CS 535 Object-Oriented Programming & Design
Fall Semester, 2010
Doc 11 Abstract Classes & Abstractions
Oct 11 200810

References

Object-Oriented Design Heuristics, Riel

Ralph Johnson Lecture notes, Lecture 3 Data Abstraction and Encapsulation, http://st-www.cs.uiuc.edu/users/cs497/lectures.html

Inheritance

What should I use as a super class?

A has a B

Indicates that an instance variable of A is an instance of B

A is a B

A is a type of B

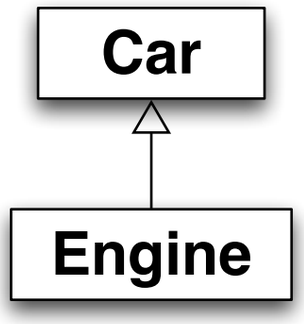
Indicates that A is a subclass of B

A car has an engine, so car object contains an engine object

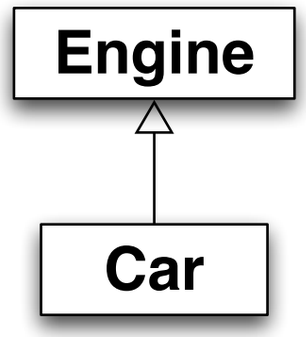
A BinarySearchTree has nodes, so it has instance variables left and right

A WordStream is a type of ReadStream so it is a subclass of ReadStream

Common Mistakes



Using has-a relation for inheritance

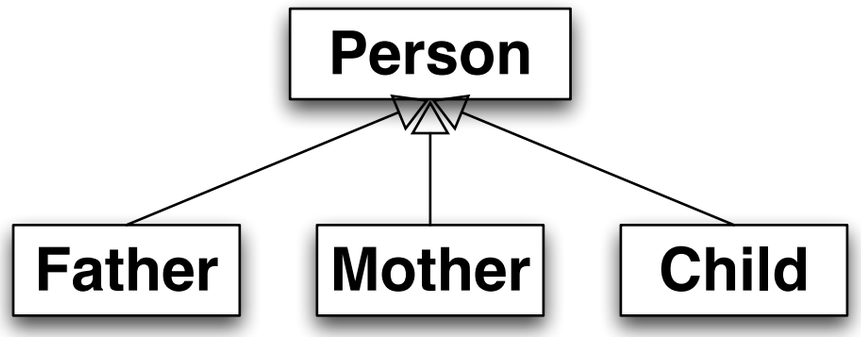


“I need access to engine methods
in the car class and now I have it.”

Roles Verses Classes

2.11 Be sure the abstractions you model are classes and not simply the roles objects play

```
mother := Mother new.
father := Father new
```




```
mother := Person new.
father := Person new.
```

