Data Structures

Subodh Kumar

Dept of Computer Sc. & Engg.
IIT Delhi

OOP Quiz 2

Class `Bird` extends class `Animal`.

```
class Cat {
   public Bird catchBird(Bird birdToCatch) {
      ...
   }
    … Later in a method …

    Animal pigeon = new Bird();
    catchBird(pigeon);
}
```

Type mismatch: compiler

Is there a problem?

OOP Quiz 3

In class Animal:

```
public void eat(Food food) {
    food.cook();
    food = new Food();
    food.digest();
}
```

Somewhere else in the program:

```
myAnimal.eat(apple);// apple is of type Fruit
                    // Fruit is a subclass of Food
```

What happens to object `apple`?

1) Nothing; only a copy of **`apple`** gets passed as a parameter, original is safe
2) It gets cooked, but not digested
3) It gets cooked and digested
4) It gets digested, but not cooked
5) After **`eat`**, **`apple`** refers to the same instance of the **`Fruit`** class as it did before we called **`eat()`**
6) After **`eat`** returns, **`apple`** refers to a different instance of the **`Fruit`** class

Cooked. Some other "Food" object is digested.

apple in caller still refers to the same object

OOP Quiz 4

True/False?

F 1. If class B extends A, which has a method named m: any method by name m declared in B must have the same signature as the one in A.

T 2. Formal parameters have the same scope as local variables (of the method).

T 3. An actual parameter may be an instance of a subclass of the type declared in the formal parameter.

F 4. An instance of a subclass may be substituted for an instance of its superclass in an actual parameter just as an instance of a superclass may serve as the substitute for an instance of its subclass in the same situation.

```
class C1 {
    C1()          { System.out.println("C1"); }
    C1(int n)     { System.out.println("C1 : " + n); }
    void print() { System.out.println("C1 print"); }
}
class C2 extends C1 {
    C2()          { System.out.println("C2"); }
    C2(int n)     { this(); System.out.println("C2 : " + n); }
    void print() { System.out.println("C2 print"); }
}
class MainClass  {
    public static void main(String[] args) {
        C1[] c1 = new C1[4];
        c1[0] = new C2();
        c1[1] = new C2(5);
        c1[2] = new C1();
        c1[3] = new C1(4);
        for (int i=0; i<4; i++) {
            c1[i].print();
        }
    }
}
```

DYI. Run it to verify.

What prints?

Lucid Programming Style

Good choice	Poor choice
`Professor`	`myClass`
`teachSections4to8`	`operate`
`col106Professor`	`x`
`numSections`	`number, count`

- One class per
- Indent: 2-3 spa
- Use names de
  - Do not reuse variables
- Capitalize parts
  - Begin class names with capital letters
  - Begin method, local and instance variable names with small letters
  - Constants in all caps, use "_" to separate parts
  - Begin package names with small letters

Example Conventions

- Write short methods
- Declare at the beginning
  - of method, block, loop
  - at least be consistent
- No literals (except for -1, 0, and 1 in loops)
- Qualify names fully
  - Access static methods by class name
- Use parentheses liberally
- Declare public only when necessary
- Declare all “fields” then “methods” in a class
  - Constructor first, Overloads together
- Initialize basic type fields at declaration

Testing

- Test, then test some more
  - Test in unit environment
  - Test full application
- Maintain regression suite
  - Directed tests for hard/corner cases
    - Ensure code coverage
  - Random test generation
- Use svn, regress check-ins
- Assertion checks (See `assert expr1:expr2`)
- Validate input

Be Mindful of Common Errors

- Array index out of bounds
- File not found
- Disk full
- Permission denied
- Type conversion
- Null reference
- Network down
- Divide by 0

Exceptions are your friend

- Helps you handle run-time errors
  - And think about types of errors that are possible
- Method raises exception to indicate "unexpected" condition
  - Implicit "return flags"
- Catch Expected exception and handle it
  - Separates error handling in a modular way
  - Eases "passing the buck" to caller
- Unhandled exception terminates the program
  - Program should handle user defined exceptions

