Programming Notes

Robust Programs

1. Introduction

A robust program is one that continues to behave reasonably even in the presence of errors. For example, say a program user is required to enter certain data, a robust program will check that what it receives is the right sort and will ask the user to re-enter it if necessary.

Some sorts of problems are not so easily remedied. In general the aim is to provide a good approximation of what should happen in the absence of error. If this is not possible then at least

- don't permit data to be corrupted,
- don't carry on regardless,
- don't lose data.

1.1 The concept of a "Failure Condition"

An error in a program gives rise to a failure condition during execution i.e. a situation that is not desired or expected. To properly characterise the idea of a "Failure Condition" we need to begin with the idea of a partial method.

A partial method is one that is not defined for all possible parameter values - it has a precondition that must be satisfied. For example, consider the following specification for a withdraw method for a BankAccount class.

```
public void withdraw(double amount)

      // Record that the given amount has been withdrawn from the
      // account. There must be sufficient available funds for this
      // operation to take place.
```

The precondition is that the amount to be withdrawn is greater than zero and less than or equal to the available funds.

A failure condition exists when an attempt is made to perform an operation in circumstances for which an outcome is not defined i.e. circumstances that do not satisfy the precondition for that operation.

Examples

- Withdrawing an amount from a bank account that is greater than available funds.
- Accessing the $(n+1)^{th}$ element of an n element array.
- Reading data from a file that has been deleted.
- Using Integer.parseInt() on a String that does not represent a number.
- Performing a method call on the null value instead of on an object.
- Dividing a number by zero.

CONTINUED

1.2 Defensive programming

How do we make a program more robust? One approach is defensive programming.

Defensive programming is the augmentation of a program with code that detects and responds to failure conditions. We can apply the principle wherever we wish. User input validation is an obvious example but it can also be applied in the construction of classes.

Applying defensive programming to class design and implementation typically involves:

- Providing accessor methods that enable calling code to avoid a provoking a failure by checking that preconditions are met.
- Modifying methods to detect and respond to failure condtions.

Consider the design of a BankAccount class and the issue of withdrawal. We might provide an accessor method for obtaining available funds and we might make a robust withdraw method such as follows:

```
public void withdraw(double amount) {
    if (amount<=0) {
        // Failure condition, take action.
    }
    else if (amount>availableFunds()) {
        // failure condition, take action.
    }
    else {
        balance = balance - amount;
    }
}
```

We've left out exact details of what the withdraw method might do if it detects a failure condition. What should it do? In fact it's probably best if it communicates the failure to the caller. Code that calls the method can be designed to recieve this communication and respond appropriately.

A component such as a BankAccount class may see use a large number of programs running on a large number of computers - cash machines, branch computers, and so on. Given the range of contexts it is not appropriate to make assumptions about what a meaningful recovery from failure might be. Better to communicate details and let the programmers writing code that uses the BankAccount class decide what to do.

In general, the place where a failure condition can (easily) be detected is often not the best place to deal with it. When developing a robust program:

- place detection code where most appropriate,
- communicate any detected failures to the place where it is best handled.
- take appropriate action in that place.

CONTINUED

1.3 Communicating failure

A traditional approach to communicating failure is to use sentinel values. For example,

```
public int withdraw(double amount) {
    if (amount≤0) {
        return -1;
     }
    else if (amount>availableFunds()) {
        return -2
    }
    else {
        balance = balance - amount;
        return 1;
    }
}
```

Here we've modified the withdraw method so that it now returns an int value. A value of -1 indicates failure because the amount is less or equal to zero, a value of -2 indicates failure because the amount exceeds the available funds, and a value of 1 indicates that the transaction was successful.

This approach seems to work well but what if the method already returns a value?

Sometimes it's still possible to use the scheme as not all values of the return type are used:

```
public static double calculateVAT(double amount) {
     if (amount<0) {
           return -1;
     else {
           return amount*1.175;
     }
}
```

Here for example, legitimate values are never negative, so returning -1 to indicate a failure condition is 'safe'.

Criticism can still be levelled, of course, because there will be methods where all possible return values are used. Also, we can criticise the scheme because the communication of failure is not sufficiently distinct from normal program behaviour. It is possible to ignore the communication. For example, a method calling calcuateVAT might fail to check that -1 is returned and assume instead that it's a legitimate value.

Java provides a better solution, an "Exception Mechanism".

2. The Java Exception Mechanism

The Java Exception Mechanism is for communication of failure conditions. The word "exception" derives from the conception of a failure condition as an exceptional situation.

- A failure condition is represented as an "exception" object.
- The condition is communicated by 'throwing' the object that represents it.
- Code can be written that receives the communication by 'catching' the exception object. This code can then carry out remedial actions.

CONTINUED

2.1 Classes of exception

There are many kinds of exceptional situation/failure condition, each may be represented by a specific class of object. For example:

`java.lang.IllegalArgumentException`

> This class of object represents the situation where a method has been called with an argument that is unsuitable.

`java.lang.IndexOutOfBoundsException`

> This class of object represents the situation where an attempt is made to access an element of an array or list that does not exist i.e. the index value used is out of range.

`java.io.FileNotFoundException`

> This class of object represents the situation where an attempt is made to open a file for reading but, as the name suggests, the file cannot be found.

`java.lang.NullPointerException`

> This class of object represents the situation where an attempt is made to perform a method on an object but the reference to that object turns out to be null i.e. an object is not actually being referred to at all.

To represent a particular failure condition, create an object of the most appropriate class.

