Inheritance: The is-a relation

Steven Zeil

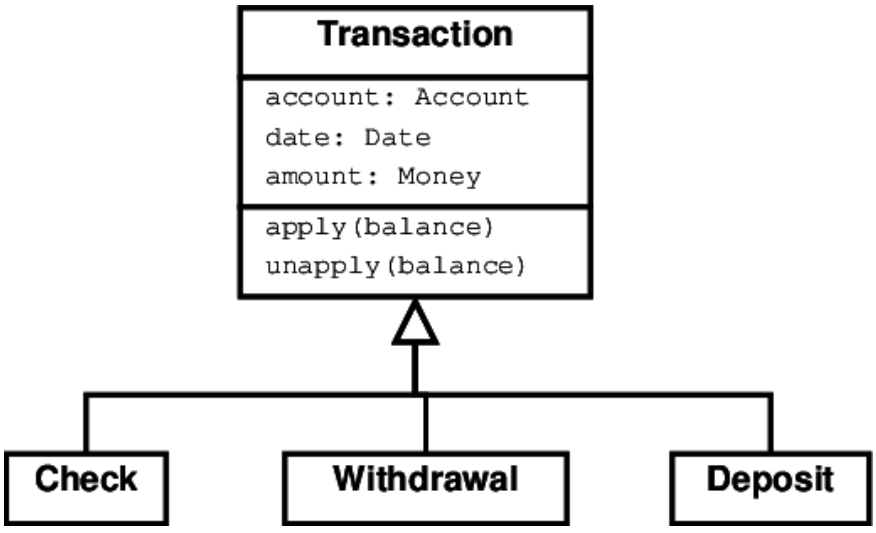
Last modified: Dec 19, 2017

Contents:

1 Generalization & Specialization

- Conceptually, class A is a generalization of class B if every B object is also an A object.
- "everything we say about an" A object "is also true for" a B object.
- At the specification/implementation level, class A is a generalization of class B if B conforms to the public interface of A.

Specialization Example



For example, a check and a deposit are actually specializations of the more general concept of a "transaction".

1.1 Inheritance

A class C *inherits* from class D if C has all the data members and messages of D.

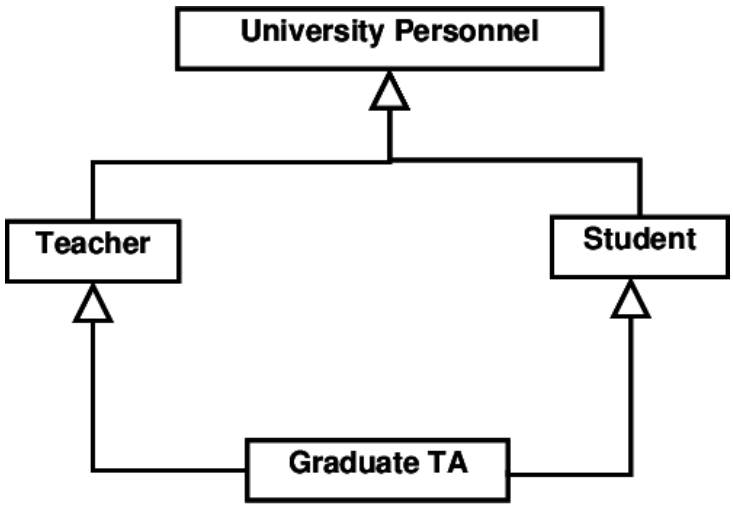
- C may have additional data members and messages not present in D.

D is called a *base class* of C.

Inheritance Example

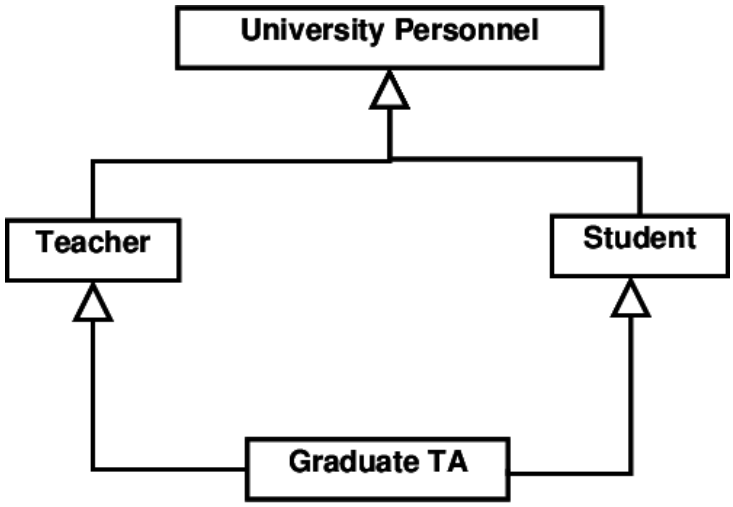
This example suggests that Teachers and Students will inherit from University Personnel.

