# CS 535 Object-Oriented Programming & Design
Fall Semester, 2008
Doc 12 Comments, Class Invariants, etc.
Oct 14 2008

Copyright ©, All rights reserved. 2008 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.

# References

Class invariants, http://c2.com/cgi/wiki?CodeClassInvariants

The benefits of object-oriented programming using class invariants, http://www.stanford.edu/~pgbovine/programming-with-rep-invariants.htm

Data Type, http://en.wikipedia.org/wiki/Data_type

# Magic Numbers

```
(each asInteger > 96 & (each asInteger < 123))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



What is the code trying to do here? What is 96 and 123 in ascii?

# Less Magic Numbers

```
"96 = $` asInteger, 123 = ${ asInteger"

(each asInteger > 96 & (each asInteger < 123))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



Well now we know that the numbers are but still it is not clear what the code is doing.

# Less Magic Numbers

```
(each asInteger > $` asInteger & (each asInteger < ${ asInteger))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



We have the same amount of information but don't have to read a comment.

# Less Magic Numbers

```
"check to see if each is between a and z"

(each asInteger > $` asInteger & (each asInteger < ${ asInteger))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



Now we know what the code claims to be doing, but not clear it is really doing it.

# Less Magic Numbers

```
"check to see if each is between a and z"

(each asInteger >= $a asInteger & (each asInteger <= $z asInteger))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



Now we don't need the comment. It just gets in our way and slows us down.

# Less Magic Numbers

```
"check to see if each is between a and z"

(each asInteger >= $a asInteger & (each asInteger <= $z asInteger))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]


"is each a lowercase character"

(each asInteger >= $a asInteger & (each asInteger <= $z asInteger))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



Why repeat your code in the comments? If we can see what you are doing by looking at the source code then your source code is too complex. What you code does not tell us is why you are doing things.

# Less Magic Numbers

```
(each asInteger >= $a asInteger & (each asInteger <= $z asInteger))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```



Does this need a comment?

# Less Magic Numbers

```
(each >= $a & (each  <= $z))
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]


Character>>>= aCharacter
    ^self asInteger >= aCharacter asInteger

Character>><= aCharacter
    ^self asInteger <= aCharacter asInteger
```



By adding methods to the Character class the code becomes cleaner. We really don't care about the integer value of characters, so why are we dealing with them. The top line is in the String class.

# Less Magic Numbers

```
(each isAlphabetic)
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```

```
Character>> isAlphabetic
    blah
```



The previous slide was concerned about how to answer the question: is this character in the alphabet. Here we leave the how to the Character class. The intent is clear for the test. Now about the assignment statement.

# Less Magic Numbers

```
(each isAlphabetic)
    ifTrue: [ sum := sum + each asInteger\\ 32]
    ifFalse: [ sum := sum + 0]
```

"Assign each character in the alphabet a number equal to its location in the alphabet (ie a = 1, b=2). Add that number to the sum if the character is a letter in the alphabet, otherwise add zero to the sum"



The previous slide was concerned about how to answer the question: is this character in the alphabet. Here we leave the how to the Character class. The intent is clear for the test. Now about the assignment statement.

# Less Magic Numbers

```
(each isAlphabetic)
    ifTrue: [ sum := sum + each alphabeticIndex]
```



Does this need a comment?

# Less Magic Numbers

```
sum := sum + each alphabeticIndex
```



if alphabeticIndex is defined correctly we can get rid of the if statement.

If your code is too complex to follow try simplifying it before adding comments

Don't repeat your code in the comments

- Tell us the why not the how in comments

# Comments

```
String>>dollarWords
    "Returns those words in self whose alphabetic value is 100"

    |words count alphabet newWords |
    alphabet = 'abcdefghijklmnopqrtsuvwxyz'.

    "break up by words"
    words := self words.

    "Build a new collection of words"
    newWords := OrderedCollection new.
    words do: [:word |
        "Count the letter values"
        count := 0.
        word do: [:char | count := count + ( alphabet indexOf: char asLowercase)).].
        "If this is a dollarword, add it to the list"
        (count = 100) ifTrue: [newWords add: word].
        ].
    ^newWords
```

# Comments

```
String>>dollarWords
    "Returns those words in self whose alphabetic value is 100"

    |words alphabet dollarWords |
    alphabet = 'abcdefghijklmnopqrtsuvwxyz'.

    words := self words.
    dollarWords := OrderedCollection new.

    words do: [:word | | count |
        "Count the letter values"
        count := 0.
        word do: [:char | count := count + ( alphabet indexOf: char asLowercase)).].
        "If this is a dollarword, add it to the list"
        (count = 100) ifTrue: [dollarWords add: word].
        ].
    ^ dollarWords
```

