現例外,再加一層try-catch來保護

補充: try-catch注意事項

- Try-with-resource
  - Java 7.0提供的一項新功能
  - 允許try使用小括號來建立(檔案)資源

```
try(FileInputStream fis = new FileInputStream("file.txt")){
    fis.read();
    fis.close();

} catch(IOException e){
    e.printStackTrace();
}
```

補充: try-catch注意事項

- **不要用try-catch來作流程控制**
  - 大量的使用try-catch將導致程式執行速度緩慢

```
String[] array = {"Java","NTHU","CS"};

int i = 0;

while(true){
    try{
        System.out.println(array[i++]);
    }catch(IndexOutOfBoundsException e){
        break;
    }
}
```

補充: try-catch注意事項

- **finally區塊永遠都會被執行**(即使在try或catch block中執行了return)

```
public static void main(String[] args) {

    try{
        return;
    }
    finally{
        System.out.println("hello");
    }

}
```

Output:
hello

Wrapper Class

- **將基本資料型態(primitive type)的變數包裝成物件型態(object type)**

Primitive type	Object type
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Wrapper Class

- 提供各種**static用途與object用途**
  - Integer.parseInt, Integer.MAX_VALUE
- 支援物件的資料流(data stream)型態
- **支援泛型(Generics),集合(collection)**

```
Integer myInt = new Integer(10);

System.out.println("static use: "+Integer.MAX_VALUE);
System.out.println("object use: "+myInt.doubleValue());
```

Output:
static use: 2147483647
object use: 10.0

Boxing, Unboxing

- **裝箱(boxing):** 將int變數打包為Integer物件
- **拆箱(unboxing):** 將Integer物件拆開成int (也可以拆開成為double或其他型態)

```
int a = 10;

Integer myInt = new Integer(a);

int b = myInt.intValue();
```

boxing

unboxing

Auto Boxing, Unboxing

- **自動裝箱(auto-boxing):** 直接把int變數assign給Integer物件
- **自動拆箱(auto-unboxing):** 直接把Integer物件assign給int變數

```
int a = 10;

Integer myInt = a;

int b = myInt;
```

Auto-boxing

Auto-unboxing

Immutable

- Integer雖然支援++以及+=操作,但實際上並非修改物件內容,而是建立另一個新物件
  - 以下面兩段程式為例:

```
int a = 10;

Integer myInt1 = 0;
Integer myInt2 = a + 10;

myInt1 = myInt2 + 5;
myInt2++;
```

```
int a = 10;

Integer myInt1 = new Integer(0);
Integer myInt2 = new Integer(a + 10);

myInt1 = new Integer(myInt2.intValue() + 5);
myInt2 = new Integer(myInt2.intValue() + 1);
```

Wrapper class所建立的物件實體,一律無法修改其內容,此特性稱為Immutable, 此外String也是immutable的

Constant Pool

- 為何a1不等於a2, 但b1卻等於b2呢?
  - JVM快取了所有-128~127之間的Integer實體,當進行自動裝箱時就會使用暫存好的實體

```
Integer a1 = 1000;
Integer a2 = 1000;
Integer a3 = new Integer(1000);

System.out.println("a1=a2? "+(a1==a2));
System.out.println("a2=a3? "+(a2==a3));

Integer b1 = 1;
Integer b2 = 1;
Integer b3 = new Integer(1);

System.out.println("b1=b2? "+(b1==b2));
System.out.println("b2=b3? "+(b2==b3));
```

Output:

```
a1=a2? false
a2=a3? false
b1=b2? true
b2=b3? false
```

Constant Pool

- **常數庫**用來保存常常會被使用到的物件實體,讓常用到的字串或整數可以**共享同一塊記憶體空間**
  - 廣義來說,String算是char[]的wrapper class,因此也有constant pool的支援

Constant Pool

- 編譯時期可以確定的常數,會先嘗試從常數庫中找尋可用的快取物件

```
String str = "Hello";
final String fstr = "Hello";

String s1 = "HelloWorld";
String s2 = "Hello"+"World";
String s3 = str+"World";
String s4 = fstr+"World";


System.out.println("s1=s2? "+(s1==s2));
System.out.println("s1=s3? "+(s1==s3));
System.out.println("s1=s4? "+(s1==s4));
```

