Comp-304 : Object-Oriented Design
What does it mean to be Object Oriented?

What does it mean to be OO?

- What are the characteristics of Object Oriented programs (later: OO design) ?
- What does Object Oriented programming add

  (as opposed to structure programming?)

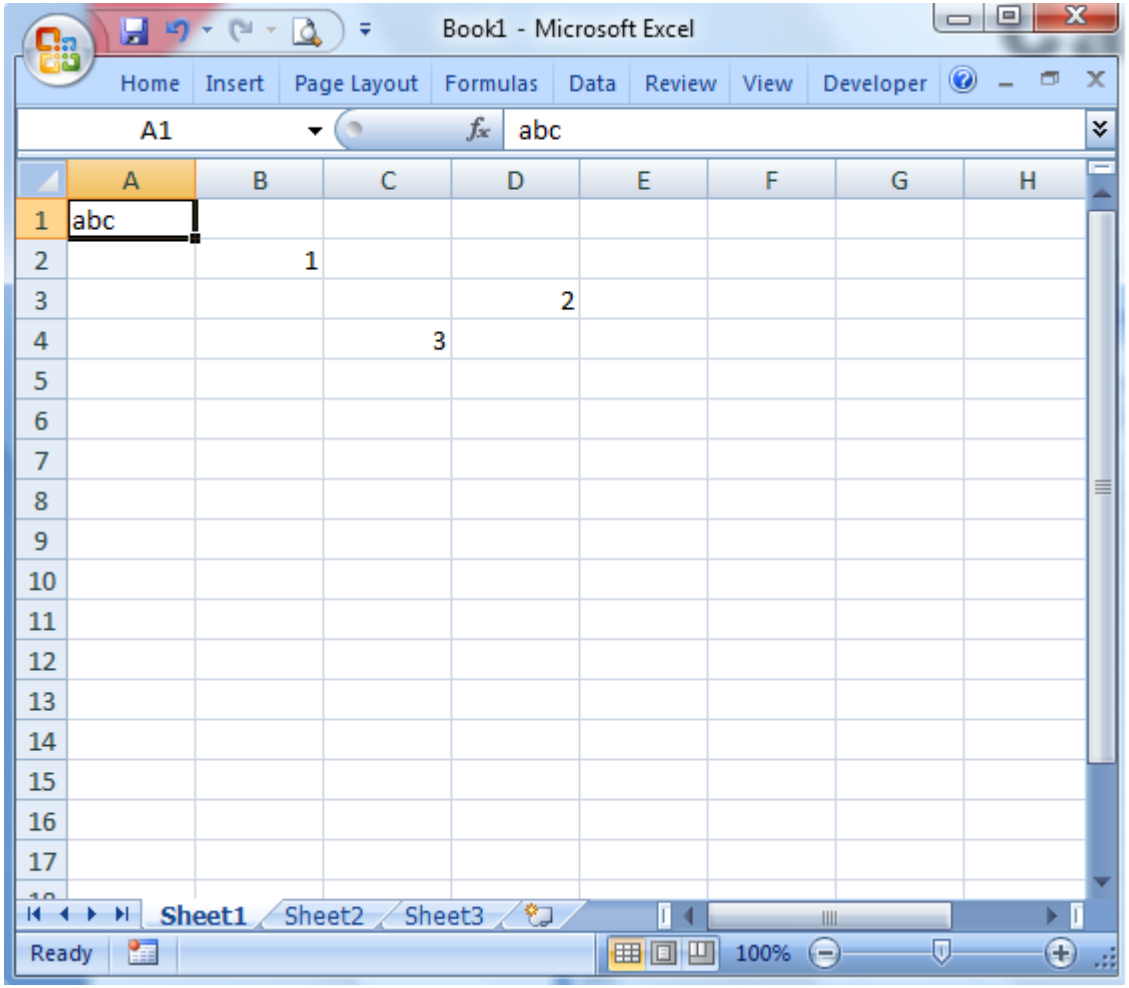
What does it mean to be OO?

1) Encapsulation
2) State Retention
3) Implementation / Information Hiding
4) Object Identity
5) Messages
6) Classes
7) Inheritance
8) Polymorphism
9) Genericity

Object Structured, Based, Oriented

- Exhibit 1 – 3, called object – structured
- Exhibit 1 – 4, called object – based
- Exhibit 1 – 7, called class – based
- Exhibit 1 – 9, called object – oriented

Case Study: Spread Sheet



Abstracted: Spread Sheet Grid

abc						
	1					
			2			
		3				

Let's design

- What classes do we need?
- What attributes should they have?
- What methods should they have?
- How should classes be related to other classes?

~ CRC (Class Responsibility Collaboration)

Encapsulation

- Definition: grouping of related concepts into a single unit referred to by a single name
- Different levels of encapsulation:
  - Level 0 : within a **line of code**
  - Level 1 : multiple lines of code, **procedure**
  - Level 2 : set of procedures, **class**
  - Level 3a : set of classes of the same domain,
    - Horizontal, **package** like or
  - Level 3b : set of classes of different domains performing a common job
    - vertical, **component** like

So what is OO encapsulation?

