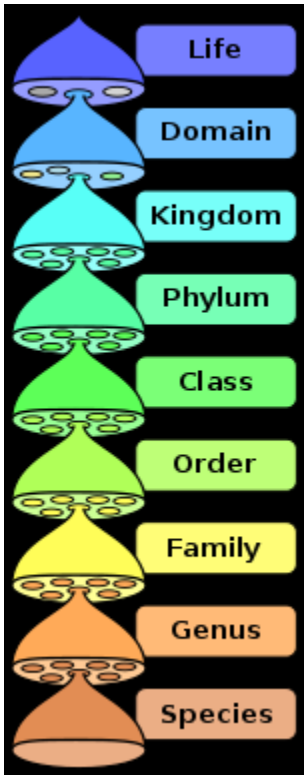
**Inheritance**

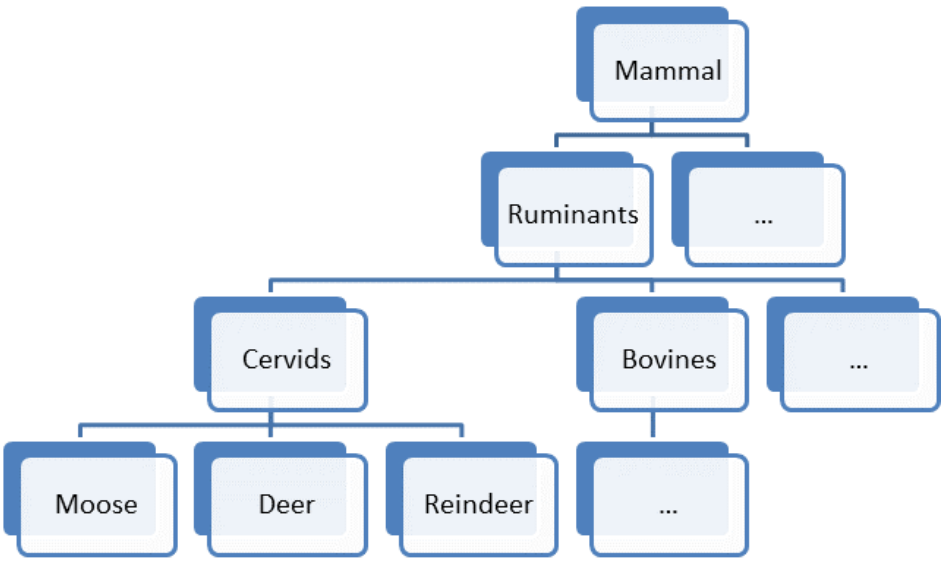
The concept of inheritance is one of the key features of an object-oriented programming language. Inheritance allows a programmer to define a general class, and then later define more specific classes that share or *inherit* all of the properties of the more general class. This allows the programmer to save time and energy that might otherwise be spent writing duplicate code.

Related to inheritance is the concept of *polymorphism*. Polymorphism allows us to invoke the "correct" method in an inheritance hierarchy. We'll see how this is used later.

The idea of inheritance is similar to the taxonomy of living things. You probably recall the following hierarchy from a biology class:




For example, to zoom in on the deer family (cervid), we could make a tree like the following.




Note that this hierarchy is an "IS-A" hierarchy. A deer "IS-A" cervid. In turn, a cervid "IS-A" ruminant, and so forth. The term "parent" is the term used to describe concepts going up the hierarchy, and "child" is the term when going down the hierarchy (imagine a family tree). For example, a deer is a "child" of cervid, and cervid is the "parent" of deer. Note that any property that holds for a parent also holds for a child. For example, if we know that mammals are warm-blooded, then we know that everything listed below mammal is also warm-blooded. We'll take advantage of this concept from a programming perspective to reduce duplicative code.