# Comments

```
String>>dollarWords
    "Returns those words in self whose alphabetic value is 100"

    |words alphabet dollarWords |
    alphabet = 'abcdefghijklmnopqrtsuvwxyz'.

    words := self words.
    dollarWords := OrderedCollection new.

    words do: [:word | | letterValues |
        "Count the letter values"
        letterValues := word sumLetterValues
        (letterValues isDollarWord) ifTrue: [dollarWords add: word].
        ].
    ^ dollarWords
```

# Comments

```
String>>dollarWords
    "Returns those words in self whose alphabetic value is 100"

    |words alphabet dollarWords |
    alphabet = 'abcdefghijklmnopqrtsuvwxyz'.

    words := self words.
    dollarWords := OrderedCollection new.

    words do: [:word | | letterValues |
        letterValues := word sumLetterValues
        (letterValues isDollarWord) ifTrue: [dollarWords add: word].
        ].
    ^ dollarWords
```

When you feel the need to comment a block of code

- Consider making the block of code a separate method

# "1 to: x size do:" Verses "x do:"

```
String>>dollarWords
    | words size collection |
    words := self words.
    collection := OrderedCollection new.
    1 to: words size do: [:n |
        | word |
        word := words at: n.
        word sumValue = 100 ifTrue: [collection add: word]].
        ^collection
```



We don't need the index of each word. We want each word. Some languages make us clutter our code with details that are not needed and we get used to it. Many times we just want the element in a collection and have no need for the index. More and more languages support constructs that allow us to iterate over the collection without having to deal with the indexing.

# "1 to: x size do:" Verses "x do:"

```
String>>dollarWords
    | words size collection |
    words := self words.
    collection := OrderedCollection new.
    words do: [:word |
        word sumValue = 100 ifTrue: [collection add: word]].
        ^collection
```



Less detail to obscure what is going on in the code. But still we are doing a lot of work. This is a common pattern so why do we have to repeat it?

# "1 to: x size do:" Verses "x do:"

```
String>>dollarWords
    | words |
    words := self words.
    ^words select: [:word | word sumValue = 100].
```



We are doing the same thing, but there is much less unnecessary code detail to obscure our intent.

# Class Invariants

“Class invariants are predicates of (statements about) a class that should always be true”

John Farrell, http://c2.com/cgi/wiki?CodeClassInvariants

Examples

An instance variable is not nil
An instance variable is an ordered collection
An integer value has to be in a certain range

# Stack

Instance variables: elements, top

elements – Array containing between 0 and N elements of the stack

0 <= top <= N,

points to element that is currently the top of the stack

```
Stack>>isEmpty
  ^top = 0
```

```
Stack>>isFull
  ^top = elements size
```

```
Stack>>pop
  self isEmpty ifTrue: [invoke your empty stack policy].
  topElement := elements at: top.
  top := top – 1.
  ^topElement
```

```
Stack>>push: anObject
  self isFull ifTrue: [invoke your full stack policy].
  elements at: (top := top + 1) put: anObject.
```

# Class Invariants should hold

After an instance is created

Before and after calling any publicly accessible method

# Uses of Class Invariants

Helps prevent bugs
Helps understand a class
Help determine private methods

# Preventing Bugs -Child Example

Child class with instance variables

birthdate

age

socialSecurityNumber

(0 <= age < 18) and

(birthdate + age == todays_date) and

isLegalSSN(social_security_number)



This example is from Philip J. Guo at http://www.stanford.edu/~pgbovine/programming-with-rep-invariants.htm

# Preventing Bugs - Child Example

```
Child>>checkInvariants
    self assert: 0 <= age;
    self assert: age <= 18;
    self assert: birthdate + age = Date today;
    self assert: socialSecurityNumber isLegalSSN

Child>>ssn: aSSN
    self checkInvariants.
    socialSecurityNumber := aSSN
    self checkInvariants
```



Create a checkInvariants method that throws an exception if any invariant does not hold. The above example requires adding an assert: method. Call the checkInvariants method at the beginning and end of each publicly accessible method.

# Determine private methods

Private methods

- methods that start or end with the class invariants not holding

# Understanding Classes

ReadStream

collection <SequenceableCollection> elements to read

position <Integer> pointer to the current access position

readLimit <Integer> size of the collection

writeLimit <Integer> farthest that has been written into the collection

policy <StreamPolicy> policy for choosing the print format for various entities, such as Dates, Times, currencies, or other context-sensitive information

# What is a Subclass?

```
WordStream>>next
    "Returns next word in stream"
```

Verses

```
Stream>>nextToken: separators
    "Return all characters up to next
    element in separators"

Stream>>nextWord
    ^self nextToken: Characters wordSeparators
```

# Types, Classes & Inheritance

Data type

Attribute of a datum which tells something about the kind of datum it is.

