COMP241
Software Engineering Development
Lecture 6: Designing Classes 2

Mark Hall

Readings: Horstman Chap 3

- Programming by Contract (adapted from slides by Mark Utting)
  - Preconditions
  - Postconditions
  - Class invariants
- Static Methods
- Static Fields

THE UNIVERSITY OF WAIKATO *DEPARTMENT OF COMPUTER SCIENCE*
TARI RO ROHIKO

Programming by Contract

- An agreement between two parties:
  - The ***provider*** provides a service/product.
  - The ***client*** uses that service/product
    - **(and usually pays for it!).**
- Each side has obligations (and benefits):
  - *Precondition* = client's obligations.
  - *Postcondition* = provider's obligations

Programming by Contract: Motivation

- Want precise specifications of classes
  - Class will be clearly documented
  - Makes implementers job easier
  - Callers (clients) know precisely how/when to call each method and what it will do
  - Computer can check correctness of class and its clients. Automatic bug catching!

Programming by Contract: Preconditions and Postconditions

- A *precondition* describes the properties that must hold whenever a procedure is called.
- A *postcondition* describes the properties that the procedure guarantees when it returns.

Contract: *If you promise to call me with Pre satisfied, then I promise to return with Post satisfied.*

Preconditions and Postconditions

Preconditions are typically stated for one of two reasons:

1. To restrict the parameters of a method
2. To require that a method is called when it is in the appropriate *state*

There are two kinds of postcondition:

1. That the return value is computed correctly
2. That the object is in a certain state after the method call is completed

Preconditions and Postconditions

- Example: `deposit` method in `BankAccount`
  - **Precondition:** the amount to be deposited should not be negative
  - **Postcondition:** the account balance will always be ≥ 0
- In a *correct* system, procedures are only called when their preconditions are true.
- If a procedure is called when its precondition is false, it may do anything!
  (The contract is broken!)

Preconditions and Postconditions

- Therefore, preconditions and postconditions are important parts of a method, and they should be documented
- Method documentation for `deposit`:

```
/**
 * Deposits money into this account
 * (Postcondition: getBalance() >= 0)
 * @param amount the amount of money to deposit
 * (Precondition: amount >= 0)
 */
```


Preconditions and Postconditions

- Note that we formulate pre- and postconditions only in terms of the *interface* of the class
- Method documentation for `withdraw`:

```
/**
 * Withdraws money from the bank account
 * @param amount the amount of money to withdraw
 * (Precondition: amount >= 0)
 * (Precondition: amount <= getBalance())
 */
```

- Thus, `amount <= getBalance()`,
  not `amount <= mBalance`
  - Caller needs to check the precondition and only has access to the public interface, not the private implementation

More examples

- Stack: `public Object pop()`
- Stack: `public void push(Object val)`

Obligations and Benefits

	Obligations	**Benefits**
Client:	Only call Pop on non-empty stack.	Value returned was on top of the stack. (will be removed)
Stack:	Must return top of stack and shrink stack	No need to handle cases where stack is empty.

Checking Preconditions

- What should a method do when it is called with inappropriate inputs?
  Eg. `account.deposit(-1000)`

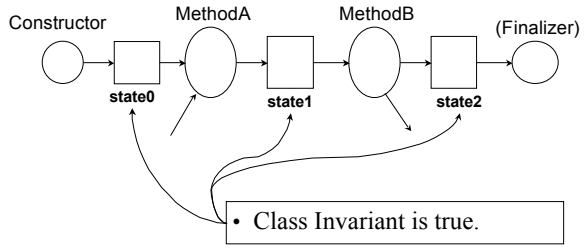
Two choices:

1. The method can check for the violation and throw an exception
2. If it is too cumbersome or expensive to carry out the check then just assume that preconditions are fulfilled
   - Any data corruption or failures are the callers fault

Class Invariants

- Pre/Postconditions only describe the properties of individual methods
- We also want to document properties of the whole class. Things which are common to its methods
- A ***class invariant*** expresses consistency constraints that apply to all objects of that class, all through their lifecycle

When is class invariant true?



Class Invariant Rules

- Constructors must *establish* C.Inv.
- Methods must *maintain* C.Inv.
  - However, within the method, the class invariant may be temporarily broken, while data structures are being updated.
- Finalizer can *assume* C.Inv.
- Subclasses should only *strengthen* C.Inv.