- Object – Oriented (referred to as OO hereafter) encapsulation is the grouping of methods and attributes representing state, into an object so that the state is accessible and modifiable via the interface provided by the encapsulation

Encapsulation in Spread Sheet

- Level 1 : Lines of code -> Actions
  - Change cell value
  - Add row
- Level 2 : Actions + State -> Key objects :
  - Cell referencing
  - Create table
  - Etc
- Level 3 : Cell objects -> Active range
- ... Level 4 ?

State retention

- The attributes of an object represent what it remembers.
- The state of an object is the set of current (“snapshot”) values for those attributes.
- If an attribute changes, so does the state of the object.
  - An object composed of 4 booleans has 16 possible states.
  - An object composed of 2 integers has 18 446 744 073 709 551 616 possible states.
- State of an object may differ before and after a method call.
  - objects don't die after “execution”.

State in SpreadSheet

- How many states can a cell have?
- A cell has the following attributes:
  - Coordinate : row + column
  - Value : string, float

SpreadSheet

- How do you store the position of a cell?
  - (1..n, 1..m)

Information / Implementation hiding

- When observing an encapsulated entity, we can have two points of view:
  - From the outside (public view)
  - From the inside (private view)
- The advantages of a good encapsulation is the separation of the private and public views.
- To access elements in the private view, users must go through the public interface.
  - Use of encapsulation to restrict internal workings of software from external user view

Cell: SpreadSheetData

- How do I store the location of a cell?
  - An integer ? N x M possible values (say 100 x 26)
    - Row then column: 26 + 26 + 23 = 75 (C23)
    - Column then row: 100 x 23 + 3 = 2303
  - An integer couple ? (1,1), (3, 23)
  - A character + integer ? “A1”, “C23”
  - A hask-key ?
- How do I hide this from the user?
  - IsOnRow(i) : boolean
  - IsOnColumn(i) : boolean
  - GetRow() : int
  - GetColumn() : int

Information vs. Implementation

Information Hiding

- We restrict user from seeing information
  - variables, attributes, data, etc.
- To access information, users must use a set of public methods.

Implementation Hiding

- We restrict user from seeing implementation
  - code, operations, methods, etc.
- Users can use the method without knowledge of how it works.

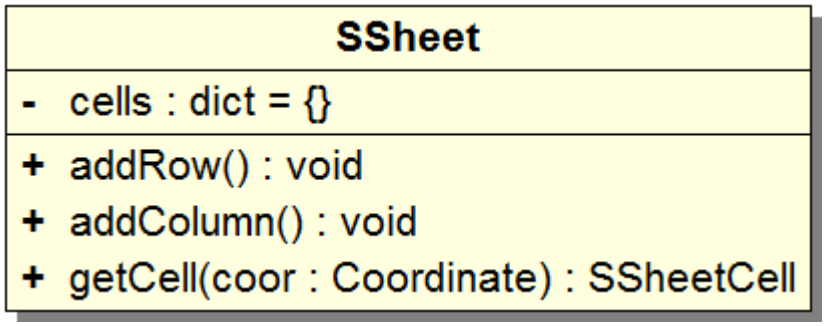
Why should we do this?

- Designer and user must agree on some interface and nothing else. They are independent. They do not need to speak the same language
- Software evolution is easier. Suppose user knows about implementation and relies on it. Later, if the designer changes the implementation, the software will break
- Code re-use is easier
- Abstraction from user is high, user need not worry about how it works!

Get / Set Rule

- Never allow other class to directly access your attribute.
- Once an attribute is public, it can never be changed.
  - Ex: img.pixeldData
- Make your attributes available using get/set methods.
  - self.row Bad!
  - self.getRow() Good!

Point



- Inside, cells could be using dictionnary or 2D-array.
  - Dictionnaries are more efficient when dealing with sparse sheets.
  - 2D-arrays are more efficient when dealing with dense sheets.