Output:

```
s1=s2? true
s1=s3? false
s1=s4? true
```

Immutable

- 為什麼wrapper class object要immutable?
  - Benefits from constant pool
  - Parameter consistency
  - Thread safety

補充: Adapter Pattern

- 假設我正在開發一個大型project
  - 首先我寫了一個類別Zombie並實作一個公開的介面IMonster

```
public interface IMonster {
    void showInfo();
    void setHp(double hp);
}

class Zombie implements IMonster {
    double hp;
    public void showInfo(){
        System.out.println("The zombie hp = "+hp);
    }

    public void setHp(double hp) {
        this.hp = hp;
    }
}
```

補充: Adapter Pattern

- 小明突然決定要加入我的project,並整合他和我的程式
  - 他寫了一個類別Animal, 希望能在不改變Animal內容的條件下實作IMonster介面

```
public interface IMonster {
    void showInfo();
    void setHp(double hp);
}

class Animal //implements IMonster {
    int hp;

    public void printInfo(){
        System.out.println("The animal hp = "+hp);
    }

    public void setHp(int hp){
        this.hp = hp;
    }
}
```

由於Animal在同學自己的程式中已經被使用,因此不能要求他修改這個class

因為當初沒有溝通過,Animal無法直接實作IMonster

補充: Adapter Pattern

- 此時我們可以幫小明寫一個**轉接器(adapter)**類別,讓他的類別可以(隱性的)實作IMonster

```
public class AnimalAdapter  implements IMonster{
    private Animal instance;

    public AnimalAdapter(Animal animal){
        instance = animal;
    }

    public void showInfo(){
        instance.printInfo();
    }
    public void setHp(double hp){
        instance.setHp((int)hp);
    }
}
```

將Animal object包裝在AnimalAdapter當中,其實這就是廣義的wrapper class

透過adapter就可以連結interface的method與實體的method

補充: Adapter Pattern

- **Adapter可以把某個類別包裝在另一個類別**,Wrapper就是屬於這個概念
  - 實作了AnimalAdapter後,我和小明的程式就可以整合在一起了

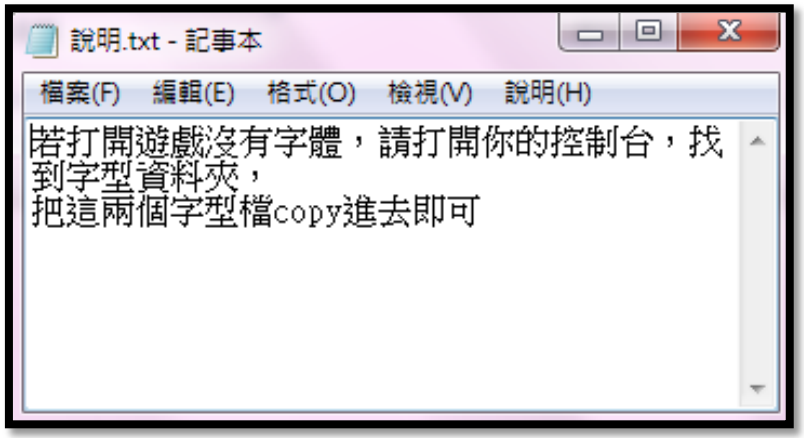
```
public static void main(String[] args)
{

    IMonster m;

    m = new Zombie();

   //m = new Animal();

    m = new AnimalAdapter(new Animal());
}
```

