A Quick Question

Consider this Java class:

```
class TaxPayer{
   ...
   public void calculateTax(){
      ...
      calculateIncomeTax();
      ...
   }
   public void calculateIncomeTax(){
      ...
   }
   ...
}
```

Do you know what method is invoked by the line of code in red?

Inheritance in Ruby

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.

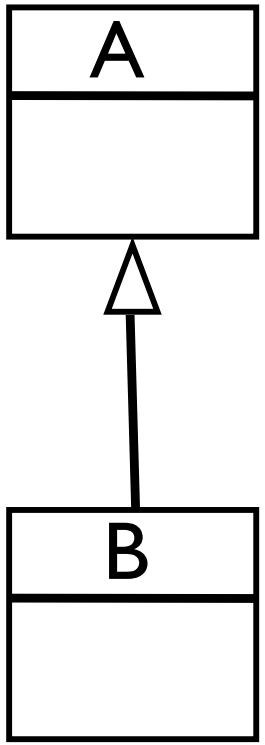
Much of this material should seem familiar to you. Remember that inheritance is essentially the same, regardless of what language it is expressed in.

At the same time, inheritance in Ruby is not exactly the same as in Java/C++, 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.
B is **derived** from A.



Superclass and subclass are the terms normally used in a Ruby context.

Which typically can do more, a subclass instance or superclass instance?

Which usually has more instances, the subclass or the superclass?

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 now have the same methods as an instance of **`Mammal`**. It inherits them.

Methods are inherited

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

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

Then

```
fido = Dog.new
fido.breathe
```

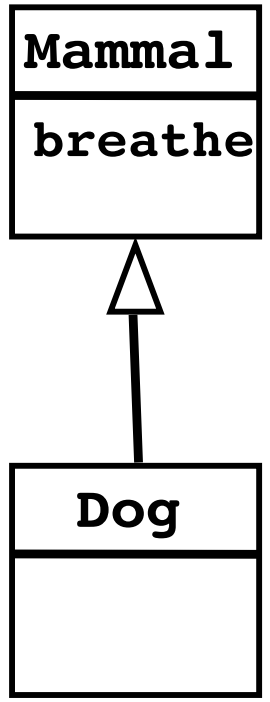
will result in invoking the method **`breathe`** as defined in the class **`Mammal`** on the object **`fido`**.

Method Lookup

When the statement

```
fido.breathe
```

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



Simplified description

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

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

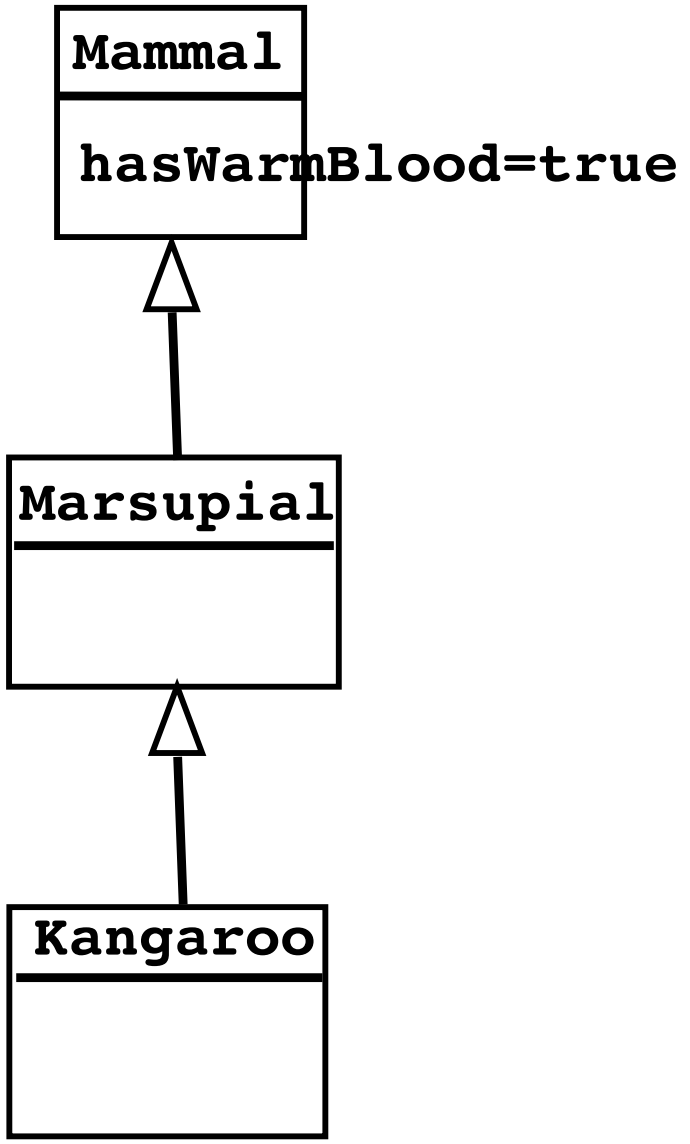
Java and C++ both support **compile-time** (or **static**) **binding** as well. This difference doesn't exist in Ruby.