Considering the BankAccount withdraw method, the failure conditions it detects would be well represented by java.lang.IllegalArgumentException objects:

```
public void withdraw(double amount) {
    if (amount<=0) {
        IllegalArgumentException e = new IllegalArgumentException();
        // Now to communicate the failure...
    }
    else if (amount>availableFunds()) {
        IllegalArgumentException e = new IllegalArgumentException();
        // Now to communicate the failure...
    }
    else {
        balance = balance - amount;
    }
}
```

If we wish we can have an exception object store a string that conveys more information on a failure. For example,

```
public void withdraw(double amount) {
    if (amount<=0) {
        IllegalArgumentException e =
            new IllegalArgumentException("withdraw: negative
amount");
        // Now to communicate the failure...
    }
    ....
```

CONTINUED

2.2 The throw statement

To communicate a failure condition, assuming the creation of a suitable exception object, a throw statement is used e.g.

```
public void withdraw(double amount) {
      if (amount<=0) {
                  // Attempt to withdraw a negative amount detected
            throw new IllegalArgumentException();
      }
      else if (amount>availableFunds()) {
                  // Attempt to withdraw more than available funds
detected.
            throw new IllegalArgumentException();
      }
    else {
           balance = balance - amount;
    }
}
```

There are two checks applied to the amount parameter. In either case if the check fails an IllegalArgumentException object is created to represent the failure condition and a throw statement 'throws' this back to the calling code, causing execution at that point to be abandoned.

2.3 The try-catch statement

A throw statement may be used in a method or constructor to communicate failure back to the calling or invoking code. If the calling or invoking code is a good place to respond to the failure then a try-catch statement may be used to receive the communication and specify what should be done.

A try-catch statement comprises a "try" block followed by one or more "catch" blocks.

- The try block contains the code that may throw an exception i.e. communicate a failure.
- A catch block describes what to do should a particular type of exception object be received.

For example, consider the following method that, given a BankAccount object as a parameter, asks the user to input the amount they want to withdraw (presuming a method called "inputNumber" that performs this function) and calls the "withdraw" method. This call statement is wrapped in a try-catch statement.

```
...
public void doWithdraw(BankAccount account) {
    System.out.println("Please enter amount to withdraw");
    int amount = inputNumber();
    try {
        // Try to withdraw amount requested.
        account.withdraw(amount);
        // Indicate success
        System.out.println("Transaction completed");
    }
    catch (IllegalArgumentException e) {
        // If execution has transferred here then the call
        // "withdraw(amount)" has failed.
        System.out.println("Sorry, that was an invalid sum");
    }
```

CONTINUED

```
    System.out.println("Goodbye");
}
...
```

The catch block has the formal parameter "IllegalArgumentException e". This indicates that it contains statements that should be executed should an "IllegalArgumentException" be thrown by the code in the try block.

If the user enters a valid amount then the statement "account.withdraw(amount)" will execute without failure, "Transaction completed" will be printed followed by "Goodbye". The catch clause will be skipped over.

If an invalid amount is entered then "account.withdraw(amount)" will fail - the withdraw method will throw a java.lang.IllegalArgument exception. Execution of the try block will be abandoned and the catch block executed. The IllegalArgumentException thrown by the withdraw method will be stored in the variable "e". This will result in "Sorry, that was an invalid sum" being printed followed by "Goodbye".

2.4 Failure propagation

Consider a method *A* that calls a method *B* that calls a method *C*. Assume that in method *C* a failure is detected and an exception is thrown back to method *B*. If in method *B* the call to method *C* is not surrounded by a try-catch statement (or if it is but there is no catch clause for the exception thrown) then execution of method *B* will be abandoned and the exception thrown back to method *A*.

If method *A* does not contain a suitable try-catch statement then execution of it is abandoned and the exception thrown back to whatever called it. This process continues until (i) the exception surfaces in a suitable try-catch statement, or (ii) the call chain is exhausted in which case the program terminates.

2.5 Checked and unchecked exceptions

As noted, there are different classes of exception. These classes themselves can be divided into two categories: "checked" verus "unchecked". By way of example, java.lang.IllegalArgumentException is categorised as an "unchecked" exception while java.io.FileNotFoundException is a"checked" exception (see notes on reading and writing text files). The distinction is based on the sort of failures the exceptions represent.

- Checked exceptions represent failures that a programmer should expect might occur in normal operation - such as failure to create a file because the hard drive is full.
- Unchecked exceptions represent failures that a programmer should not expect - such as an attempt to read the $(n+1)^{th}$ element of an n element array.

Given the nature of checked exceptions, if a programmer uses a method or constructor that could give rise to one then they MUST write code to handle it. Their work will not compile otherwise. This means surrounding the relevant code with a try-catch clause, or, if they wish to propagate it in the fashion described above, using a throws clause.

2.6 The throws clause

A throws clause is used in the signature of a method to indicate that (i) execution of the body of the method may result in a checked exception being thrown (ii) and the method itself will not catch it but instead propagate it to the caller.

CONTINUED

For example,

```
...
public void printFile(String fileName) throws FileNotFoundException
{

    FileReader reader = new FileReader(fileName);
    Scanner fileScan = new Scanner(reader);

    while (fileScan.hasNextLine()) {
        System.out.println(fileScan.nextLine());
    }
}
...
```

The method accepts the name of a file as parameter, and uses a FileReader and Scanner to open it and print it line by line. The checked exception "java.io.FileNotFoundException" could be thrown by the code.

If required, more than one class of checked exception may be listed in a throws clause. The general form of the clause is

```
... throws <exception class name>, ..., <exception class name>
```

END

CONTINUED