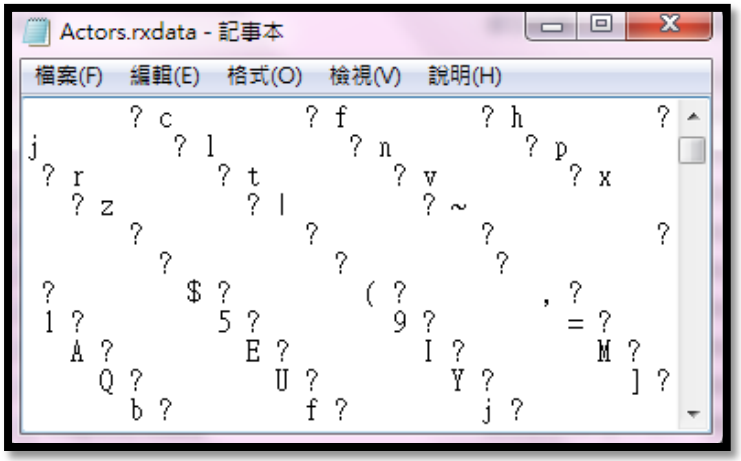
由於Animal無法實作IMonster,
因此不能做此assignment

必須透過額外一層包裝來
達成這個目的

Text/Binary Files

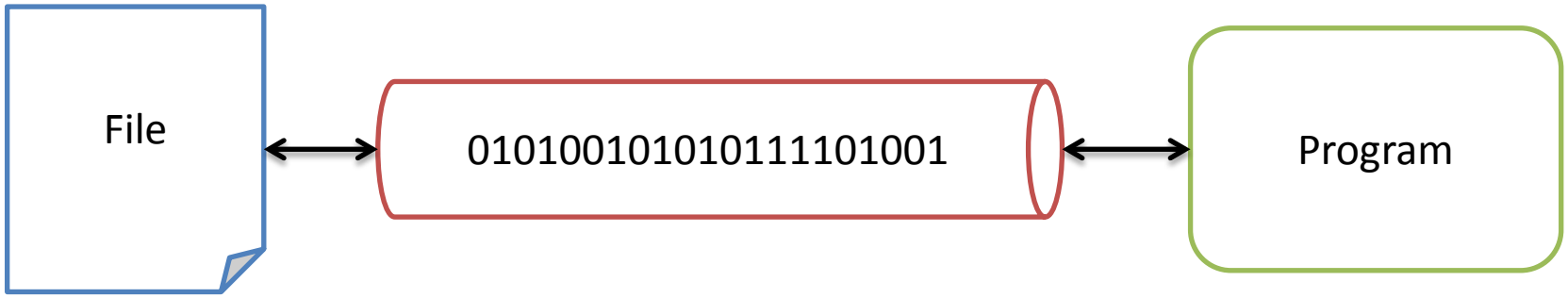
- 文字檔案 v.s. 位元檔案
  - 文字檔案使用char作為讀寫的基本單位
  - 位元檔案使用byte作為讀寫的基本單位





File Stream

- 無論是文字檔案或是位元檔案,皆可以**位元串流(byte stream)**來讀寫



FileInputStream and FileOutputStream

Stream本身由byte組成,因此寫入的基本單位都是byte

開啟檔案時可選擇是否用append模式

```
try {
    FileOutputStream fos = new FileOutputStream("text.txt",false);
    fos.write("Write some bytes\r\n".getBytes());
    fos.write("Hello World".getBytes());
    fos.close();

    byte[] buf = new byte[256];
    FileInputStream fis = new FileInputStream("text.txt");
    int num = fis.read(buf);

    String str = new String(buf,0,num);
    System.out.println(str);
    fis.close();

} catch (IOException e) {
    e.printStackTrace();
}
```

嘗試讀入256 個bytes進來並存在buf

Output:

```
Write some bytes
Hello World
```

OutputStreamWriter and InputStreamReader

由於byte的操作並不方便,因此我們將FileOutputStream加上一些字元(char)處理的功能,重新包裝成OutputStreamWriter

```
try {
    FileOutputStream fos = new FileOutputStream("text.txt",false);
    OutputStreamWriter writer = new OutputStreamWriter(fos);

    writer.write("Write some bytes\r\n");
    writer.write("Hello World");
    fos.close(); writer.close();

    char[] cbuf = new char[256];
    FileInputStream fis = new FileInputStream("text.txt");
    InputStreamReader reader = new InputStreamReader(fis);

    int num = reader.read(cbuf);
    String str = new String(cbuf,0,num);
    System.out.println(str);
    fis.close(); reader.close();
}
```

