Inheritance

From your experience in Java programming, you are familiar with the idea of inheritance and how to use this in programming.

In this introduction, I'll describe inheritance in Ruby from scratch.

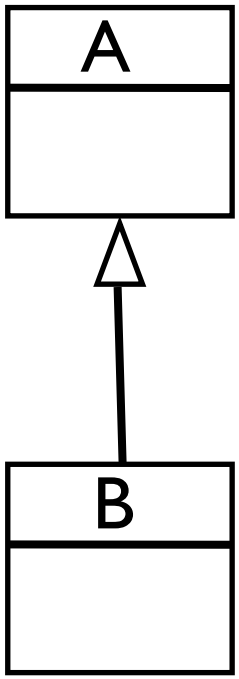
Much of this material should seem familiar to you. Remember that inheritance is essentially the same, regardless of what language it is expressed in.

At the same time, inheritance in Ruby is not exactly the same as in Java, so be alert to the differences.

Inheritance in Ruby

Ruby allows us to define a new class in terms of an existing one, mimicking the way we typically define a new concept in terms of an existing one.

These statements are equivalent:
B **inherits** from A.
A is a **superclass** of B.
B is a **subclass** of A.



Superclass and subclass are the Ruby terms.

Simple Subclassing

Say we have defined a class **`Mammal`**:

```
class Mammal
    ...
end
```

By now writing:

```
class Dog < Mammal
end
```

We state that **`Dog`** is a new class, a subclass of **`Mammal`**.

An instance of **`Dog`** will have the same methods as an instance of **`Mammal`**.

Methods are inherited

For example, if **`Mammal`** were defined as:

```
class Mammal
   def breathe
     puts "breathe in, breathe out"
   end
end
```

Then

```
   ruskin = Dog.new
   ruskin.breathe
```

will result in invoking the operation **`breathe`** as defined in class **`Mammal`** on the object **`ruskin`**.

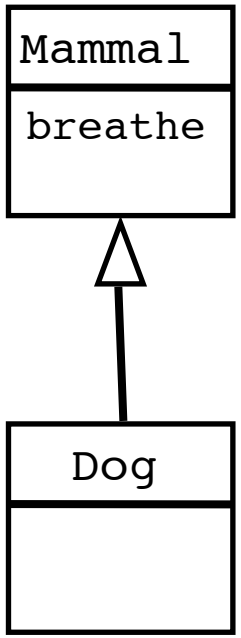
Method Binding

When the statement

```
ruskin.breathe
```

is executed, Ruby interpreter tries to find an operation called **`breathe`** in the class **`Dog`**.

It fails, so the search continues up the inheritance hierarchy. The operation **`breathe`** in the class **`Mammal`** is found and used.



The process whereby an invocation is linked to an actual operation is called **binding**. What has been described here is done at run-time and so is called **run-time binding** or **dynamic binding**.

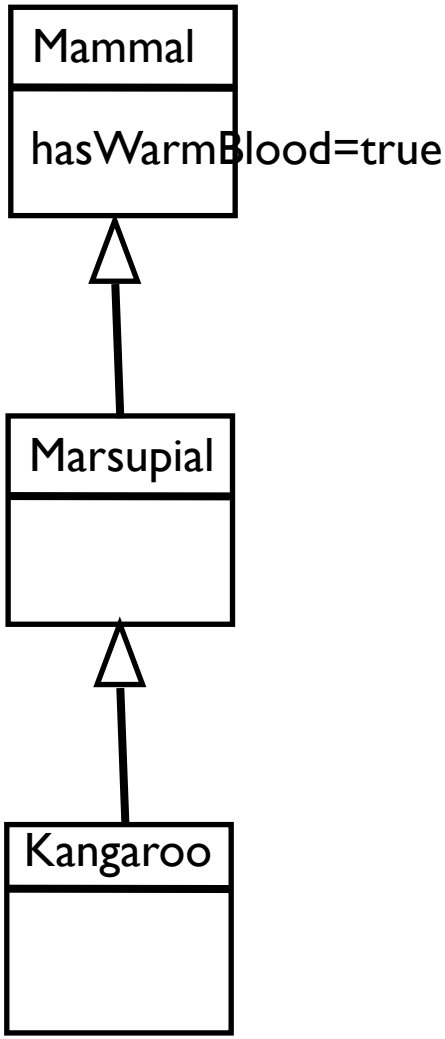
Java and C++ both support **compile-time** or **static binding**. This difference doesn't exist in Ruby.