Network Engine Example

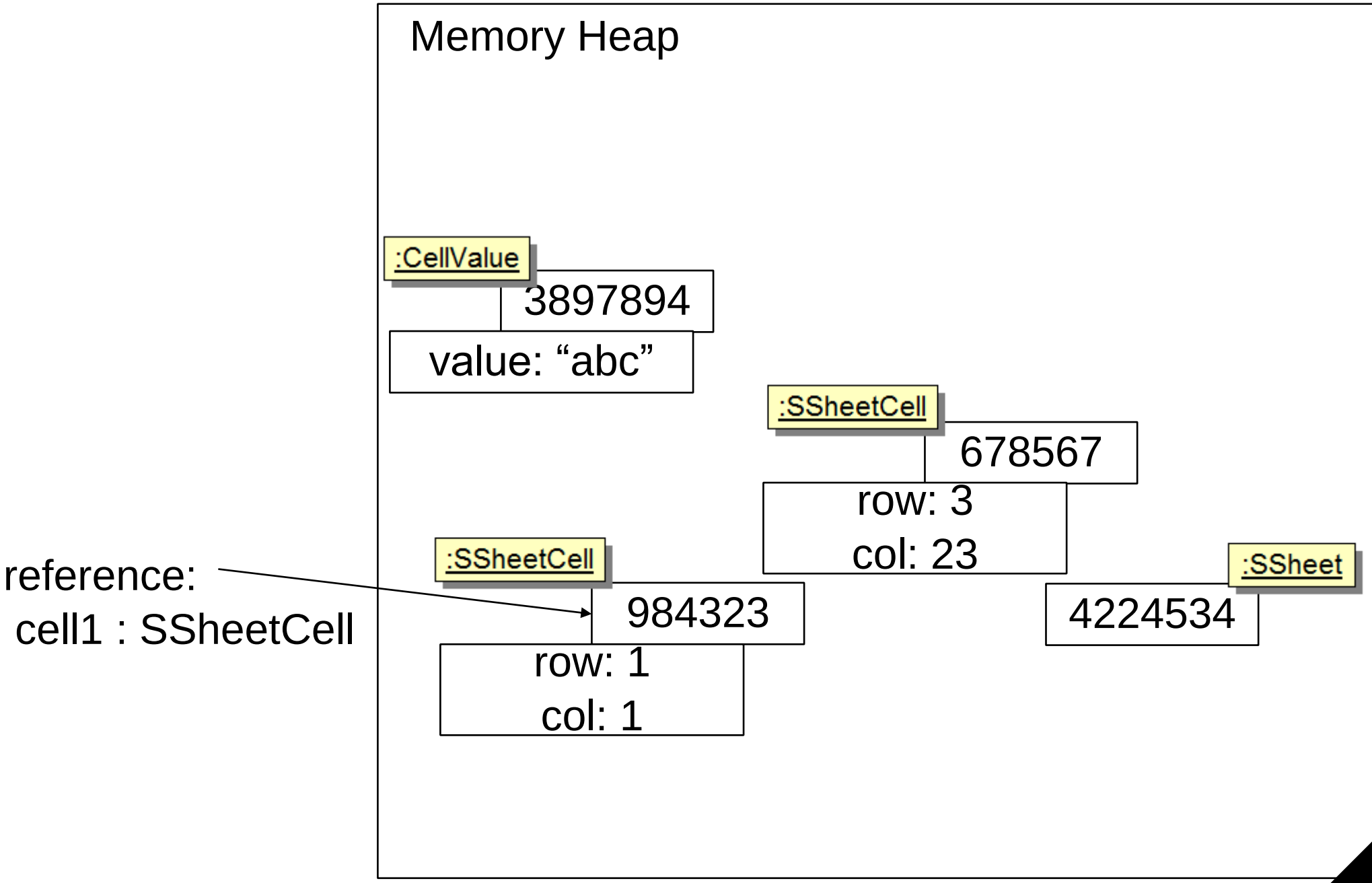
```
public interface NetworkClient {
    public connect(String address);
    public void send(Object obj);
    public Object receive();
    public void close();
}
```

- This kind of network interface can be implemented using multiple protocols.
- The user doesn't even need to know which underlying protocol is used.

Object Identity

- Each object can be identified and treated as a distinct entity.
- Use **unique** names, labels, handles, references and / or object identifiers to distinguish objects. This unique identifier remains with the object for its **whole life.**
- We cannot use objects' states to distinguish objects, since two distinct objects may have the same state (i.e., same attribute values).

Distinct Identity



Mutable vs Immutable Objects

- An Immutable object is an object that is created once and is never changed.
  - Type: String, Long, tuple, etc.
  - Two Immutable objects are considered the same if they have the same state.
- A Mutable object is an object whose state can change.
  - Vector, Array, etc.
  - Two different Mutable objects are never considered the same (different identity).

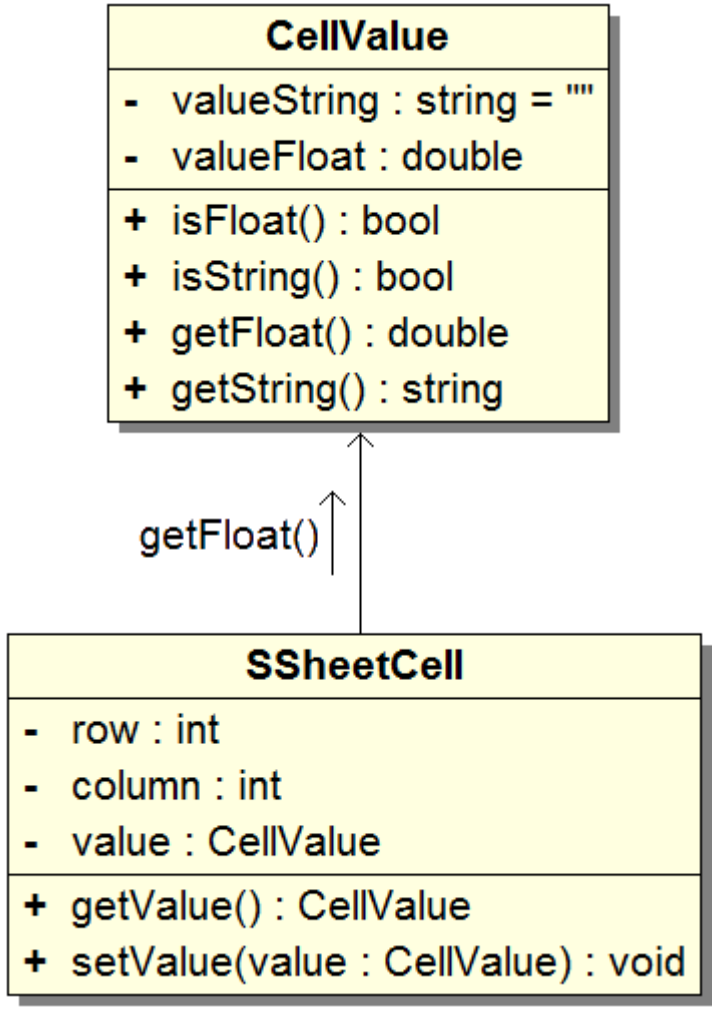
What does it mean to be Object Oriented?