加入字元處理的功能以後就可以直接寫字串囉

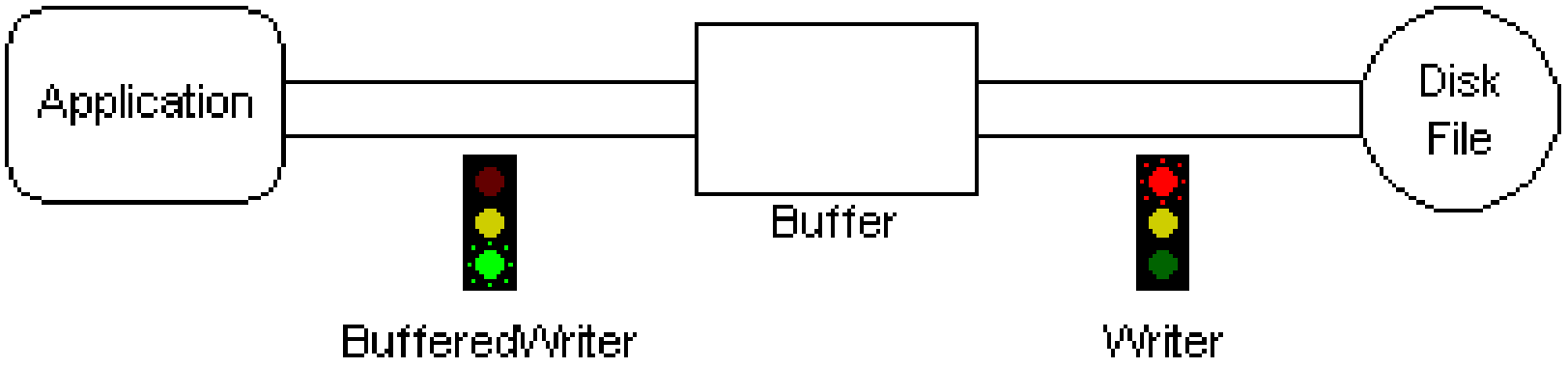
試圖讀入256個字元進來並存在cbuf

Output:

```
Write some bytes
Hello World
```

Buffering

- **緩衝區(buffer)**有效的提升I/O讀寫的效率
  - 將需要讀寫的資料暫存在buffer中
  - 等到適當時機,一次將buffer中所有資料寫入檔案(讀入程式)



BufferedWriter and BufferedReader

將原本的字元寫入機制(char writer)包裝在一個具有緩衝區的寫入機制(buffered writer)中,buffer size可調整,預設為8192 bytes

```
FileOutputStream fos = new FileOutputStream("text.txt",false);
OutputStreamWriter writer = new OutputStreamWriter(fos);
BufferedWriter bufOut = new BufferedWriter(writer);

bufOut.write("Write some bytes\r\nHello World");
bufOut.flush();
```

將緩衝區內的所有字元都強制往檔案輸送

```
FileInputStream fis = new FileInputStream("text.txt");
InputStreamReader reader = new InputStreamReader(fis);
BufferedReader bufIn = new BufferedReader(reader);

String str;
while((str = bufIn.readLine()) != null)
    System.out.println(str);

//Close all resources
```

由於有緩衝區,因此reader可以辨識換行的情況,所以可以一次讀入一行

補充: Decorator Pattern

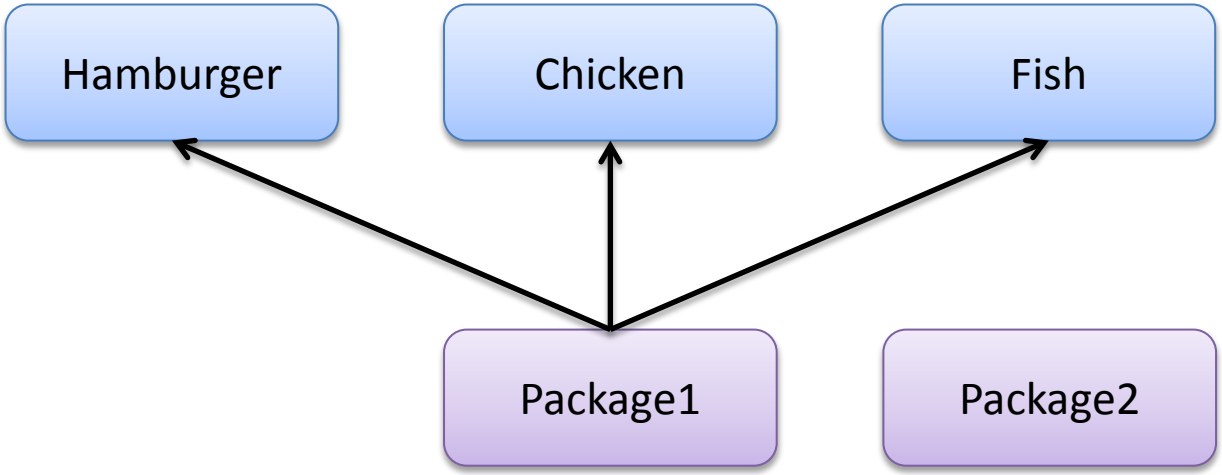
- 假設我正在寫一個速食點餐程式
  - 我們有一個漢堡類別,但此外顧客可能會買小菜(薯條,可樂或是冰淇淋)