- So if University Personnel have data members *name* and *uin* (University ID Number), the same will be true of `Teachers` and `Students`.
- However, `Teachers` may have data members not common to all `UniversityPersonnel`, such as a salary.
- `Students` may likewise have data members not common to all `UniversityPersonnel`, such as a grade point average *gpa*.



Inheritance Example

- The diagram also suggests that Graduate TAs will inherit from both `Teachers` and `Students`, so `GraduateTA`s will have a *name*, *uin*, *and* a *salary* and a *gpa*.



Multiple Inheritance

Inheriting from multiple base classes is called *multiple inheritance*.

It's reasonably common in domain and analysis models, but designers often try to remove it before they get to the stage of coding. That's because multiple inheritance can lead to complications.

- For example, does a `GraduateTA` get one *name* or two?

- One *uin* or two?

1.2 Subtyping

A closely related idea:

- A type C is a *subtype* of D if a value of type C may be used in any operation that expects a value of type D.
    - C is called a subclass or subtype of D.

D is called a superclass or *supertype* of C.

What's the Difference?

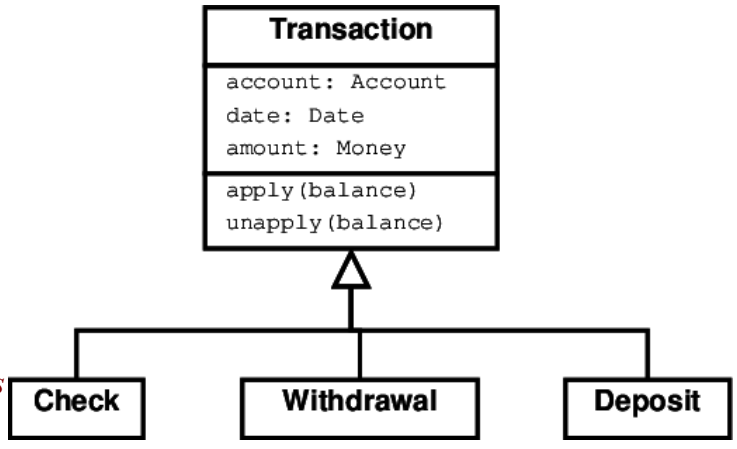
- Inheritance deals with the class's members.
- Subtyping deals with non-members

Effects of Subtyping and Inheritance

applyToBal.cpp +

```
void applyToCurrentBalance (CheckBook cbook, Transaction trans) {
   Balance b = cbook.getCurrentBalance();
   trans.apply (b);
   cbook.setCurrentBalance(b);
}
   ⋮
CheckBook myCheckBook;
Check check;
Transaction transaction;
Balance bal;
   ⋮
bal = myCheckBook.getCurrentBalance();
transaction.apply (bal);                          ①
check.apply (bal);                                ②
applyToCurrentBalance(myCheckBook, transaction); ③
applyToCurrentBalance(myCheckBook, check);       ④
```

For example, the last four statements in the code above are all legal, but the reasons vary:



- ① We can make this call because apply is a member function of Transaction and *transaction* is of type Transaction.

  Nothing special there.

- ② We can make this call because apply is a member function of Transaction, Check *inherits* from Transaction, and Check therefore has that same member function.

- ③ We can make this call because applyToCurrentBalance is a non-member function that takes a Transaction as a parameter, and *transaction* is indeed of type Transaction.

- ④ Finally, we can make this call because applyToCurrentBalance is a non-member function that takes a Transaction as a parameter, *check* is of type Check which is a *subtype* of Transaction, and we can use a subtype object in any operation where an object of the supertype is expected. Inheritance does not come into play in this case, because the function we are looking at is not a member function.

C++ Combines Inheritance & Subtyping

In most OOPLs, including C++, inheritance and subtyping are combined.

- A base class is always a superclass.
- An inheriting class is always a subclass.
- A superclass is always a base class.
- An subclass is always an inheriting class.

That makes the distinction between inheritance and subtyping moot in C++.

The same does not hold, however, of Java, where only the first two of the four above statements are true.