1) Encapsulated
2) State Retention
3) Implementation / Information Hiding
4) Object Identity
5) Messages
6) Classes
7) Inheritance
8) Polymorphism
9) Generacity

Messages (Calls)

- *Sender* object (o1) uses messages to demand *target* object (o2) to apply one of o2's methods
- For o1 to send a meaningful message to o2, it must adhere to some *message structure*
  - o1 must know o2's unique identifier
  - o1 must know name and signature of o2's method it wants to call
  - o1 must supply any arguments to o2 so that the method may execute properly
- i.e. in Java, we write o2.method(args)

Messages (Calls) (cont.)



Messages (Calls) (cont.)

- In “pre-OO” language, we might have written method(o2, args). Note: Python's “syntactic sugar”
- This doesn't allow polymorphism!
- For o1's message to properly execute o2's method, o1 must
  - know the signature of o2's method
  - pass the proper arguments (inputs)
  - know if the method will return any values (outputs) and be ready to store them accordingly

Types of Messages

- Three types of messages:
  - **Informative**: supplies target object with information to update its attribute(s)
    [Past-oriented: Forward, Push]
  - **Interrogative**: asks target object to supply information about it's attribute(s)
    [Present-oriented: Backward, Pull]
  - **Imperative**: tells target object to do some action
    [Future-oriented: Force]

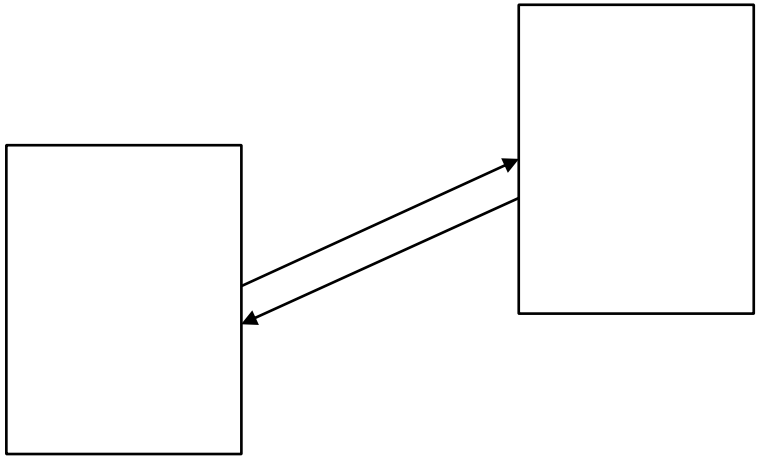
Informative, Interrogative or Imperative ?

- cell1.getValue()
- cellVal.isFloat() ?
- ssheet.insertRow(5)
- cell2.setValue(5)
- ssheet.addRow()
- cell.isEmpty() ?
- cellVal.computeFormula()

Synchronous vs Asynchronous

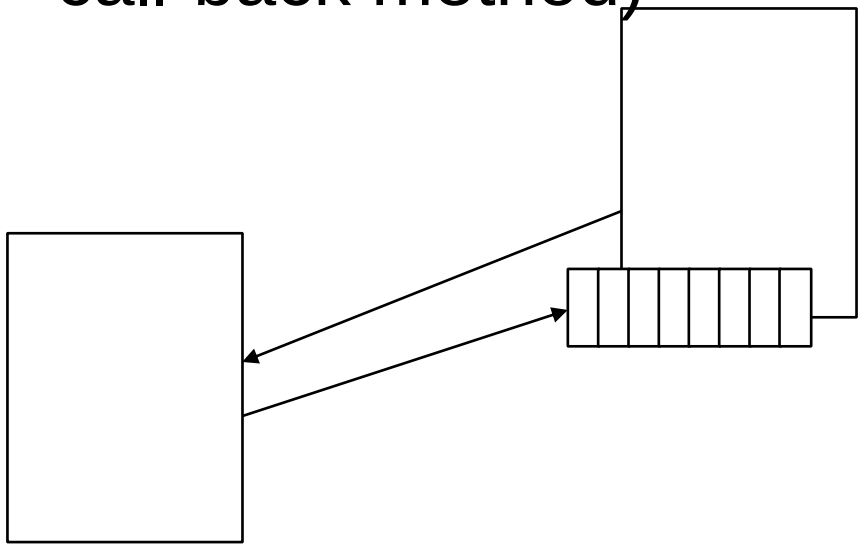
Synchronous Messaging

- An object receiving a request executes it immediately and returns the result. 1 thread of ctrl.