Abstract Classes

Abstract class
A class that can not be instantiated

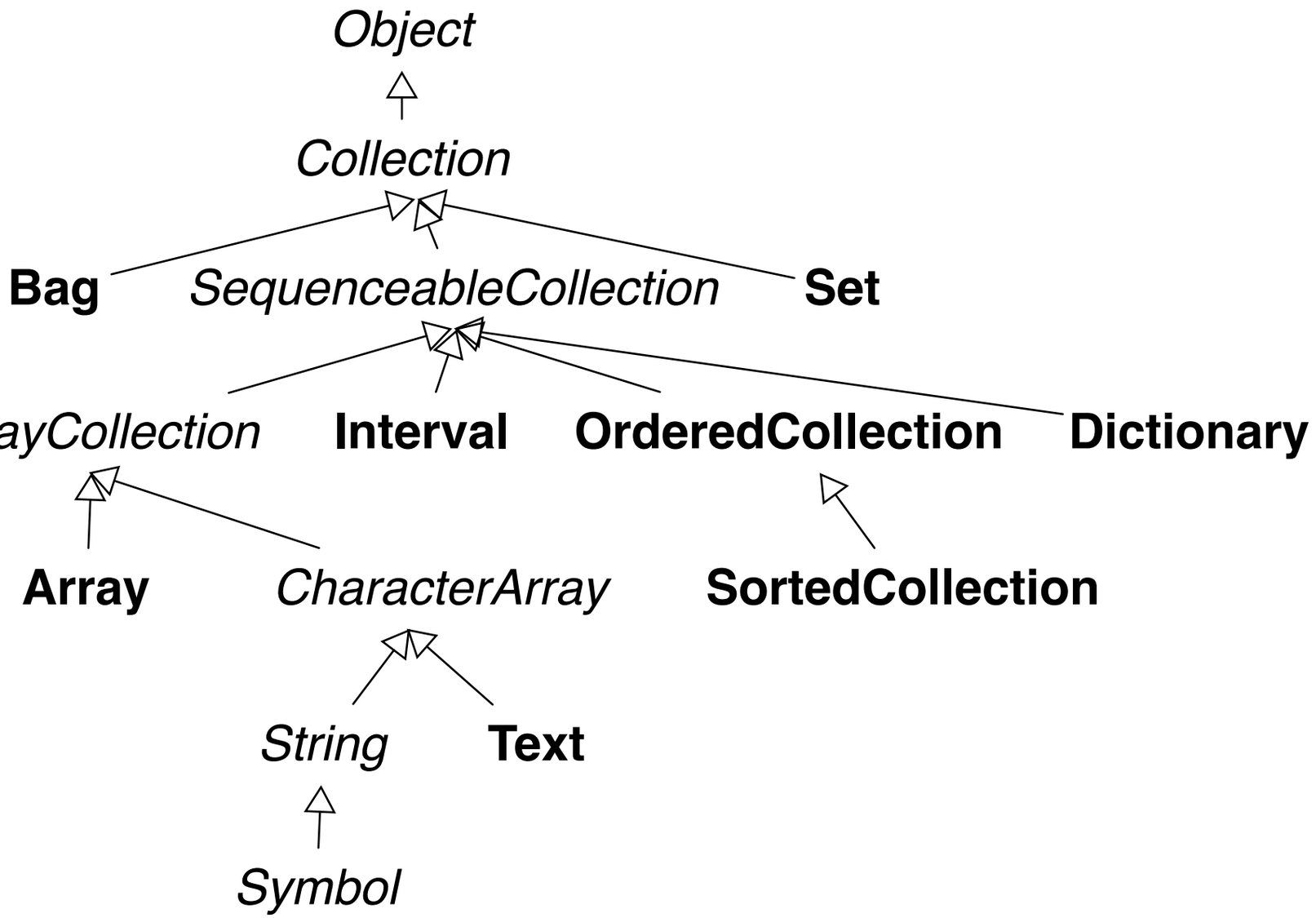
Concrete class
A class that can be instantiated

Why Abstract Classes

Define an abstraction
Define a type
Define interface for subclasses
Define methods for subclasses

Hide the existence of concrete subclasses

Smalltalk Collections



Italic - Abstract Class

Bold - Concrete class

Collection Class

No instance variables

60 methods

Three abstract methods

add:

remove:ifAbsent:

do:

Use three abstract methods to implement other 57 methods

```
detect: aBlock ifNone: exceptionBlock
	"Evaluate aBlock with each of the receiver's elements as the argument.
	Answer the first element for which aBlock evaluates to true."

	self do: [:each | (aBlock value: each) ifTrue: [^each]].
	^exceptionBlock value
```

Defining Abstract Classes

Some languages have special syntax

```
public abstract  class  NoObjects  {
   public  void  aFunction()  {
      System.out.println(  "Hi Mom"  );
   }
   public  abstract  void  subClassMustImplement(  int  foo  );
}
```

Defining Abstract Classes - Smalltalk

Mark methods as abstract with “self subclassResponsibility”

```
Collection>>do: aBlock
    self subclassResponsibility
```

Indicate class is abstract in class comment

Include list of abstract methods

Browser will create methods stubs in subclass

What does self subclassResponsibility do?