This involves setting constraints on the datum

What values it can take and

What operations may be performed upon it.

# Types, Classes & Inheritance

Class

Template for instances (objects)

This involves setting constraints on the instance

What values it can take and

What operations may be performed upon it.

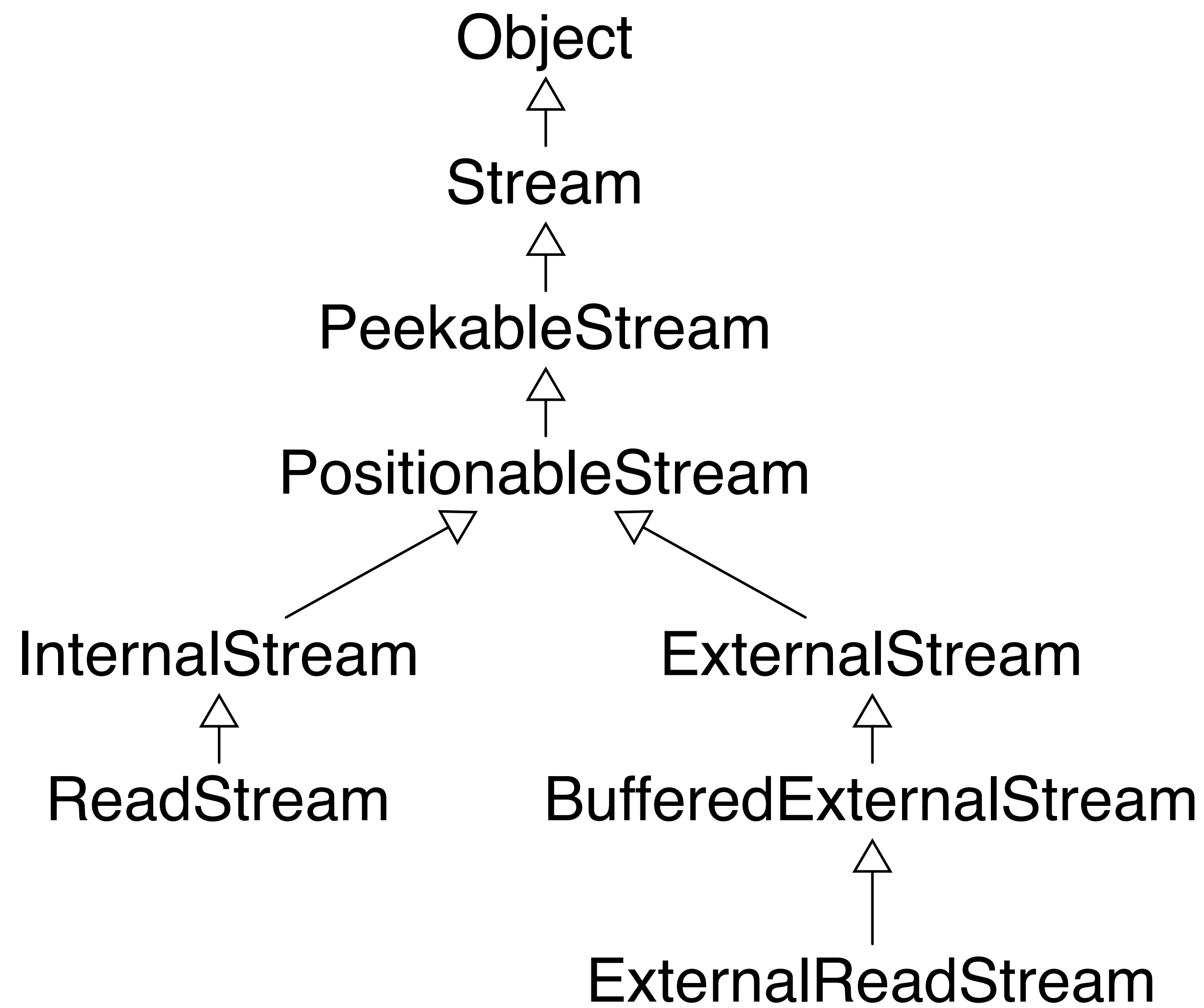
# Types, Classes & Inheritance

In some languages

- A class defines a type
- A subclass defines a subtype

But not all OO languages equate class with type

# Types, Classes & Inheritance




'foo' asFilename readStream class

results in

ExternalReadStream