For an example in C++, recall the CreditCardTransaction class that we wrote earlier:

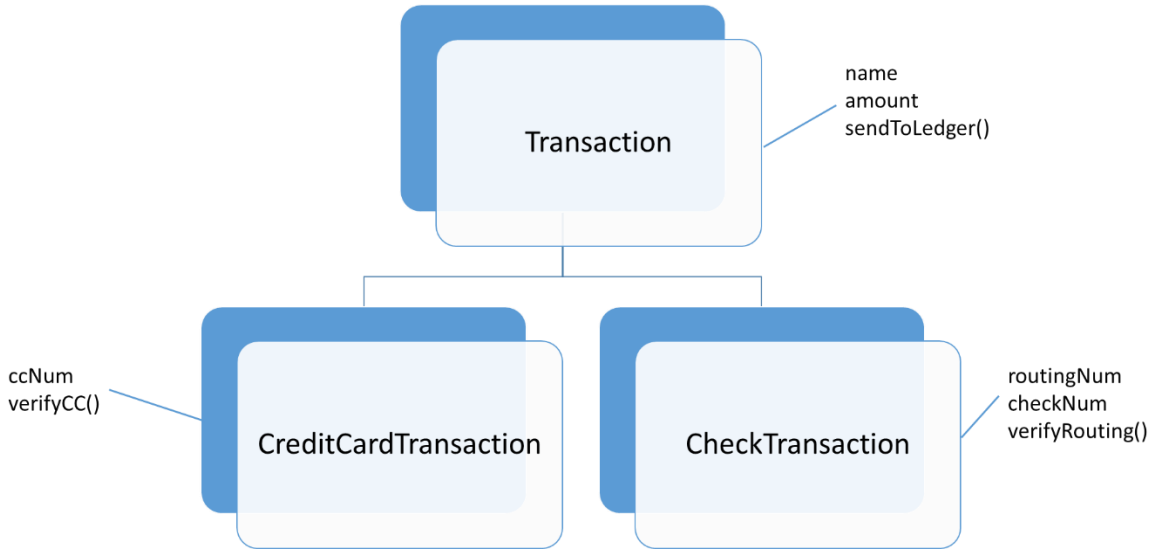
```
class CreditCardTransaction
{
public:
    void CreditCardTransaction();
    void  CreditCardTransaction(string name, string number,
            double amount);
    string getName();
    string getNumber();
    double getAmount();
private:
    string name;
    string number;
    double amount;
};
```

There are other kinds of transactions we might want to support, such as CashTransaction or CheckTransaction or PaypalTransaction. Let's just look at how we might write CheckTransaction:

```
class CheckTransaction
{
public:
     void CheckTransaction();
     void  CheckTransaction(string name, string checkNum,
                       string routingNum, double amount);
     string getName();
     double getAmount();
     string getCheckNum();
     string getRoutingNum();
private:
     string name;
     string checkNum;
     string routingNum;
     double amount;
};
```

These two classes will work fine, but it's totally inefficient. The code is almost identical except for some values in the constructor and related accessors and mutators. The name and amount portions of the classes are the same. This is where inheritance lets us eliminate duplicate code.

In this case we'll make a parent class called `Transaction` and define the `CheckTransaction` and `CreditCardTransaction` classes as children of the `Transaction` class. In OOP terminology, `Transaction` is called the **base** class and `CheckTransaction` and `CreditCardTransaction` are called the **derived** classes. Derived classes will inherit all of the public (or protected) methods and instance variables of its parents so we don't have to write these more than once!




Private instance variables defined in a parent class exist in derived classes, but can't be accessed directly by name. Instead they must be accessed through a method. Private

methods defined in a parent class can't be accessed directly from a derived class. There is a modifier, **protected**, that allows access in any derived classes but not any other class.

To create a derived class, we add a colon after the class definition then the word "public" followed by the name of the base class. E.g.:

```
class CreditCardTransaction : public Transaction
```

(There is also a type of private inheritance, but we won't cover it here.) Here is the first version of our new class for `Transaction`. We only put in code that is relevant to all sub-classes.