In Ruby, there is only run-time.

A Real-World Equivalent

This mimics how we would search for information in a real-world hierarchy, e.g., "Does a kangaroo have warm blood?".

This information is not stored in the class `Kangaroo`, but in the class `Mammal`, which is an indirect superclass of Kangaroo.



…implemented in Ruby

The above hierarchy could be implemented in Ruby as follows:

```
class Mammal
   def hasWarmBlood?
      true
   end
end

class Marsupial < Mammal
    ...
end

class Kangaroo < Marsupial
    ...
end
```

Extending the Subclass

Creating new classes that are exactly the same as existing classes isn't of course useful. What is useful is that we can extend the subclass in various ways.

Consider again the example of a **`Mammal`** class that provides one method **`breathe`**.

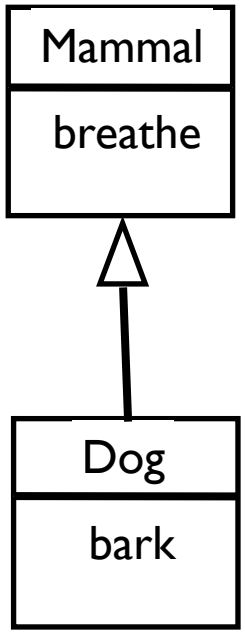
```
class Mammal
  def breathe
    puts 'breathe in, breathe out'
  end
end
```

Say we wish now to create a **`Dog`** class that can also bark…

Extending the Subclass

A dog is a type of mammal, so it has mammal behaviour, in our example breathing. It can also bark, and this we might represent as follows:

```
class Dog < Mammal
  def bark
    puts 'woof, woof'
  end
end
```



So a `Dog` is a `Mammal` which is also able to bark. This code uses both methods:

```
ruskin = Dog.new
ruskin.breathe
ruskin.bark
```

Inheritance only goes one way

Any operation that is available in the **`Mammal`** class can also be invoked on an object of the **`Dog`** class. New operations added to the **`Dog`** class can only be invoked on **`Dog`** objects.

So this won't work (of course):

```
claude =  Mammal.new
claude.bark
```

In general we extend the subclass by adding new methods and possibly some new instance variables.

So, how are instance variables inherited?

Instance Variables are NOT inherited!

```
class Mammal
  def initialize name
    @name = name
  end
  def breathe
    puts 'breathe in and out'
  end
  def get_name
    @name
  end
end

class Dog < Mammal
  def initialize
  end
  def bark
    puts "woof, woof"
  end
end
```

In this code, the **`Mammal`** class has an instance variable **`@name`**.

What does the following code do??

```
my_seal = Mammal.new "claude"
puts my_seal.get_name

my_dog = Dog.new
puts my_dog.get_name
```

Here's how to get what we want

```
class Mammal
  def initialize name
    @name = name
  end
  def breathe
    puts "breathe in and out"
  end
  def get_name
    @name
  end
end

class Dog < Mammal
  def initialize name
    super name
  end
  def bark
    puts "woof, woof"
  end
end
```

This is called **chaining**. Sign of good design!

Here the initializer in the **Dog** class invokes the initialiser in the **Mammal** class.

What does the following code do?

```
my_dog = Dog.new "ruskin"
puts my_dog.get_name
```

Omitting the initializer has the same effect.

Instance variables in a class hierarchy

Say an object has never executed a statement using the variable `@count` before and it encounters

```
puts @count
```

what will happen?

Say an object has never executed a statement using the variable `@count` before and it encounters

```
@count = 10
```

what will happen?

Any object can have only ONE instance variable called **`@count`**. This is never "declared." The variable comes into existence when **`@count`** is first used.

So to use an instance variable introduced in a superclass, just call a method that introduces it (often the initialiser).

Public, Private and Protected

Instance variables are **private** to the object as we've seen.
However, they are accessible throughout the class hierarchy.

```
class Mammal
   def initialize name
      @name = name
   end
end

class Dog < Mammal
   def bark
      puts "#{@name} goes woof"
   end
end
```

This would not be allowed in Java/C++.

Private Methods

Methods are **public** by default, so they can be invoked from outside the object.