直覺的做法就是設定主餐為父類別,主餐搭小菜為子類別

```
class Hamburger {
    public int price(){
        return 49;
    }
}

class Package1 extends Hamburger {
    @Override
    public int price(){
        return super.price()+30;
    }
}
```

補充: Decorator Pattern

- 但是如果我們有麥香魚,麥香雞,六塊雞塊等主餐時,無法讓小菜class多重繼承這麼多類別



補充: Decorator Pattern

- 因此我們決定要用一個主餐介面讓所有主餐類別實作

```
interface Meal {
    int price();
    String getContent();
}
```

```
class Hamburger implements Meal {
    public int price(){
        return 49;
    }
    public String getContent(){
        return "丹丹漢堡";
    }
}
class Chicken implements Meal {
    public int price(){
        return 59;
    }
    public String getContent(){
        return "吮指炸雞";
    }
}
```

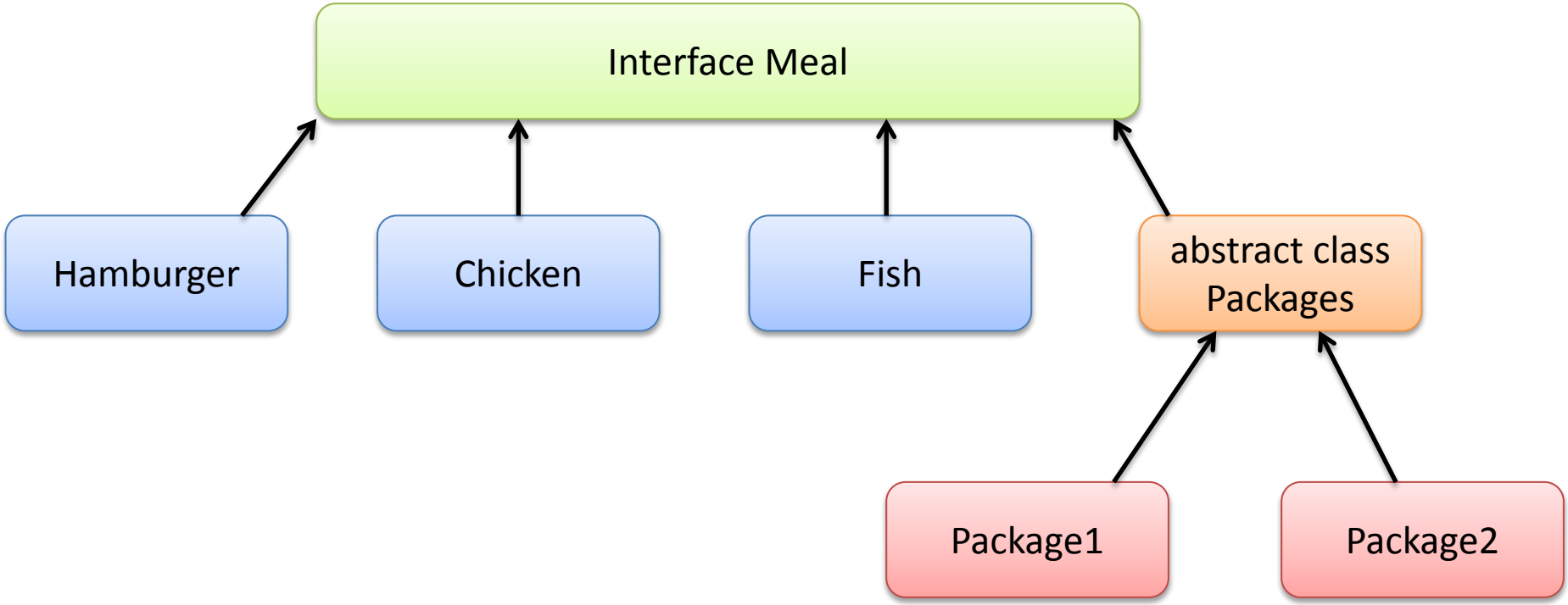
- 並且令套餐也實作Meal,然後於套餐中包入一個主菜實體,就大功告成囉

```
class Package1 implements Meal {
    private Meal meal;

    public Package1(Meal m){
        meal = m;
    }

    public int price(){
        return meal.price()+30;
    }

    public String getContent(){
        return meal.getContent()+"+薯條+可樂";
    }
}
```