Asynchronous Messaging

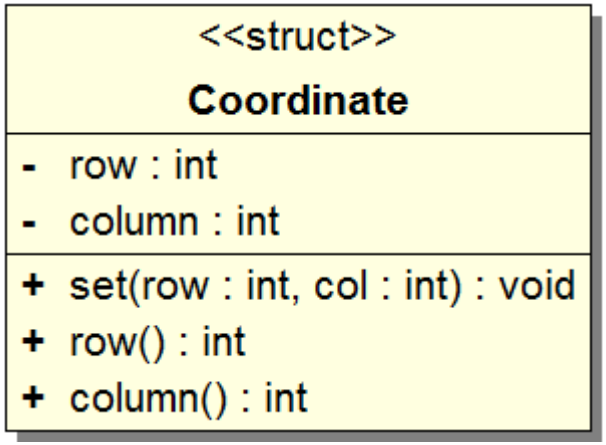
- A object receiving a request acknowledges it.
- The request is executed later and the return value is eventually returned (often through the use of a call-back method)



Classes

- A class is the stencil from which objects are created (instantiated).
- Each object has the same structure and behaviour as the class from which it is instantiated.
  - same attributes (same name and types)
  - same methods (same name and signature)
- If object **obj** belongs to class C (intention: classification)
  - then **obj** is an instance of C.
- So, how do we tell objects apart?
  - Object Identity

Instantiation



Classes vs Objects

- Classes are static and are evaluated at compile time.
  - Only one copy of the class exist.
  - Memory to store methods is only allocated once.
- Objects are dynamic and are created at run time.
  - One copy of the object is created every time the object is instantiated
  - Thus, memory to store the attributes (“state”) is allocated for every instantiated object.

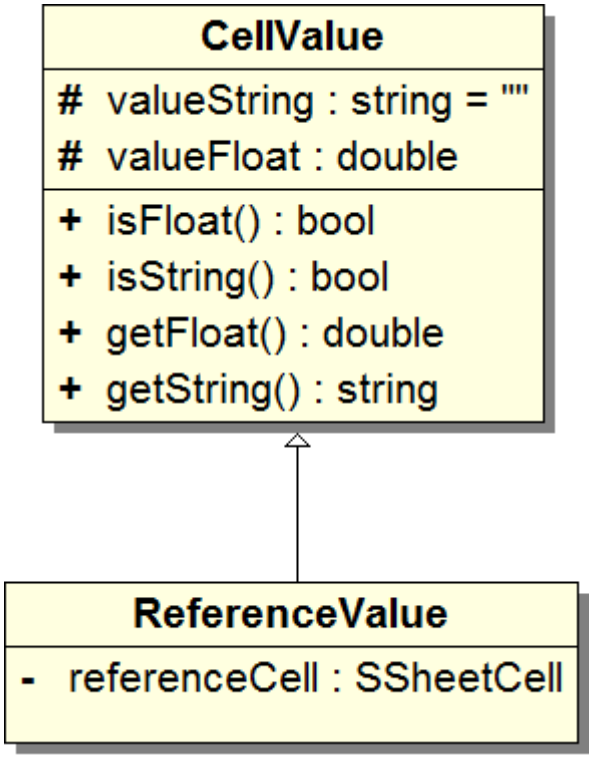
Inheritance

- Suppose you have classes C1 and C2. At design time, you notice that everything in C1 (attributes and methods) should also be in C2, plus some extra attributes/methods.
- Instead of rewriting all of C1's code into C2, we let C2 inherit from C1.
- Thus, C2 has defined on itself (implicitly) all the attributes and methods of C1, as if the attributes and methods had been defined in C2 itself.

Relationship

- Inheritance should be an “is a” relationship
- Suppose we have a class MotorVehicle
  - An Automobile is a MotorVehicle
  - A Motorcycle is a MotorVehicle
- We call MotorVehicle the superclass and Automobile is a subclass
  - MotorVehicle is more general(ized)
  - Automobile is more specialized

Specialization



Multiple Inheritance

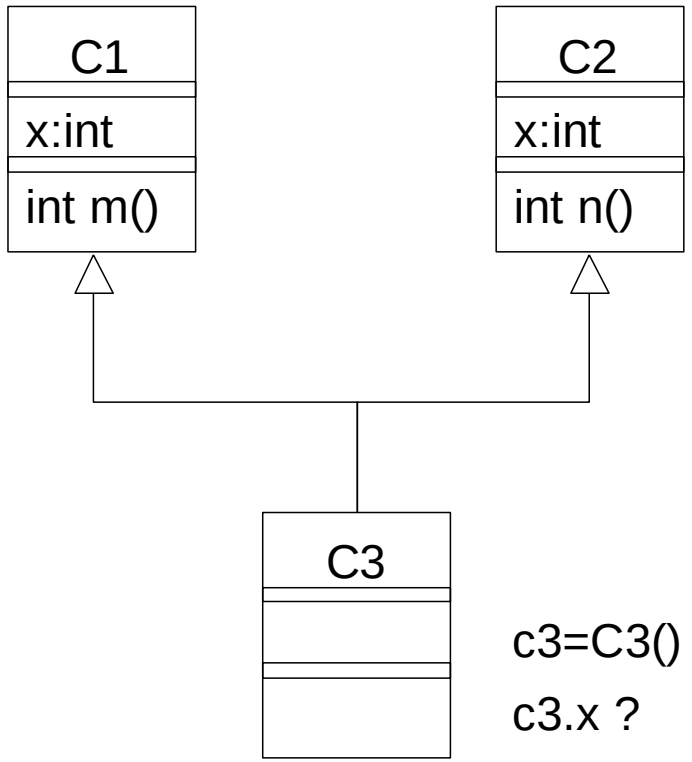
- Many classes can inherit from one class
- One class can inherit from many classes
  - Why is this good ?
  - Why is this bad?

