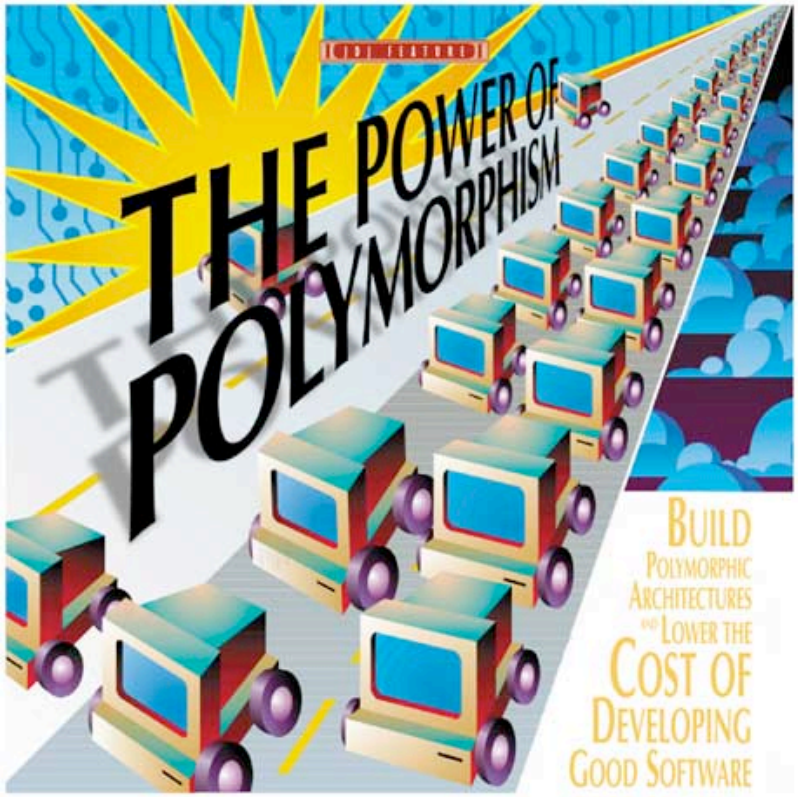
Polymorphism

Polymorphism ('many shapes') is probably the most important software reuse mechanism in the object-oriented approach.

It's been an established part of mainstream software practice since the early/mid nineties.

This is one of the central topics of object-oriented programming.



Polymorphism in Ruby

Ruby supports polymorphism.

It's hard to program in Ruby in a non-polymorphic fashion.

Polymorphism Definition

A variable or method argument is said to be **polymorphic** if it is permitted to hold values of different **types** during execution.

(So all Ruby variables are polymorphic. How about Java variables?)

One precise definition of a polymorphic method now runs as follows:

> **"A polymorphic method is one that has polymorphic arguments."**

Another one is:

> **"Polymorphism is the ability of heterogeneous objects to respond to the same message."**

Real-World Polymorphism

How would you implement in Java?

A Java Implementation

```
abstract class Animal{
    public abstract void speak();
}

class Cat extends Animal{
    public void speak(){System.out.println("Meow!");}
}

class Dog extends Animal{
    public void speak(){System.out.println("Woof!");
    }
}

class AnimalLover{
    public static void main(String args[]){
        Animal animal = new Dog();
        // ...
        animal.speak();
    }
}
```

In Java, Inheritance is a pre-requisite for polymorphism.

A Ruby Implementation

```
class Cat
  def speak
    puts "Meow!"
   end
end

class Dog
  def speak
    puts "Woof!"
   end
end


animal = Dog.new
# ...
animal.speak
```

In Ruby, you can use polymorphism without inheritance.

What Dynamic Binding means

```
class Cat
  def speak
    puts "Meow!"
   end
end

class Dog
  def speak
    puts "Woof!"
   end
end


if rand(2)==1 # toss a coin
   animal = Dog.new
else
   animal = Cat.new
end
animal.speak
```

Which method is invoked?

Message passing, not method invocation

Consider the statement on the previous slide:

```
animal.speak
```

This is often referred to as a **method invocation**.