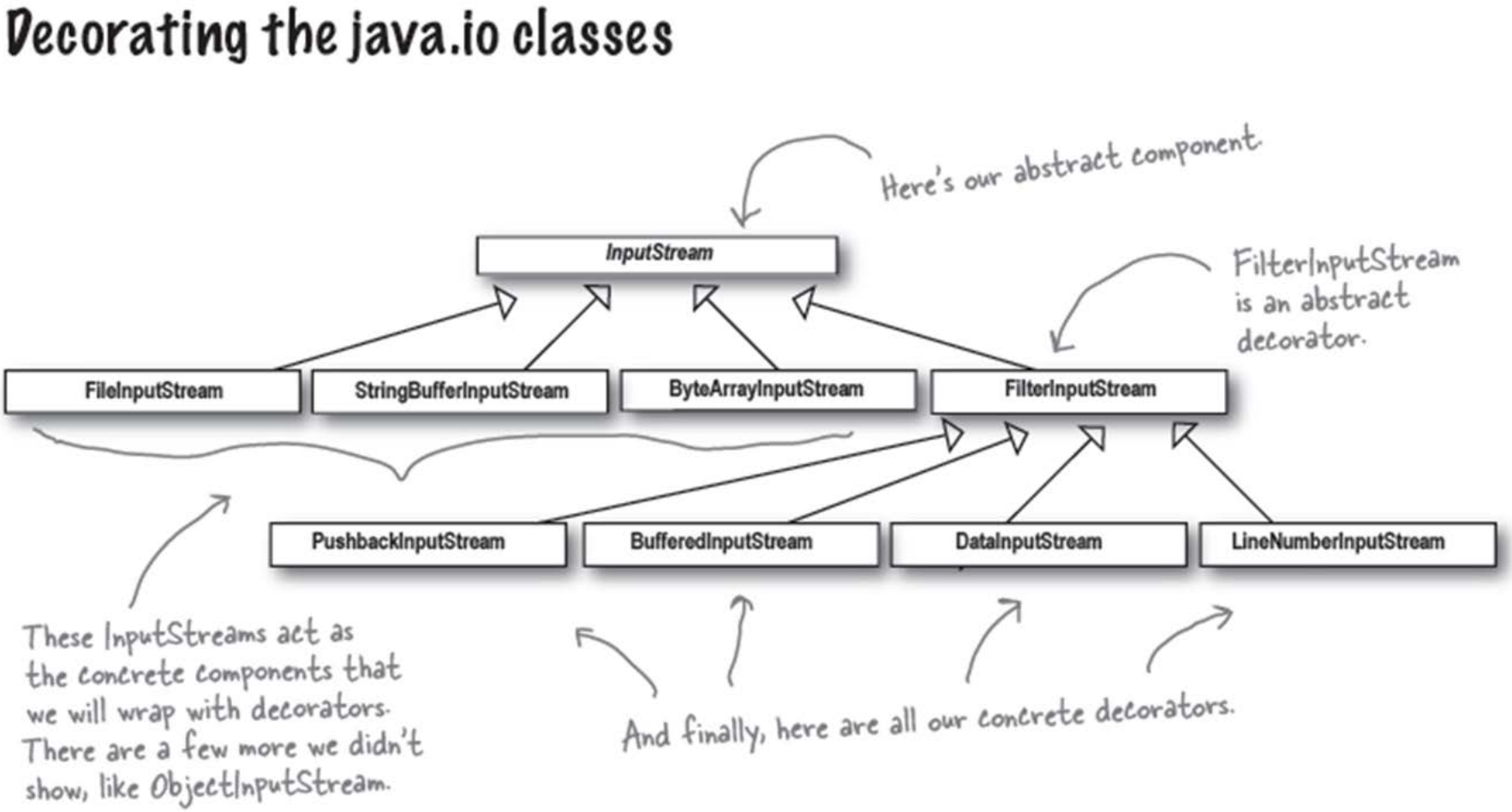
```
public static void main(String[] args) {
    Package1 p = new Package1(new Chicken());
    System.out.println(p.getContent());
}
```