Unchecked Exception

```
public  class  Propagate {
   void  divide() {
      int  m = 25,  i = 0;
      i = m / i;
   }
   void  process() {
      divide();
   }
   public static void main(String[] args) {
      Propagate p = new Propagate();
      p.process();
   }
}
```

Exception

```
java.lang.ArithmeticException:      / by zero
        at Propagate.orange(Propagate.java:4)
        at Propagate.apple(Propagate.java:8)
        at Propagate.main(Propagate.java:11)
```

JAVA default handler

Exception Handling

```
try {
    … normal program code
}

catch(Exception e) {
    … exception handling code
}

finally { // optional: execute after try
}
```

Exception Handling

```
try {
    … normal program code
}

catch(SomeExceptionClass e) {
    … exception handling code
}

catch(SomeExceptionClass e) {
    … exception handling code
}

finally { // this is optional
}
```

System Exception

```
public void aMethod() {
    try {
        int a[] = new int[2];
        a[2] = 1;
    } catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println(
                "exception: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Pass Exception Along

- A method that might encounter an unhandled exception, should use “throws” clause:

```
public void myMethod throws IOException
{
    … normal code with some I/O
}
```

- It can generate exception as well

Implementation

- Exception are objects
  - Instances of Throwable class, or one derived from it
- You should makes new class
  - extending JAVA Exception class
  - class Exception extends Throwable

```
class MyException extends Exception {
    …
}
```

Example

```
class MyException extends Exception  {}

class MyClass {
   void someMethod() throws Myexception {
        MyException x = new MyException();
        // ... some code here
        if (val < 1) throw x;
        }
}
```

Example

```
class MyException extends Exception  {
   MyException(String s) { super(s); }
}
…

void someMethod throws MyException {
   MyException x = new MyException("Message");
   ...
   if (val < 1) throw x;
}
```

Program Correctness

- Starts with a correct specification of the requirements
  - Correctness only with respect to the spec
    - If program A sends data D to program B, B will eventually get D
    - There is no deadlock or livelock
    - For all $n \geq 0$, $f(n) = k$
- Correct design
  - Algorithmically correct
- Correct implementation
  - Re-use of already correct code helps

Programming

- Understand specification
  - Formulate as formally as you can
- Devise the test plan
- Algorithmic design
  - Mind the performance
- Refine the test plan
- Implement incrementally
  - Test each time
  - Error debugging + performance debugging

Proving Correctness

- What is correct?
  - Program? Or entire running enviroment?
  - System
- What is a proof?
  - Not: *system meets requirement*
  - But: *system cannot fail requirement*
  - Challenging

Testing is Important

Bill Gates:

> **"When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development. We have as many testers as we have developers. Testers basically test all the time, and developers basically are involved in the testing process about half the time…**
>
> **… We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing…**
>
> **…The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one."**

Proof?

Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence"

"One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes'"

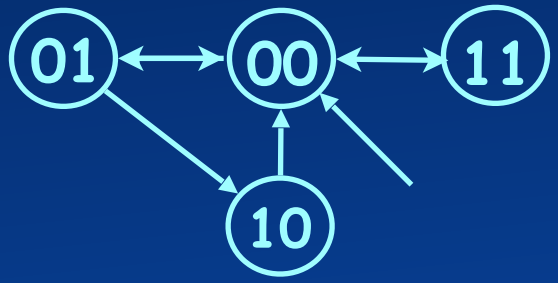
- Automated proof?
  - Impossible in the general case
- Mechanical verification is possible in some cases

Formal Specification

- Precise syntax and semantics
  - Another language
  - Maybe, “programming” in this language is simpler
  - It only needs to specify behavior
- Specification of the system is not:
  - specification of some of its properties
- Specification could be given as
  - actual program
  - state machine
  - set of logical expressions

Example Verification Methods

- Invariants
- Precondition ⇒ Postcondition
- Condition X is impossible
- Condition Y is true in every execution
- Condition Z is always eventually true
- Value of a variable is *f*(..) at time > *t0*
- Given state machine, show <P> satisfied



<P>: NEVER IN STATE 10