2 Inheritance & Subtyping in C++

The construct

```
class C : public Super {
```

indicates that

- Inheritance
- C inherits from Super
- Super is a base class of C
- Subtyping
- C is a subtype of Super
- Super is a supertype of C

2.1 Inheritance Example - Values in a Spreadsheet

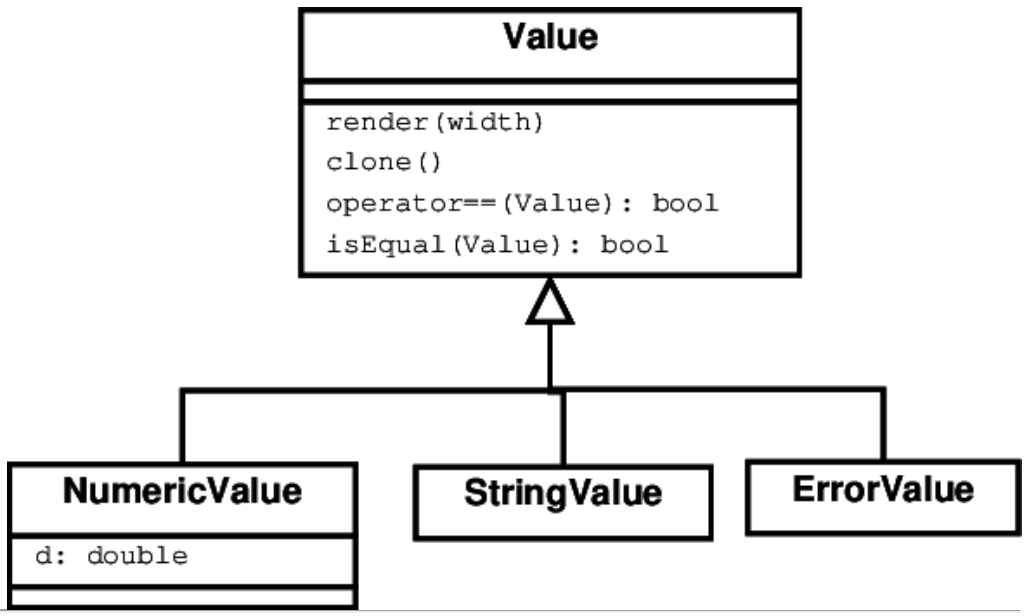
cell.h +

```
#ifndef CELL_H
#define CELL_H

#include "cellname.h"
```

```
#include "observable.h"
#include "observer.h"
#include "strvalue.h"

class Expression;
class Value;
class SpreadSheet;


// A single cell within a Spreadsheet
class Cell: public Observable, Observer
{
public:
  Cell (SpreadSheet& sheet, CellName name);
  Cell(const Cell&);

  ~Cell();

  CellName getName() const;

  const Expression* getFormula() const;
  void putFormula(Expression*);

  const Value* getValue() const;
  const Value* evaluateFormula();

  bool getValueIsCurrent() const;
  void putValueIsCurrent(bool);


  virtual void notify (Observable* changedCell);

private:
  SpreadSheet& theSheet;
  CellName theName;
  Expression* theFormula;
  Value* theValue;
  bool outOfDate;
  static StringValue defaultValue;
};

#endif
```

Every cell in the spreadsheet contains

- a formula (expression)
- a value



Values

The interface for values is

value.h +

```
#ifndef VALUE_H
#define VALUE_H

#include <string>
#include <typeinfo>

//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
class Value
{
public:
  virtual ~Value() {}


  virtual std::string render (unsigned maxWidth) const = 0;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const = 0;
  // make a copy of this value

protected:
  virtual bool isEqual (const Value& v) const = 0;
  //pre: typeid(*this) == typeid(v)
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

  friend bool operator== (const Value&, const Value&);
};

inline
bool operator== (const Value& left, const Value& right)
{
  return (typeid(left) == typeid(right))
    && left.isEqual(right);
}

#endif
```

- Values come in many different kinds

Numeric Values

Numeric values hold numbers.

numvalue.h +

```
#ifndef NUMVALUE_H
#define NUMVALUE_H

#include "value.h"

//
// Numeric values in the spreadsheet.
//
class NumericValue: public Value
{
  double d;

public:
  NumericValue():d(0.0)  {}
  NumericValue (double x): d(x) {}

  virtual std::string render (unsigned maxWidth) const;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const;

  double getNumericValue() const {return d;}

protected:
  virtual bool isEqual (const Value& v) const;
  //pre: typeid() == v.typeid()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

};

#endif
```

- We infer by inheritance that NumericValue has function members render(), clone(), etc.
- Therefore this code is OK (by inheritance):

```
void foo (NumericValue nv) {
    cout << nv.render(8) << endl;
}
```

