15 Lecture: Being Honest

Goals: - Integrity of data: Exceptions.

Introduction

Until now we have ignored the possibility that the data supplied to the constructor may not satisfy our constraints. So, it would be possible to construct a date with the values for the month and day being 23 and 45 — even when we know that 23rd month and 45th day in the month cannot represent any date in our calendar. We will now learn how we can design the constructor that prevents the client from instantiating an object with invalid data.

Note: We do not favor the design of programs that are overrun with clauses that validate user supplied data against the stated contracts. It is a bad programming style and results in obfuscating the essence of the code. In general, the programming language constructs should allow the programmer to state such contracts and report violations automatically.

The Problem

Let us consider the class ClockTime that represents time of day measured in hours and minutes as shown in the class diagram in figure 1.

The assertion that the value of the hour data should be from 0 to 23 and the value of the minute data should be from 0 to 59 is only given in the comments following the relevant fields. Nothing prevents the client from providing incorrect data to the constructor.

Exceptions

Java has a mechanism for signalling errors in the program at runtime. The Java Library contains a class *RuntimeException* and a program that detects a fatal error may abort by *throwing an exception*. Throwing a *RuntimeException* is fatal to the program — it stops and produces an error message.

One example of an error for which a program that will fail is an attempt to divide an integer by zero. So, including the following statement in your code

```
System.out.println("result: " + 3/0);
```

```
+---------------------------+
| ClockTime                 |
+---------------------------+
| int hour /* 24 hr clock */ |
| int minute   /* 0 to 59 */ |
+---------------------------+

class ClockTime {
  int hours;       /* 24 hr clock */
  int minutes;         /* 0 to 59 */

  ClockTime(int hours, int minutes){
    this.hours = hours;
    this.minutes = minutes;
  }
}
```

Figure 1: A class diagram for recording the time of day

will produce the following message:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Examples.main(ClockTime.java:109)
```

The *ArithmeticException* is a subclass of *RuntimeException*. Other subclasses of the *RuntimeException* worth noting are *IllegalArgumentException*, *NoSuchElementException*, *UnsupportedOperationException*, and *ClassCastException*.

Our program can specify when to throw an exception by including the statement

```
throw new XyzException("error diagnostic message");
```

at the appropriate place in the program.

So, for example, we can define the method *remove*(*Object obj*) that removes the given object from **this** list. If the given object is not in the list, the program fails — our specification dictates that this method can only be invoked when we are certain that the list does indeed contain the given object. The code for the class *MTLoObj* would then be:

```
ALoObj remove(Object obj){
  throw new RuntimeException("object to be removed not found");
}
```

Data Verification

Let us now design an error-checking constructor for the class *ClockTime*. All we need to do is assure that the hours are between 0 and 23 inclusive, and minutes are between 0 and 59 inclusive. Here is our first version:

```
// construct a new ClockTime object
//verifying the hour and minute data
ClockTime(int hours, int minutes){
  if (0 <= hours && hours < 24)
    this.hours = hours;
  else throw new IllegalArgumentException(
                  "hours are outside 0 - 23 range");

  if (0 <= minutes < 60)
    this.minutes = minutes;
  else throw new IllegalArgumentException(
                  "minutes are outside 0 - 59 range");
}
```

There is a small problem here. For now the values of our fields once initialized never change. However, soon we will see the situations where this is no longer the case. If some methods in the program, or the client of the *ClockTime* class can change the values of *hours* or *minutes* at a later time, we want to make sure that the test for validity of this data appears in the program in only one place. We can design a method that consumes the given *hours* or another one that consumes the *minutes* and produces a boolean value indicating the validity of the data.

```
// does the given number represent valid hours?
boolean validHours(int hours){
  return 0 <= hours && hours < 24;
}

// does the given number represent valid minutes?
boolean validHours(int minutes){
  return 0 <= minutes && minutes < 60;
}
```

The constructor would then change to the following:

```
// construct a new ClockTime object
//verifying the hour and minute data
ClockTime(int hours, int minutes){
  if (this.validHours(hours))
    this.hours = hours;
  else throw new IllegalArgumentException(
                  "hours are outside 0 - 23 range");

  if (this.validMinutes(minutes))
    this.minutes = minutes;
  else throw new IllegalArgumentException(
                  "minutes are outside 0 - 59 range");
}
```

There is something unusual about these methods. The purpose statement does not include the word **this**. The method invocation within the constructor uses **this** to invoke the method, but at that point we do not even have a complete **this** object. To indicate that the method is really a function that can be invoked without any reference to the instance of the class where it is defined, we define the method as *static*. We also make it *private* — as there is no reason it should be invoked by any of the clients of this class.

Note: We do not advocate using static methods. They are rarely needed and in most cases they indicate poor programming style. We have been using *static* methods in the class *Math* — those that compute the sqrt, the trigonometric functions, and other such numerical calculations.

The complete class definition with examples follows:

```
public class ClockTime {

  int hours;
  int minutes;

  // data verifying constructor
  public ClockTime(int hours, int minutes){
    if (this.validHours(hours))
      this.hours = hours;
    else
      throw new RuntimeException("Invalid hours");

    if (this.validMinutes(minutes))
      this.minutes = minutes;
    else throw new RuntimeException("Invalid minutes");
  }
```

```
// constructor that defines minutes by default
public ClockTime(int hours){
  this(hours, 0);
}


private static boolean validHours(int hours){
  return 0 <= hours && hours < 24;
}

private static boolean validMinutes(int minutes){
  return 0 <= minutes && minutes < 60;
}

// to verify the data represents a correct time
public boolean validTime(int hours, int minutes){
  return this.validHours(hours)
  && this.validMinutes(minutes);
}

// to represent this data as a string
public String asString(){
  return new String("class ClockTime \n" +
      "  hours = " + this.hours + "\n" +
      "  minutes = " + this.minutes + "\n");
}

public static void main(String argv[]){

  ClockTime ct1 = new ClockTime(23, 38);
  ClockTime ct2 = new ClockTime(3);
  //ClockTime badHours = new ClockTime(25, 30);
  //ClockTime badMinutes = new ClockTime(22, 70);


  System.out.println(ct1.asString());
  System.out.println(ct2.asString());

  boolean testValidHours1 = ct1.validHours(23) == true;
  boolean testValidHours2 = ct1.validHours(27) == false;

  boolean testValidMinutes1 = ct1.validMinutes(38) == true;
  boolean testValidMinutes2 = ct1.validMinutes(70) == false;

  boolean testValidTime1 = ct1.validTime(12, 38) == true;
  boolean testValidTime2 = ct1.validTime(12, 70) == false;
  boolean testValidTime3 = ct1.validTime(27, 38) == false;
  boolean testValidTime4 = ct1.validTime(27, 70) == false;
```

```
    System.out.println(
        "testValidHours1: " + testValidHours1 + "\n" +
        "testValidHours2: " + testValidHours2 + "\n" +
        "testValidMinutes1: " + testValidMinutes1 + "\n" +
        "testValidMinutes2: " + testValidMinutes2 + "\n" +
        "testValidTime1: " + testValidTime1 + "\n" +
        "testValidTime1: " + testValidTime1 + "\n" +
        "testValidTime2: " + testValidTime2 + "\n" +
        "testValidTime3: " + testValidTime3 + "\n" +
        "testValidTime4: " + testValidTime4);
  }
}
```

For the benefit of the clients we also added a method that validates the entire time data - both the hours and minutes in one method.