The Good

- Allows code reuse
  - code in superclasses doesn't have to be rewritten in subclasses
- Ease of maintenance
  - if we add an attribute to a superclass, all subclasses will automatically inherit it

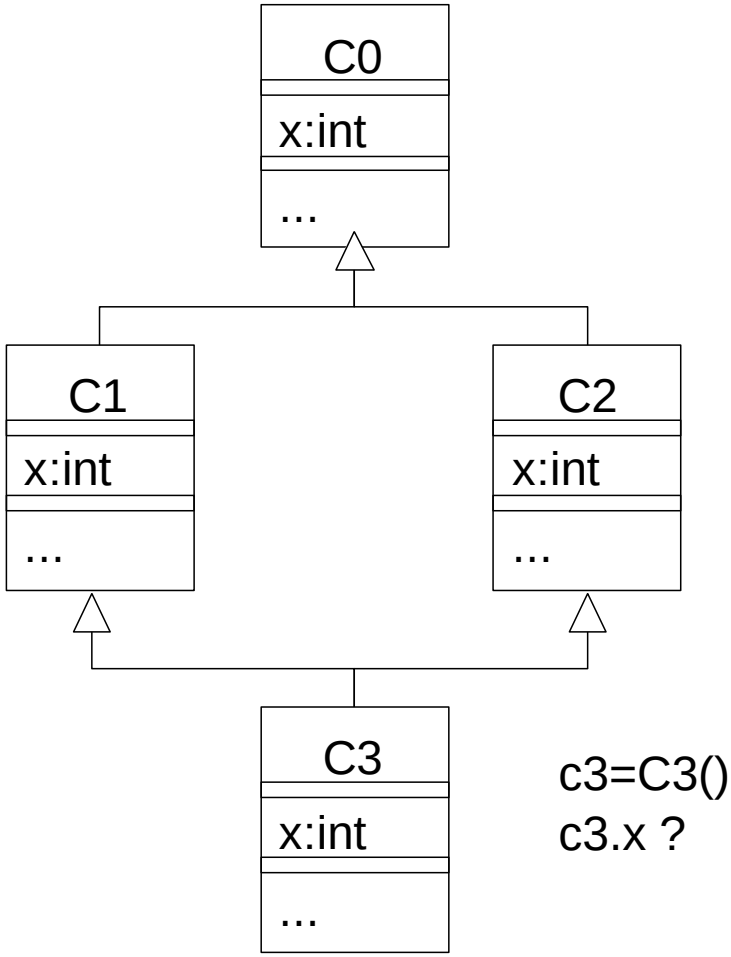
The Bad

- If one class can inherit from many classes, we may get multiple inheritance
- Which x should C3 inherit, the one from C1 or the one from C2?
- How can this be taken care of?



The Not So Bad

- If many classes can inherit from one class, we may get repeated inheritance
- C1 and C2 inherit x from C0. Now, they are all the “same” x, but which x does C3 inherit?