In Ruby, all binding is dynamic.

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`**.

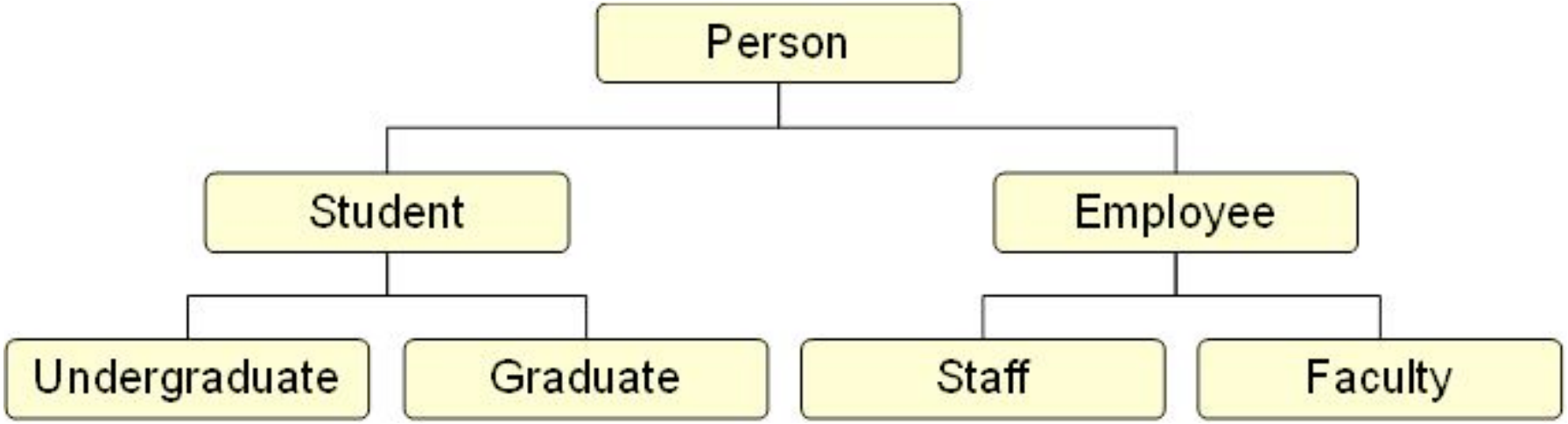


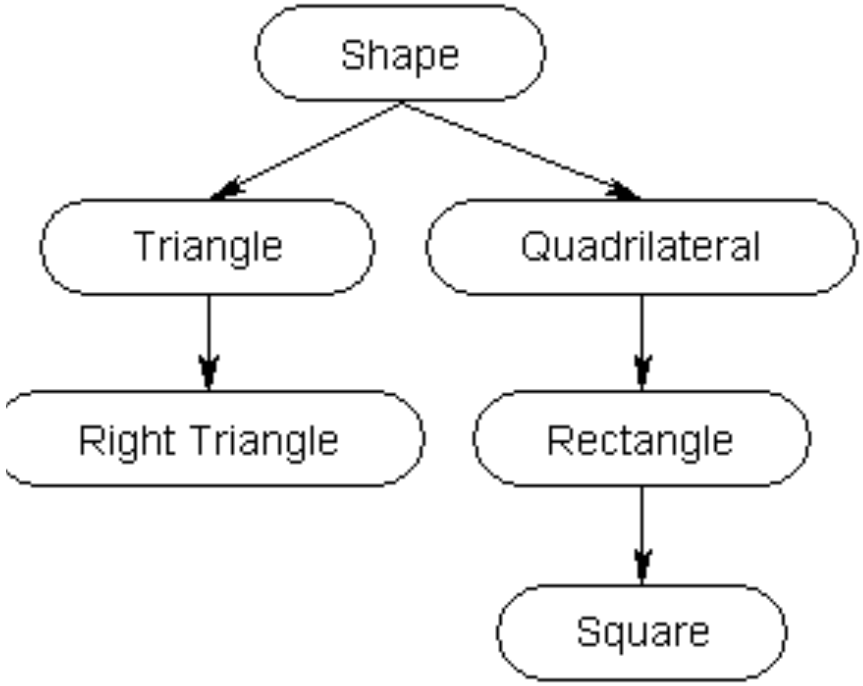
…implemented in Ruby