Here is a simple version of our classes; this is where we need to organize them in separate header and implementation files:

**File: Transaction.h**

```
#pragma once
#include <iostream>
#include <string>

using std::string;

class Transaction
{
  public:
        // Constructors
        Transaction();  // Default constructor
        Transaction(string newName, double newAmount);
        string getName();
        double getAmount();
  private:
        string name;
        double amount;
};
```

**File: Transaction.cpp**

```
#include <iostream>
#include <string>
#include "Transaction.h"
using std::string;

Transaction::Transaction()
{
        name = "Unknown";
        amount = 0;
}
Transaction::Transaction(string newName, double newAmount) :
   name(newName), amount(newAmount)
{
}
```

```
string Transaction::getName()
{
        return name;
}
double Transaction::getAmount()
{
        return amount;
}
```

**File: CreditCardTransaction.h**

```
#pragma once
#include <iostream>
#include <string>

using std::string;

class CreditCardTransaction : public Transaction
{
  public:
        // Constructors
        CreditCardTransaction();  // Default constructor
        CreditCardTransaction(string newName, string newNumber,
                              double newAmount);
        string getNumber();
  private:
        string number;
};
```

**File: CreditCardTransaction.cpp**

```
#include "Transaction.h"
#include "CreditCardTransaction.h"

using std::string;

CreditCardTransaction::CreditCardTransaction() : Transaction(),
        number("")
{
}

CreditCardTransaction::CreditCardTransaction(string newName, string
newNumber,
                                             double newAmount) :
   Transaction(newName, newAmount), number(newNumber)
{
}

string CreditCardTransaction::getNumber()
{
        return number;
}
```

The following test code in main could go anywhere:

```
int main()
{
        CreditCardTransaction t1;
        CreditCardTransaction t2("Chuck Jones",
                                 "1234 1234 1234 1234", 55.45);

        cout << t1.getName() << " " << t1.getNumber() << " "
                << t1.getAmount() << endl;;
        cout << t2.getName() << " " << t2.getNumber() << " "
                << t2.getAmount() << endl;
}
```

Things to note: we call the parent constructors from the child constructors.
We can call getName, getNumber, or getAmount with a CreditCardTransaction object. It has "inherited" the getName and getAmount functions!

Questions: How might you add on a CashTransaction class?

If we had a "sendToLedger" function that applies to all kinds of transactions, where should we put it?

If we want to write a function to verify a credit card number where would we put it?

To do in class: If we make variables protected instead of private then we can access them inside the derived class.

C++ lets you redefine methods in the derived class that are defined in the parent class. For example, let's add a "print" method to the Transaction class:

```
void Transaction::print()
{
  cout << "Name: " << name << " Amount: " << amount << endl;
}
```

And we add a method of the same name to the CreditCardTransaction class:

```
void CreditCardTransaction::print()
{
        Transaction::print(); // Call parent print function
        cout << "Number: " << number << endl;  // Add number
}
```

We have now redefined the print method if we invoke it from a CreditCardTransaction object:

```
CreditCardTransaction t2("Chuck Jones", "1234 1234 1234 1234",
                                55.45);
t2.print();           // prints out name, amount. number
```

Note that if we create a Transaction object and call print, then it calls the Transaction print function. If for some reason we want to we can also force calling the Transaction print function for a CreditCardTransaction object:

```
Transaction t3("Bob", 10.05);
t3.print();
t2.Transaction::print();
```

Next time we will do something different than redefining a function – instead we'll make a virtual function, which turns out to be one of the big ideas in object-oriented programming.

**Takeaways**

1. A derived class inherits public (or protected) methods from the base class
2. Derived methods override methods of the same name in the base class
3. Use the scope resolution operator to invoke methods from the base class
4. Private variables are inherited by the derived class, but are not directly accessible by name. Public or protected variables are directly accessible by name.

**Multi-Level Classes**

In our example we only had one parent and one child class, but we could have a chain of multiple parents. In fact, in C++ it is possible for one class to have two separate parents, but this is very uncommon (and not allowed in most programming languages).