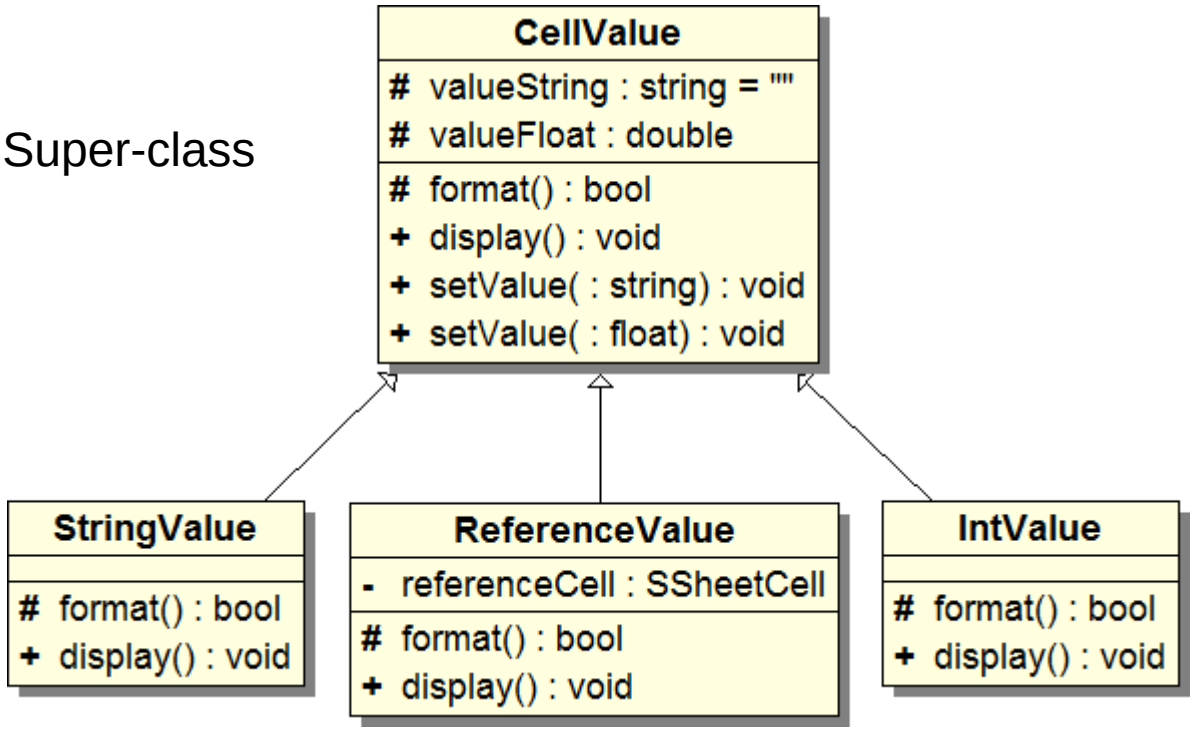
Polymorphism

Polymorphism == “many forms” in Greek

- A single method (or attribute) defined on more than one class that may take on different implementations in each different class
- An attribute or variable that may refer to objects of different classes at different times during program execution

Example of Polymorphism

Super-class



Sub-classes

Polymorphism

- Method format() demonstrates polymorphism
- When we call someObject.format(), the object which is being referenced will know how to show itself
- It must be ensured that format() is properly implemented for each subclass (and possibly the superclass) and that the user need not worry about the implementation

Which show() to call?

- Which format() to execute will be determined at run-time (and NOT at compile-time). This is known as dynamic, run-time or late binding

```
Object o
o = Object.new()
s = String.new()
t = Tuple.new()
...
if condition :
  o = s
else :
  o = t
...
o.format()
```

Example

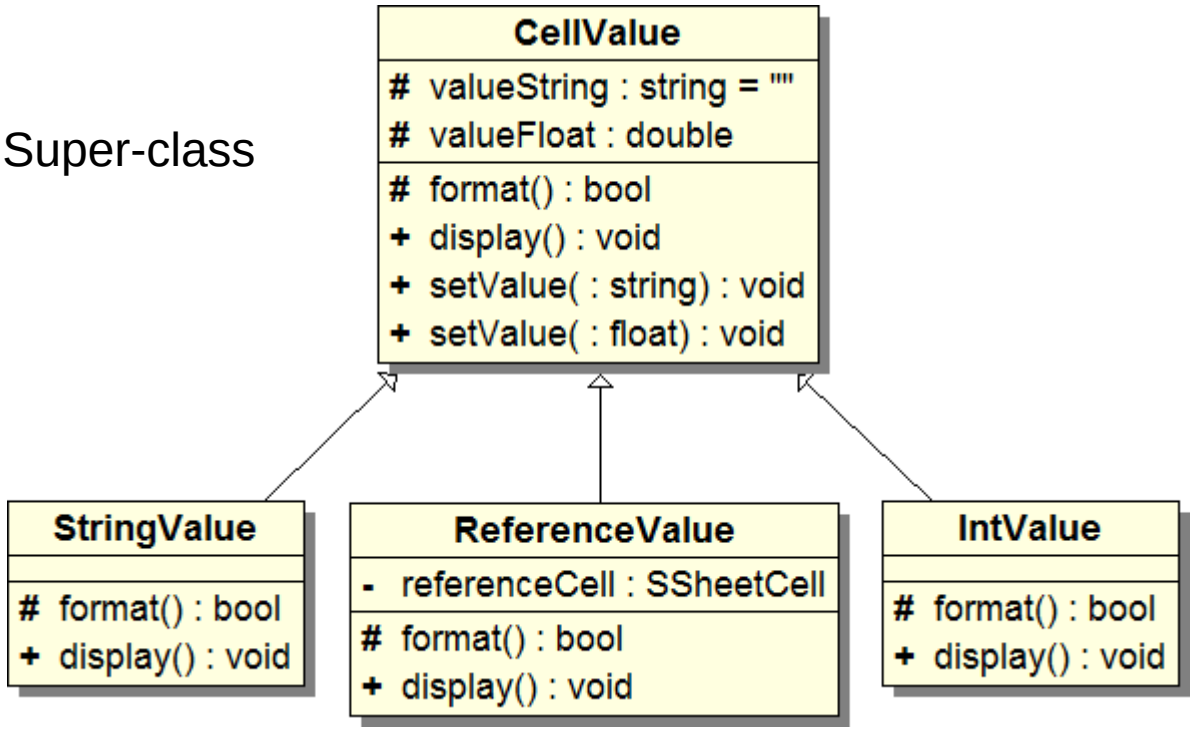
- At run-time, the object o may be an object of type String or of type Tuple.
- What o actually is will only be determined at run-time, after the user's input.
- When o.format() is executed, the method format() of the appropriate object will be executed.

Overloading vs. Overriding

- Overriding is the redefinition of a method defined on a class C in one of C's subclasses.
  - We say that the name or symbol is “overridden”.
- Overloading of a name or symbol occurs when several operations (or operators) defined on the same class have that name or symbol.
  - We say that the name or symbol is “overloaded”.