Informs reader

- Method is abstract
- Concrete subclasses need to implement the method

Raises an exception when executed to indicate

- Subclass did not implement an abstract method
- Created an instance of an abstract class

Informs browser which methods subclasses need to implement

How to Prohibit Instances of Abstract Class

Documentation is normally enough

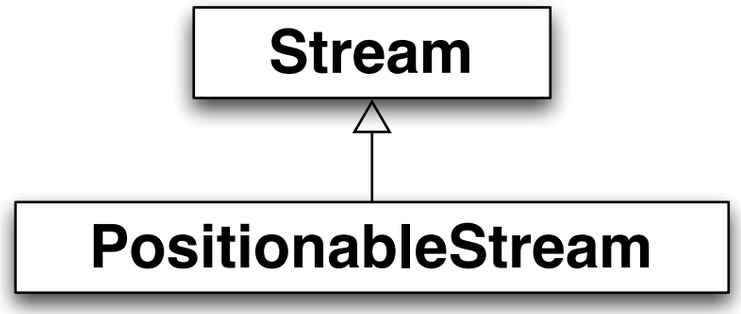
Implement new so it throws an exception

```
Stream class>>new
    "Provide an error notification that Streams are not created
    using this message."
    self error: ('Streams are created with on: and with:')
```

How do subclass objects get created?

```
Stream class>>new
    self error: ('Streams are created with on: and with:')


PositionableStream class>>on: aCollection
    ^super new on: aCollection
```



What happens when this is done?

```
PositionableStream on: String new
```

How do subclass objects get created?

Use basicNew

PositionableStream class>>on: aCollection
  ^self basicNew on: aCollection

basicNew

- Does the same thing as new
- Is used to get around super class's new method
- Only used in class instance creation methods
- Never implement basicNew

Abstract Classes and Data

Abstract classes commonly do not have instance variables

How can they implement methods?

- Identify a core set of abstract operations
- Implement other methods using core methods

Collection Class

No instance variables

60 methods

Three abstract methods

add:

remove:ifAbsent:

do:

Use three abstract methods to implement other 57 methods

```
detect: aBlock ifNone: exceptionBlock
	"Evaluate aBlock with each of the receiver's elements as the argument.
	Answer the first element for which aBlock evaluates to true."

	self do: [:each | (aBlock value: each) ifTrue: [^each]].
	^exceptionBlock value
```