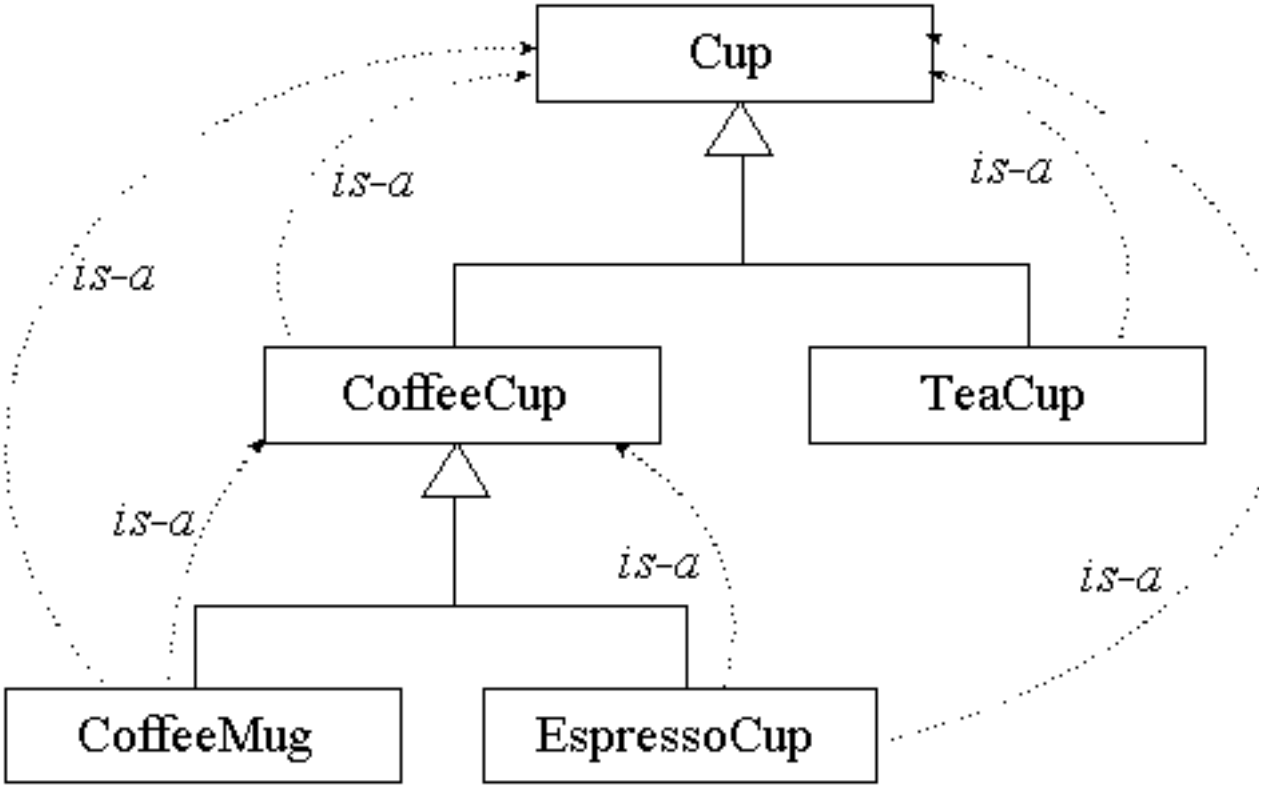
The above hierarchy could be implemented in Ruby thus:

```
class Mammal
  def has_warm_blood?
    true
  end
end

class Marsupial < Mammal
   ...
end

class Kangaroo < Marsupial
   …
end
```