- We infer via subtyping that the following code is OK:

```
void foo (Value v; NumericValue nv) {
    if (v == nv) {
       ⋮
```

String Values

String values hold numbers.

strvalue.h +

```
#ifndef STRVALUE_H
#define STRVALUE_H

#include "value.h"

//
// String values in the spreadsheet.
//
class StringValue: public Value
{
  std::string s;
  static const char* theValueKindName;

public:
  StringValue()  {}
  StringValue (std::string x): s(x) {}


  virtual std::string render (unsigned maxWidth) const;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  std::string getStringValue() const {return s;}

  virtual Value* clone() const;



protected:
  virtual bool isEqual (const Value& v) const;
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

};

#endif
```

- Note how numeric and string values each add data members that actually store the data they need and that other types of values would find irrelevant.

Error Values

Error values store no data at all, but are used as placeholders in a cell whose calculations have failed for some reason.

- (E.g., in any spreadsheet program you have available, try dividing by zero in a cell).

errvalue.h +

```
#ifndef ERRVALUE_H
#define ERRVALUE_H

#include "value.h"
```

```
//
// Erroneous/invalid values in the spreadsheet.
//
class ErrorValue: public Value
{
  static const char* theValueKindName;

public:
  ErrorValue()  {}

  virtual std::string render (unsigned maxWidth) const;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const;

protected:
  virtual bool isEqual (const Value& v) const;
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

};

#endif
```

3 Overriding Functions

When a subclass inherits a function member, it may

- inherit the function's body from the superclass, or
- override the function by providing its own body

Overriding: Declaring Your Intentions

overriding.h +

```
class A {
public:
  void foo();
  void bar();
  void baz();
};

class B: public A {
public:
  void foo();          ①
  void bar(int k);     ②
  void bar() const;    ③
};                     ④
```

- ① B declares that it will override A::foo(). B inherits the declaration of foo() but will provide its own body.
- ② This does not override A::bar().
  - Changing parameters *overloads* a function, but (access to) the original function is unaffected.
- ③ This does not override A::bar(), either. It also overloads it with different parameter types.
  - The implicit parameter *this* is a const B* instead of an A*.
- ④ Because B did not override A::bar() or A::baz(), it inherits those declarations *and* their bodies from A.

Access to Functions

- Even when we do override, the original function can be called explicitly:

```
B b;
b.foo();     // calls B::foo()
b.A::foo();  // calls the original A::foo()
```

- This is often useful when the overriding function is supposed to do the same thing as the original *plus* something extra.

```
void B::foo()
{
  A::foo();
  doSomethingExtra();
}
```

Example of Overriding

As an example of overriding, consider these four classes, which form a small inheritance hierarchy.

animalOv.cpp +

```
class Animal {
public:
  String eats() {return "food";}
  String name() {return "Animal";}
};
```

```
class Herbivore: public Animal {
public:
   String eats() {return "plants";}
   String name() {return "Herbivore";}
};

class Ruminants: public Herbivore {
public:
   String name() {return "Ruminant";}
};

class Carnivore: public Animal {
public:
   String name() {return "Carnivore";}
};

void show (String s1, String s2) {
    cout << s1 << " " << s2 << endl;
}
```

Note that several of the inheriting classes override one or both functions in their base class.

Question: Now, suppose we run the following code. What will be printed by each of the show calls?

```
Animal a;
Carnivore c;
Herbivore h;
Ruminant r;
show(a.name(), a.eats());        // AHRC fpgm
show(c.name(), c.eats());        // AHRC fpgm
show(h.name(), h.eats());        // AHRC fpgm
show(r.name(), r.eats());        // AHRC fpgm
```

Answer: +

Inheritance and Encapsulation

An inheriting class does *not* get access to private data members of its base class:

```
class Counter {
   int count;
public:
   Counter() {count = 0;}
   void increment() {++count;}
   int getC() const {return count;}
};

class HoursCounter: public Counter {
public:
   void increment() {
     counter = (counter + 1) % 24; // Error!
   }
};
```