Checking Class Invariants

- Define an *invariant* method in each class
  - `public/protected void invariant()`
- Call it at the end of each constructor, and at the end of each method that modifies the object.
  - Alternatively, call from unit tests
- Advantages:
  - Clear documentation of class data structures
  - Catches corrupt data errors ASAP
  - Subclasses can refine `invariant()` adding their own additional checks (*strengthen*)

Invariant Example

```
public class Stack extends Vector {
   . . .
   public void invariant() {
      super.invariant();
      if (size() < 0) {
         throw new Exception(. . .);
      }
      if (!(empty() == (size() == 0))) {
         throw new Exception(. . .);
      }
      if (!empty()) {
          if (top() != elementAt(size() - 1) {
             throw new Exception(. . .);
          }
      }
   }
}
```

Pretend, for the sake of example, that this method exists in the Vector class

Static Methods

- So far we've looked a instance methods that all have an *implicit* parameter
  - *Recall that an implicit parameter is a reference to the object containing the method. It can be accessed by the method, if necessary, by using the* `this` *keyword*

```
BankAccount myAccount = new BankAccount();
myAccount.deposit(100);
```

myAccount is the object on which the deposit method is invoked. It is the ***implicit*** parameter to the deposit method

Static Methods

- Sometimes you write methods that don't need an implicit parameter
  - Eg. the `sqrt` method in the `Math` class is a *static* method that doesn't require an implicit parameter
  - Also, every application has a static `main` method

```
double result = Math.sqrt(x);
```

"Math" is the name of the class, not an object

Static Methods

- Why would you want to write a method without an implicit parameter?
  - Common reason: compute something that involves only (primitive) numbers

    Since numbers aren't objects, you can't pass them as implicit parameters

Static Methods

- Eg. simple static method that calculates whether two floating point numbers are approximately equal:

$$\frac{|x - y|}{\max(|x|, |y|)} \leq \varepsilon$$

- Where ε is a small number, eg. 10^{-14}
- However, if *x* or *y* is zero, then you need to test if the absolute value of the other quantity is at most ε

Static Methods

- This computation is just complex enough to warrant encapsulating it in a method
  - Since the parameters are numbers, the method doesn't operate on any objects at all, so we make it into a `static` method

```
public static boolean approxEqual(double x, double y) {
   final double EPSILON = 1E-14;
   if (x == 0) { return Math.abs(y) <= EPSILON; }
   if (y == 0) { return Math.abs(x) <= EPSILON; }
   return Math.abs(x - y) /
      Math.max(Math.abs(x), Math.abs(y)) <= EPSILON
}
```

Static Methods

- We need to find a home for this method
- There are two choices:
  1. Add this method to a class whose other methods need to call it
  2. Come up with a new class (similar to the `Math` class of the standard Java library) to contain it
- Number 2 is nice from the *cohesive* point of view as we end up with general purpose *Utility* class full of related static methods

Static Methods

```
public class Numeric {
   public static boolean approxEqual(double x, double y) {
      . . .
   }
   // More numeric methods can be added here
}
```

- When calling a static method, you supply the name of the class containing the method so that the compiler can find it

```
   double r = Math.sqrt(2);
   if (Numeric.approxEqual(r * r, 2)) {
      System.out.println("Math.sqrt(2) squared is " +
         "approximately 2");
   }
```

Static Methods

- Note that you do not supply an object of type `Numeric` when you call the method—static methods have no implicit parameter
- The `main` method is static because when a program starts, there aren't any objects yet

  Therefore, the first method in the program must be a static method
- In general, minimize the use of static methods
  - If you are using a lot of static methods then you may not have found the right classes to solve your problem in an object-oriented way

Static Fields

- Consider a slight variation of the `BankAccount` class:
  - Has both a balance and an *account number*
- We want to assign account numbers sequentially
  - Constructor should construct first `BankAccount` with number 1, the next with number 2, etc
  - Therefore, we must store the last assigned account number somewhere

Static Fields

- Can't make this into an instance field—in that case each instance of `BankAccount` would have its own value of the last assigned account number
- Need to have a single value that is the same for the entire *class*
- Such a field is called a class field, or `static` field

```
public class BankAccount {
   . . .
   private double mBalance;
   private int mAccountNumber;
   private static int sLastAssignedNumber;
}
```


Static Fields

- Every method of a class can access its static fields

```
public class BankAccount {
   . . .
   public BankAccount() {
      // generate next account number to be assigned
      sLastAssignedNumber++; // incr the static field

      // assign to account number of this bank account
      mAccountNumber = slastAssignedNumber;
      // set instance fields . . .
   }
}
```


Static Fields

- How do we initialize static fields?
  - Can't initialize them in the constructor—they would get reset every time a new object is created
  - Could do nothing. The static field is then initialized with 0 (for numbers), `false` (for boolean) or `null` (for objects)
  - Use an explicit initializer:

```
public class BankAccount {
   . . .
   private static sLastNumberAssigned = 0;
}
```

  - The initialization is executed once when the class is first loaded into the virtual machine

Static Fields

- Like static methods, be careful with the use of static fields
  - Methods that modify static fields have side effects—their behaviour does not simply depend on their inputs
  - Don't use static fields as a mechanism to transfer values between methods

Summary

- Preconditions, postconditions and class invariants help in documenting code and designing tests
- Static methods are often found in *utility classes*, but shouldn't be abused
- Similarly, be sparing in the use of static fields