Sample Inheritance Hierarchies







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`**:

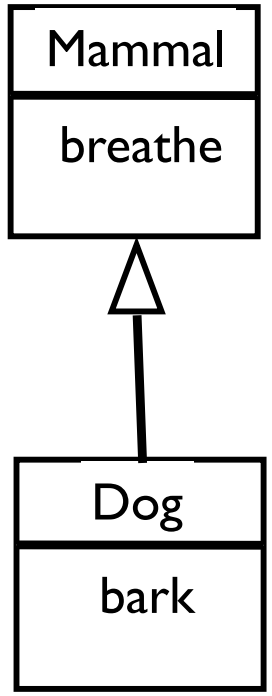
```
class Mammal
  def breathe
    puts 'breathe in, breathe out'
  end
end
```

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 can represent as follows:

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



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

```
fido = Dog.new
fido.breathe
fido.bark
```

Inheritance only goes one way

Any method that is defined in the **`Mammal`** class can also be invoked on an object of the **`Dog`** class. New methods 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!

In Ruby, instance variables are not inherited in the same way they are in Java.

The rule in Ruby is very simple:

An instance variable is dynamically added to an object when it is first referenced.

Adjust your thinking! We're not in Java anymore.

In Ruby, **classes** don't have instance variables like in Java. Instance variables are added to **objects** as the program executes.

Example of instance variable creation

```
class Person
  def initialize(name)
    @name = name
  end

  def buy_house
    @house = House.new
  end
  #...
end

john = Person.new('John')

john.buy_house
```

Q: How many instance variables has this class?

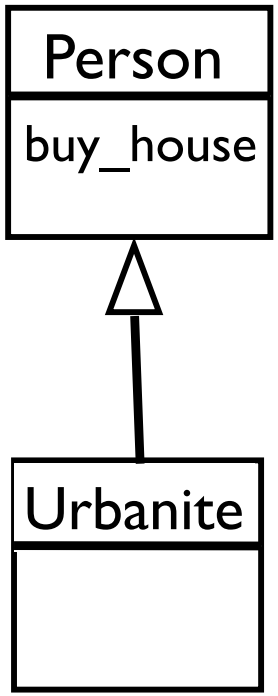
A: Classes don't have instance variables!

john has one instance variable

Now **john** has two instance variables

Instance variables and Inheritance

```
class Person
  def initialize(name)
    @name = name
  end

  def buy_house
    @house = House.new
  end
  #…
end

class Urbanite < Person
  ...
  def foobar
    # How to access @house here?
  end
  ...
end
```



Instance variables in a class hierarchy

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

```
@count = 10
```

what will happen?

1. A new **`Fixnum`** object is created with the value **`10`**,
2. The current object gets a new instance variable, **`@count`**,
3. **`@count`** is set to refer to the new **`Fixnum`** object.

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 initializer).

Invoking the superclass initializer

```
class Person
  def initialize(name)
    @name = name
  end
  # …
end

class Student < Person
  def initialize(name, number)
    super(name)
    @number = number
  end
  # …
end
```

super invokes the method of the same name, but starts the lookup in the superclass

Invoking **super** in a method suggests a well-designed class hierarchy, as it implies a semantic coherence between the methods.