If we make a method **private**, it can only be invoked from *inside* the object.

Thus, a private method is accessible to subclasses. unlike Java/C++!

The following both make **`foo`** and **`foobar`** private:

```
class Example
  ...
  private
  def foo
    ...
  end
  def foobar
    ...
  end
end
```

```
class Example
  ...
  def foo
    ...
  end
  def foobar
    ...
  end
  private :foo, :foobar
end
```

More about Private Methods

If you want a helper method in a class, but clients don't need it, make it private.

Private methods can never be invoked on an object! In the previous example, only the first statement is a valid way to invoke foo:

```
foo
o.foo
self.foo
```

both these are wrong!

initialize is just a private method that is invoked when an object is created. You can invoke it again (from within the object) if you wish.

Initializers are like normal, private methods

When an object instance is created using

```
Classname.new
```

the `initialize` method is invoked on the object.

Apart from that, `initialize` is just like a normal, private method. In particular:

- it can be invoked anywhere from within the object
- if it's not defined in the current class, it's looked for in the superclass, and so on

Protected Methods

A protected method is like a private one, with one difference.

It may also be invoked from **another object** of the **same class**.

Use a protected method when you want an object to share state with other objects of the same class, but not external clients.

Protected methods are fairly uncommon in Ruby. We may see an example later in the module.

Access rights to the Superclass **in Java**

In Java, what access does a subclass have to its superclass? The following example illustrates the rules:

```
class A {
    public  void pub();
    protected void prot();
    private void priv();
}

class B extends A {
    void foo() {
        pub();   // fine
        prot();  // fine
        priv();  // compilation error! z() is not accessible.
    }
}
```

So in Java public and protected methods are visible to subclasses; private methods are hidden to subclasses.

Access rights **in Ruby**

In Ruby, what access does a subclass have to its superclass? The following example illustrates the rules:

```
class A
  public
  def pub
  end

  protected
  def prot
  end

  private
  def priv
  end
end
```

```
class B < A
  def foo
    pub     # fine
    prot    # fine
    priv    # fine
  end
end
```

So in Ruby, everything is visible to subclasses.
=> subclasses and superclasses are tightly coupled
=> only subclass a class you know well

Banking Example

We'll consider how to model this simple banking example as a class hierarchy in Ruby.

"A bank account has an associated name (the account holder) and a balance. Funds can be deposited to and withdrawn from the account. A savings account is a type of bank account that has an interest rate and enables deposit interest to be added to the balance."

Highlight the nouns

A bank account has an associated name (the account holder) and a balance. Funds can be deposited to and withdrawn from the account. A savings account is a type of bank account that has an interest rate and enables deposit interest to be added to the balance.

Which are the likely classes?
Which are the likely methods?
Which are the likely instance variables?

BankAccount class

```
class BankAccount
  def initialize account_owner
    @name = account_owner
    @balance = 0
  end

  def deposit(amount)
    @balance += amount
  end

  def withdraw(amount)
    @balance -= amount
  end
end
```

SavingsAccount class

```
class SavingsAccount < BankAccount
  def initialize account_owner, interest_rate
    super account_owner
    @interest_rate = interest_rate
  end

  def deposit_interest
    @balance += @interest_rate/100.0 * @balance
  end
end
```

Overriding

Say we want a SpecialSavingsAccount that penalises withdrawals (but has a higher rate of interest?

So the withdraw method in SpecialSavingsAccount must be **overridden** to apply a penalty.

(We'll also add the penalty to the initializer)

SpecialSavingsAccount class

```
class SpecialSavingsAccount < SavingsAccount
  def initialize account_owner, interest_rate, penalty
    super account_owner, interest_rate
    @penalty = penalty
  end

  def withdraw amount
    @balance -= @penalty
    super amount
  end
end
```

If a method in a subclass has the same name as one in a superclass, we say **overrides** it.

Which methods are invoked?

```
acc1 = SavingsAccount.new “Lucy” 6
acc2 =  SpecialSavingsAccount.new “John” 10 25

acc1.deposit 1000
acc2.deposit 1000

acc1.withdraw 10
acc2.withdraw 10

acc1 = acc2
acc1.withdraw 10
```

Inheritance Summary

We have reviewed inheritance in general and shown how it is used in a Ruby program.

We'll examine **polymorphism** in more detail later in the course.