Abstract Classes, Types and Hinges

Declaring a variable to be an Abstract class instance

Indicates which operations are allowed on the variable

Allows any subclass to be used in the variable

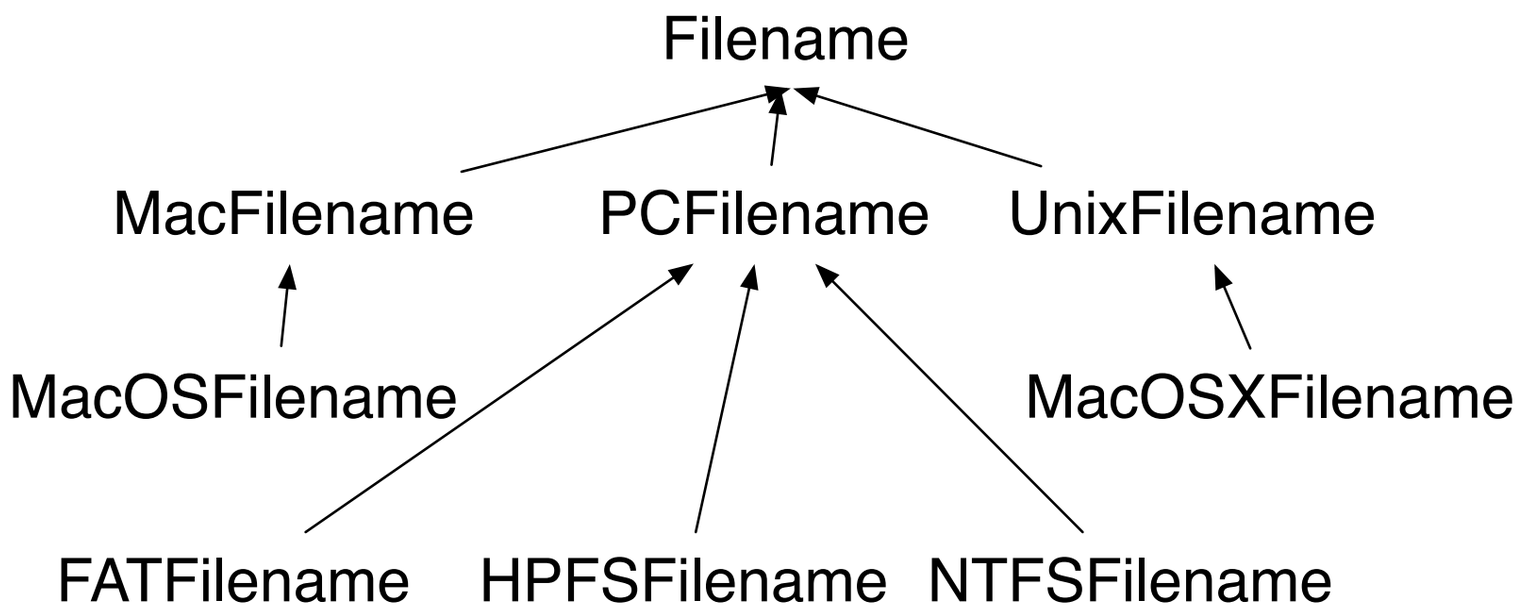
Provides flexibility particularly in languages with static type checking

```
SomeClass>>foo: aCollection
   ^aCollection fold: [:a :b | a max: b].

public class SomeClass {
   public int foo(Collection a) { blah}
}

public class Resticted {
   public int foo(Array a) { blah}
}
```

Abstract Classes and Hiding Subclasses



Smalltalk VM on startup informs Filename of the correct concrete class for the current platform

```
file := 'foo' asFilename.
file class                    "MacOSXFilename (on my machine)"
```

How it works

```
file := 'foo' asFilename.
```

```
String>>asFilename
	^Filename named: self string
```

```
Filename Class>>named: str
	str isEmpty
		ifTrue:	[OSErrorHolder invalidArgumentsSignal raiseWith: str].
	^self concreteClass createInstanceNamed: str
```

```
Filename Class>>concreteClass
	self == Filename ifTrue: [^self defaultClass].
	^self
```

```
Filename Class>>defaultClass
	^DefaultClass
```

```
Filename Class>>defaultClass: cls
	"Assign the appropriate concrete subclass for this platform.
	Only done at start-up."

	DefaultClass := cls
```

Platform Independence Aside

Mac, PC and Unix have different end of line characters

When you read a file:

- Smalltalk converts the platform's end of line character to cr

When you write a file

- Smalltalk converts cr to the platform's end of line character

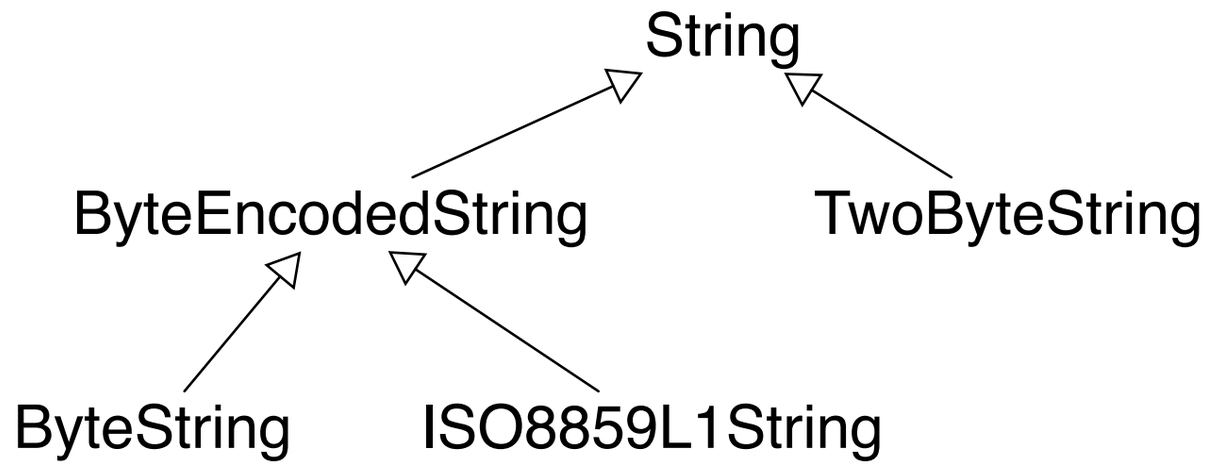
Same code

- Works on all three platforms
- Produces files with the correct end of line character

Hide the existence of concrete subclasses

String is an abstract class

String new
- Does not create a string object
- Creates an instance of a subclass
- Appears to create a String object



String subclasses
- Don’t add new methods
- Provide specific implementations

Strings Continued

```
| a |
a :=String new.
a class.                "returns ByteString"

| b |
b :=(String with: (Character value: 3585)) "3585 is a Thai character".
b class                 "returns TwoByteString"

| c |
c := String with: $a.
c class.                “returns ByteString”
c at: 1 put:  (3585 asCharacter).
c class                 “returns TwoByteString”
```


To learn about character encodings read: http://www.joelonsoftware.com/articles/Unicode.html

become: Smalltalk Magic

```
| c |
c := String with: $a.
c class.                              “returns ByteString”
c at: 1 put:  (Character value: 3585).
c class                               “returns TwoByteString”
```

How did c change class?

a become: b

Change all references to ‘a’ to reference ‘b’

Change all references to ‘b’ to reference ‘a’

‘a’ basically becomes ‘b’ and ‘b’ becomes ‘a’

String ClassTransformation without become?

Use composition

String has instance variable that holds real string

String forwards messages to the real string

String can replace the real string with a different object

Sample Implementation

```
Smalltalk.Core defineClass: #String
   superclass: #{Core.CharacterArray}
   instanceVariableNames: 'realString'

size
   ^realString size

at: anInteger
   ^realString at: anInteger

at: anInteger put: aCharacter
   aCharacter value > 256
      ifTrue: [realString := realString asTwoByteString].
   realString at: anInteger put: aCharacter.
```

Abstraction
Information Hiding
Polymorphism

Information Hiding

An object should hide design decisions from its users

Hide

What is stored & what is computed

Classes used

How does Point story its data?

How does OrderedCollection hold elements?

Heuristic 2.1

All data should be hidden within it class

Smalltalk instance variables in can be accessed in:

Instance methods of Class where they are defined

Instance methods of subclasses of the Class where they are defined

Language Support for Global Data

Smalltalk has shared variables

Use sparingly

Use for constants

What is a constant?

Hiding Instance Variables

Some argue that only two methods should access an instance variable

```
Class BankAccount
Instance variable: balance

balance
   ^balance

balance: aNumber
   balance := anumber

deposit: aNumber
   self balance: (self balance + aNumber)
```

Why

This protects the class from changes in instance variables

Makes easy to enforce constraints

```
balance: aNumber
   aNumber < 0 ifTrue: [ NegativeBalanceError raiseSignal].
   balance := aNumber
```

Hiding Instance Variables &Tools

Refactoring browser

- Lists all methods accessing an instance variable
- Change all accesses to be through access methods
- Removes all access through access methods

So don’t worry about hiding instance variables

If later you need to hide them it is easy to do

Abstraction

“Extracting the essential details about an item or group of items, while ignoring the unessential details.”