Will `my_student` be initialised correctly?

```
class Person
  def initialize(name)
    @name = name
  end
  # …
end

class Student < Person
  # No initialise method!
  # ...
end

my_student = Student.new('Aoife')
```

Yes! The **`initialize`** method is bound just like any other method.

Terminology: what's a **client**?

Aside I

A **client** of a class is any other class that uses it.
Here the class **`Alpha`** is a client of the class **`Beta`**.

```
class Alpha
  def initialize
     ...
  end

  def some_method
     …
    @my_beta = Beta.new
    ...
  end
end
```

In general, a class never knows who its clients are.

A subclass is a special type of client of its superclass.

How can a class "**know**" something?

When we say a class "knows" something, we mean that this information is evident from the code of the class, e.g.

```
class Alpha
    ...
  def some_method
    ...
    @my_beta = Beta.new
    beta.foo
    ...
  end
end
```

The less a class knows about other classes, the better.

Here we'd say that **`Alpha`** "knows" that **`Beta`** is a class and that objects of **`Beta`** have a **`foo`** method that takes no arguments.

Public, Private and Protected

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

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

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

Accessing a private field would not be allowed in Java

If you want to grant **clients** of the class access to an instance variable, use an **attribute reader/writer/accessor** as seen in the Ruby Tour.

Private Methods

Methods are **public** by default. If we make a method **private**, it can only be invoked from *inside* the object.

Thus, a private method is also 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

Why would you make a method private?

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

A private method cannot be invoked on an explicit object.

Assuming foo is private, this is the correct way to invoke it:

```
foo
```

i.e., the receiving object is implicitly the current object.

Both of these are incorrect:

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

Initializers are normal, private methods

When an object instance is created using e.g.,

```
Person.new
```

the **`initialize`** method is invoked on the newly-created **`Person`** 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 mainly used in creating a comparison method (i.e. overriding the == operator). Other than that, they are not very common.

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();  // Error! priv() 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 in a class is visible to subclasses.
=> a subclass is potentially very **tightly coupled** to its superclass. More on coupling later in the module.

Ruby access rights summarised

In **Ruby**, what access do the various categories of class have to the public, protected and private methods of a class? The following table summarises the rules:

	public	protected	private
Inside class	✓	✓	✓
Subclasses	✓	✓	✓
Client object (same class)	✓	✓	
Client object (different class)	✓		

Java access rights summarised

In **Java**, what access do the various categories of class have to the public, protected and private methods/fields of a class? The following table summarises the rules:

	public	protected	private
Inside class	✓	✓	✓
Subclasses	✓	✓	
Client object (same class)	✓	✓	✓
Client object (different class)	✓		

Banking Example

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

"A bank account stores the name of 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 interest to be added to the balance."

Finding Classes — highlight the nouns

A **bank account** stores the **name** of 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(name)
    @name = name
    @balance = 0.0
  end

  def deposit(amount)
    @balance += amount
  end

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

SavingsAccount class

```
class SavingsAccount < BankAccount

  def initialize(name, interest_rate)
    super(name)
    @interest_rate = interest_rate
  end

  def apply_interest
    @balance += (@interest_rate/100.0) * @balance
  end

end
```

(What's wrong with doing it this way?)

```
class SavingsAccount < BankAccount

  def initialize(name, interest_rate)
    @name = name
    @balance = 0.0
    @interest_rate = interest_rate
  end

  def apply_interest
    @balance += (@interest_rate/100.0) * @balance
  end

end
```

Overriding

Now we want a Special Savings Account (an SSA) that penalises withdrawals (but e.g. provides a higher rate of interest).

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

We'll also add the penalty as an argument to the initializer.

SpecialSavingsAccount class

```
class SSA < SavingsAccount
  def initialize(name, interest_rate, penalty)
    super(name, interest_rate)
    @penalty = penalty
  end

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

has same meaning as **`super(amount)`**

If a method in a subclass has the same name as one in a superclass, it **overrides** it, with the same meaning as in Java.

More on this when we look at **polymorphism**.

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

Inheritance is a technique for creating a new class based on an existing class.

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

The next topic we look at is **Software Quality**.

We'll return to inheritance later in the context of **type systems** and **polymorphism**.