Example of Overriding/Overloading

Super-class



Sub-classes

Overriding

- display() is an example of overriding as subclasses Number, String and Tuple redefined display().
- If we wish to actually execute setValue() of the superclass (Object), we would execute super.display() in the subclass.

  In Python: SuperClass.display()

Overriding can also be used to cancel certain inherited methods.

- Suppose we have a subclass Hash that cannot show itself, then we can override display() in class Hash to return some error.
- This is BAD DESIGN!

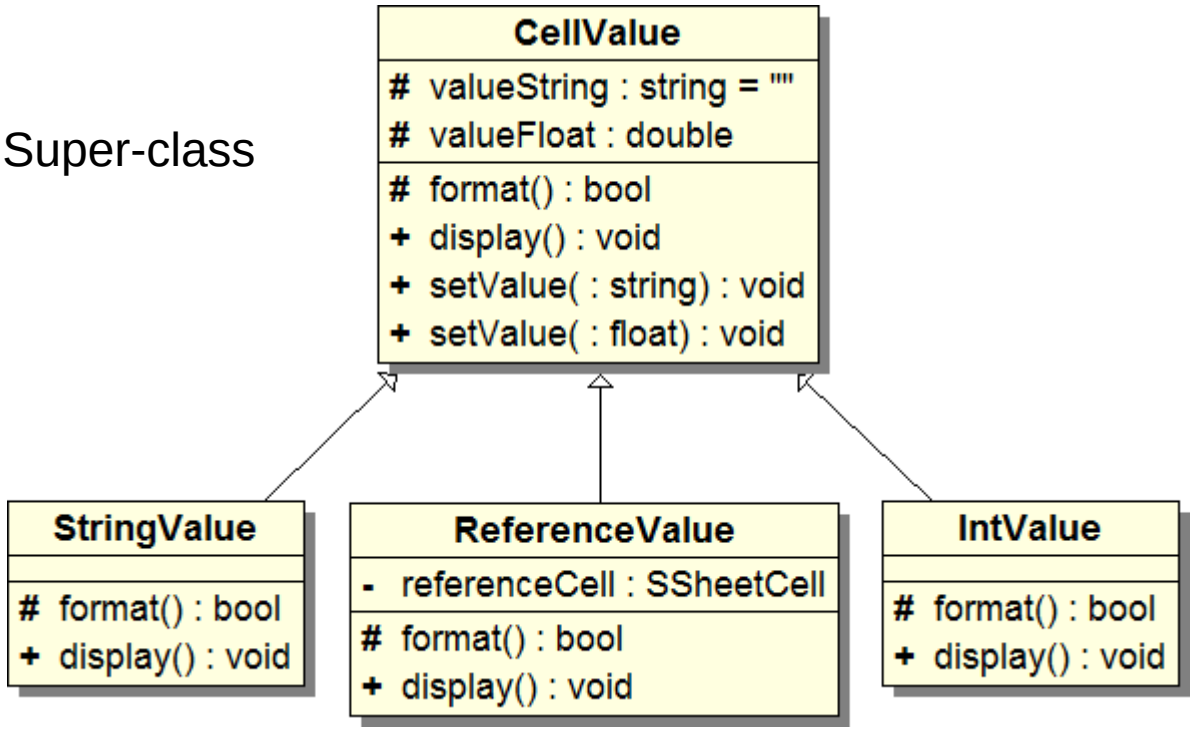
Overloading

- setValue(:string) is an example of overloading
  - setValue(:float) will update the float attribute of the object
  - setValue(:string) will update the string attribute of the object
- Which method will be executed depends on which method signature is used to call it.

  Python ?

CellValue : format()

Super-class



Sub-classes

More tricky

- If B and C are subclasses of A.
- Class D has the following methods.
  - m(B b)
  - m(C c)
- What happens if?
  - A var = new B(); D d = new D();
  - d.m(var)
- Depends on the lookup:
  - Lookup static : how to show A instance ?
  - Lookup is dynamic : call to m(B b) is made

Genericity

- Imagine we spend a lot of effort developing an algorithm to sort integers.
- I don't want to rebuild the algorithm if I store floats or strings.
- I want a generic algorithm for comparing items.
- Solution : Genericity (also known as templates)

Definition

- Genericity – one or more classes are used internally by some class and are only supplied at run-time (or upon instantiation)
- Genericity can be emulated using inheritance, but ...

Example

```
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

i
```

```
int main () {
  int i=5, j=6, resInt;
  long l=10, m=5, resLong;
  string s1="hello", s2="abc", resString;

  resInt    = GetMax<int>(i,j);
  resLong   = GetMax<long>(l,m);
  resString = GetMax<string>(s1,s2);
  cout << resInt    << endl;
  cout << resLong   << endl;
  cout << resString << endl;
  return 0;
}
```

Example

```
#include <iostream>
using namespace std;

template <class T>
class Pair {
    T a, b;
  public:
    Pair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};
```

```
template <class T>
T Pair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  Pair<int> myIntPair(100, 75);
  cout << myIntPair.getmax() << endl;

  Pair<string> myStringPair ("hello", "ab");
  cout << myStringPair.getmax() << endl;

  return 0;
}
```