Protected Members

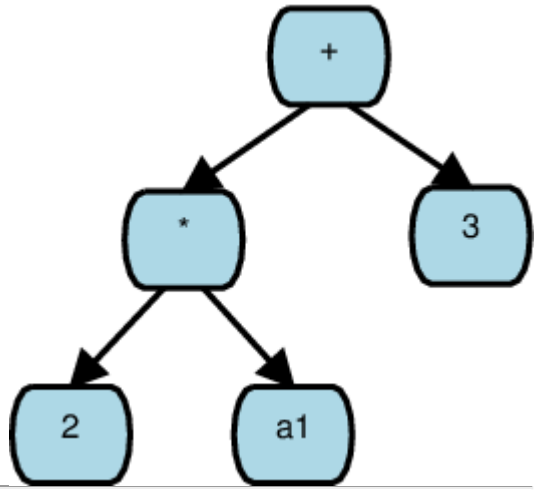
Data members marked as `protected` are accessible to inheriting classes but private to all other classes.

```
class Counter {
protected:
   int count;
public:
   Counter() {count = 0;}
   void increment() {++count;}
   int getC() const {return count;}
};

class HoursCounter: public Counter {
public:
   void increment() {
     counter = (counter + 1) % 24; // OK
   }
};
```

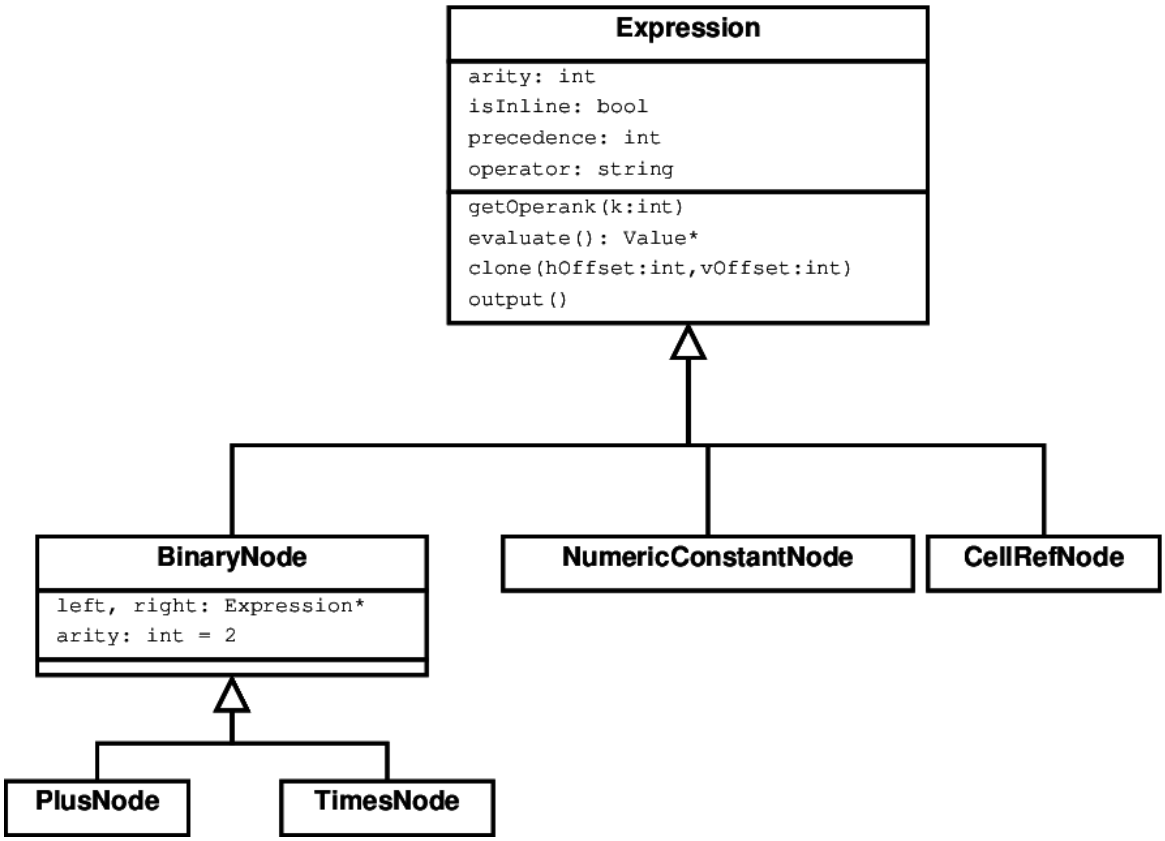
4 Example: Inheritance and Expressions

Expression Trees

- An Expression is represented as a tree
  - This tree represents `2*a1 + 3`
- An Expression comes in several parts
  - Internal nodes represent an single operator being applied to some number of operands
    - Each operand is a (sub)Expression
  - Leafs represent constants and variables (cell names)



Expression Inheritance Hierarchy



We would expect the lower-level classes like PlusNode and TimesNode to override the evaluate function to do addition, multiplication, etc., or whatever it is that distinctively identifies that particular “kind” of expression from all the other possibilities.