Edward Berard

“The process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use.”

Richard Gabriel

Pattern: Priority queue

Essential Details: length

items in queue

operations to add/remove/find item

Variation: link list vs. array implementation

stack, queue

How to Find Abstractions

Look at nouns in requirements specification or system description

A refrigerator has a motor, a temperature sensor, a light and a door. The motor turns on and off primarily as prescribed by the temperature sensor. However, the motor stops when the door is opened. The motor restarts when the door is closed if the temperature is too high. The light is turned on when the door opens and is turned off when the door is closed.

Look at these phrases. Some will be obvious classes, some will be obvious nonsense, and some will fall between obvious and nonsense. Skip the nonsense, keep the rest. The goal is a list of candidate objects. Some items in the list will be eliminated, others will be added later. Finding good objects is a skill, like finding a good functional decomposition

Ralph Johnson's Suggestions for Finding Abstractions

Do one thing
Eliminate duplication
Keep rate of change similar
Decrease coupling, increase cohesion
Minimize interfaces
Minimize size of abstractions
Minimize number of abstractions

Do One Thing

Methods should do one thing

Method's name should tell what it does

findString:startingAt:
asNumber
asUppercase
dropFinalVowels

Class should be what its name says

String
OrderedCollection
Array
ReadStream

Break complex classes/methods into simpler ones

Eliminate Duplication

(self asInteger - $a asInteger + anInteger) \\ 26 – (self asInteger - $a asInteger)

(self alphabetValue + anInteger) \\ 26 - self alphabetValue.

Keep rate of change similar

An object should not contain both

- An instance variable that changes every second
- An instance variable that changes once a month

- Code that is different for each hardware platform
- Code that is different for each OS

Separate tax tables from employee data from time cards

Minimize interfaces

Use the smallest interface you can

Use Number instead of Float

Avoid embedding classes in names

add: instead of addNumber:

Minimize the size of abstractions

Lots of Little Pieces

Methods should be small

Median size is 3 lines
10 lines is starting to smell

Classes should be small

7 variables is starting to smell
40 methods is starting to smell

VW 7.6

	Average	Median	Max
Variables / class	2.1	1	72
Methods / class	16.6	8	359
LOC / method	3.0	2	156

Code used to generate Numbers

Variables Per Class

```
classes :=Smalltalk allClasses reject: [:each | each isMeta]
variablesInClass :=classes collect: [:each | each instVarNames size].
average :=((variablesInClass fold: [:sum :each | sum + each] )/
                variablesInClass size) asFloat.
median := variablesInClass asSortedCollection at: variablesInClass size // 2.
max := variablesInClass fold: [:partialMax :each | partialMax max: each]
```

Methods Per Class

```
classes :=Smalltalk allClasses reject: [:each | each isMeta]
methodsInClass :=classes collect: [:each | each selectors size].
average :=((methodsInClass fold: [:sum :each | sum + each] )/
                methodsInClass size) asFloat.
mean := methodsInClass asSortedCollection at: methodsInClass size // 2.
max := methodsInClass fold: [:partialMax :each | partialMax max: each]
```


Note how the above code could use the application of these ideas

LOC / Method

```
methodSizes := OrderedCollection new.
classes
	do: [:class |
		class selectors
			do: [:method |
				| periodCount |
				periodCount := (class compiledMethodAt: method) decompiledSource
					occurrencesOf: $..
				methodSizes add: periodCount + 1]].
average :=((methodSizes fold: [:sum :each | sum + each] )/
			methodSizes size) asFloat.
median := methodSizes asSortedCollection at: methodSizes size // 2.
max := methodSizes fold: [:partialMax :each | partialMax max: each]
```

Minimize number of abstractions

A class hierarchy 6-7 levels deep is hard to learn

Break large system into subsystems, so people only have to learn part of the system at a time