補充: Decorator Pattern

- 通常我們提供不只一種小菜,所以再加上一個abstract class來提供重複利用的程式

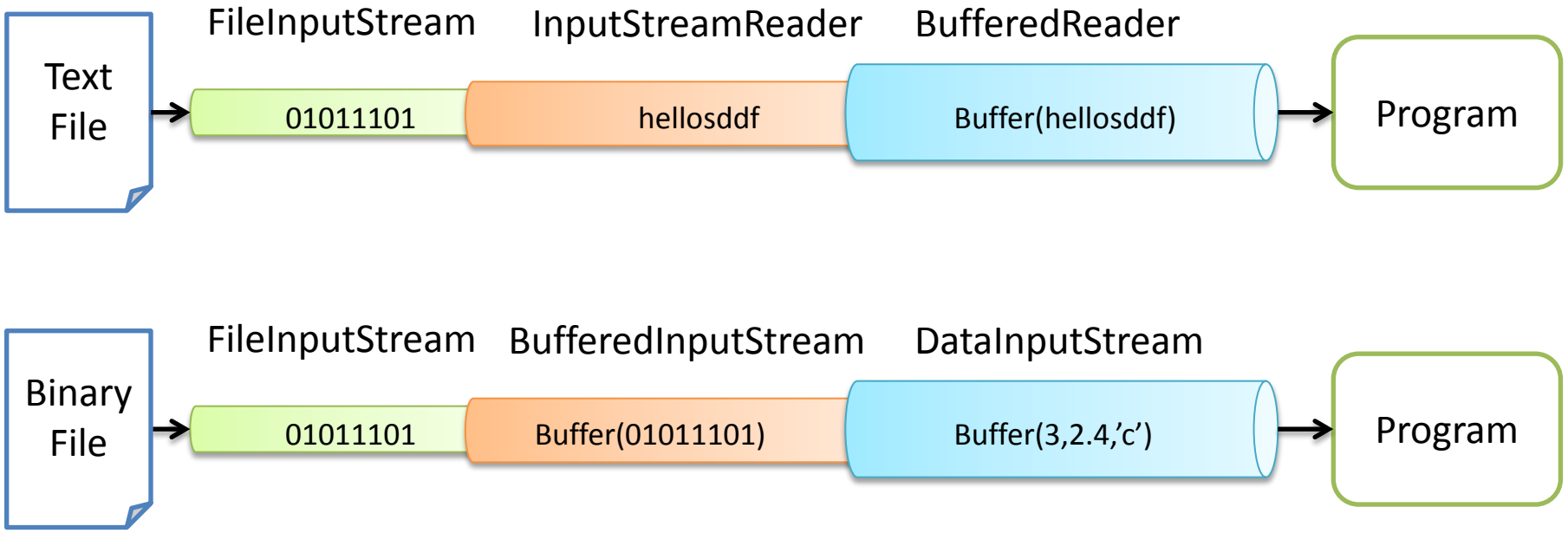


StreamReader V.S. Decorator



StreamReader V.S. Decorator

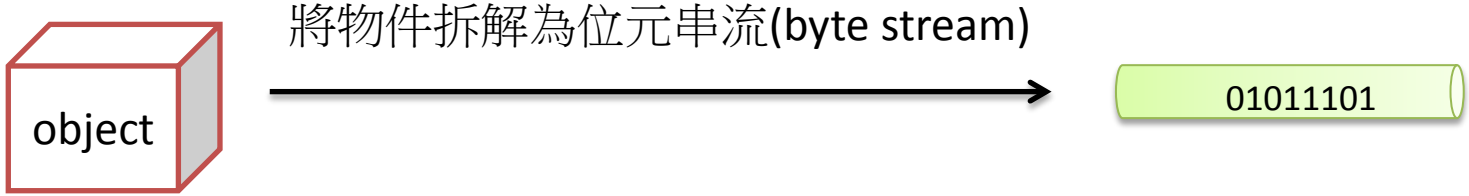
- Java在檔案處理中用了大量的decorator pattern



Serializable

```
public class Student implements Serializable{

    String name;
    int grade;

    public Student(String n, int g){
        name = n;
        grade = g;
    }

    public void show(){
        System.out.println(name+"考了"+grade+"分");
    }
}
```

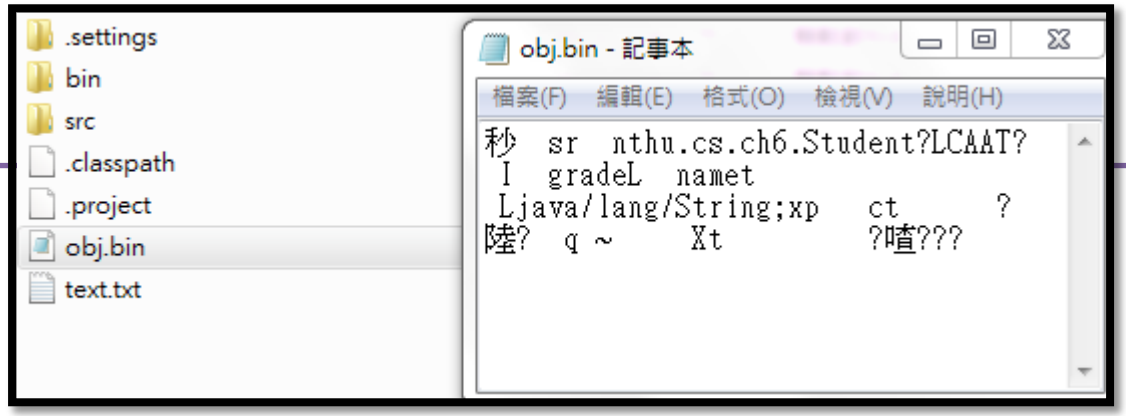
實作Serializable介面的類別所建立的物件,才能被寫入(讀出)檔案



Serializable

ObjectOutputStream用來寫入"物件"

```
public static void main(String[] args) throws IOException{


    ObjectOutputStream os = new ObjectOutputStream(
                                new FileOutputStream("obj.bin"));

    Student s1 = new Student("王泰元",99);
    Student s2 = new Student("陳冠文",88);

    os.writeObject(s1);
    os.writeObject(s2);

    os.close();
}
```



Serializable

ObjectInputStream用來寫入"物件"

```
ObjectInputStream is = new ObjectInputStream(
                           new FileInputStream("obj.bin"));
try {
    Student s3 = (Student) is.readObject();
    Student s4 = (Student) is.readObject();

    s3.show();
    s4.show();

} catch (ClassNotFoundException e) {
    e.printStackTrace();
}

is.close();
```

readObject回傳的是Object,因此需進行強制轉型

有可能讀入的Object根本不是Student類別,因此需考慮此例外

Output:

王泰元考了99分
陳冠文考了88分

Serialization and Deserialization

- 寫檔時需要將物件**序列化**,反之讀檔時則進行**反序列化**
- 若某A類別的物件中含有B類別的物件,則**A與B都要實作Serializable才能對A進行讀寫檔**

Serialization and Deserialization

```
public class Student implements Serializable{
    String name;
    int grade;

    public Student(String n, int g){
        name = n;
        grade = g;
    }

    private void writeObject(ObjectOutputStream os) throws IOException{
        os.writeUTF(name);
        os.writeInt(grade);
    }

    private void readObject(ObjectInputStream is) throws IOException{
        name = is.readUTF();
        grade = is.readInt();
    }
}
```

藉由反序列化所重組的物件,並不會經過建構子來初始化

我們可以自定義物件要如何被拆解(序列化)

自定義物件的反序列化,要注意的是順序必須要和序列化相同

Serialization and Deserialization

- 為什麼自定義的writeObject與readObject不採用override (封裝權限為private)
- ObjtectOutputStream是如何呼叫Student class當中的writeObject方法?
  - **反射(Reflection)技術**
- 為什麼不自定義序列化方法,依然能成功把物件寫到檔案裡?

Scanf in Java

- 從使用者讀取鍵盤指令時,把System.in作為一個InputStream串流即可逐行讀取

```
public static void main(String[] args) throws Exception{
    System.out.println("請輸入一個字串:");

    BufferedReader buf = new BufferedReader(
                         new InputStreamReader(System.in));

    String input = buf.readLine();

    System.out.println(input);
}
```

Stream V.S. Writer(Reader)