However, we can't tell which method gets invoked...

It's better to think of this as a **message send**. We send the **`animal`** object the message **`speak`**, and it decides which method to invoke (depending on what type of object it is).

Gary Larson on Static Types

"Now! *That* should clear up
a few things around here!"

Static typing can get in the way

Duck Typing

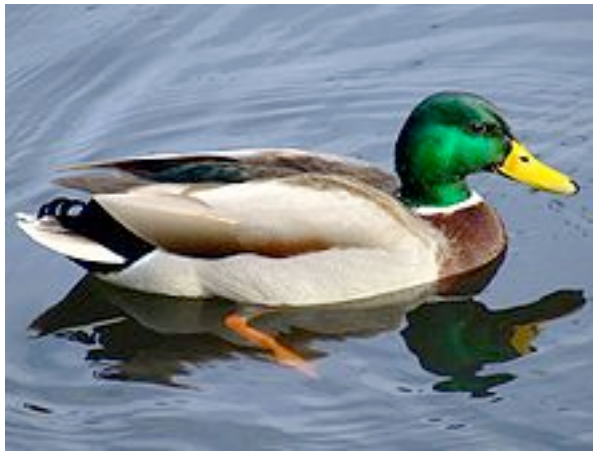
Ruby is dynamically typed.

Types are not checked until the code is executed.

By contrast, Java/C++ both perform extensive type checks as part of compilation.

Ruby uses what is commonly called **Duck Typing**.

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." -- James Whitcomb Riley



Can you identify this bird?

Static vs. Dynamic Typing

Static typing aims to catch as many errors as possible at compile-time. Traditional wisdom is that this is required for serious, production development.

Dynamic typing is better for quick development, but more error-prone. Traditionally seen as more suitable for rapid prototyping or non-production development.

The current shift is certainly in favour of dynamically-typed languages,

Uses of Polymorphism

There are many different ways in which polymorphism can be used. We look at three main examples here:

1. **Overriding**

2. **Abstract Method** (aka Deferred Method)

3. **Template Method**

Overriding

A superclass provides an implementation for its methods. This implementation then becomes the default implementation for all the subclasses of this superclass.

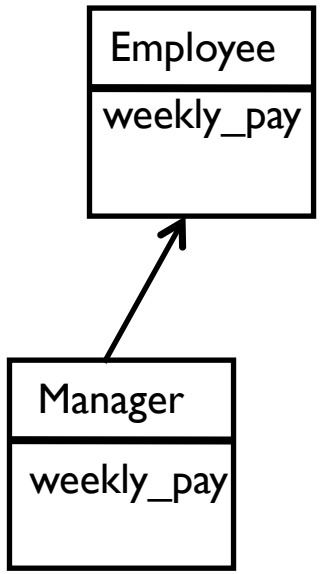
In some cases this is not what we want. We may want to redefine the method in the subclass so that it does something different or more efficient.

We look at two examples…

Employee Example

```
class Employee
   ...
   def weekly_pay
     @hours_worked*@hourly_rate
   end
end

class Manager < Employee
   ...
   def weekly_pay
      super + @bonus
   end
end
```

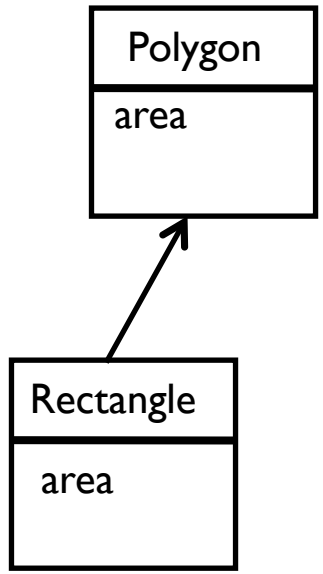


`Manager` overrides **`weekly_pay`** in **`Employee`** to do the correct thing for a **`Manager`** object.

Shapes Example

```
class Polygon
   def area
      # compute polygon area
   end
   ...
end

class Rectangle < Polygon
   def area
      length * breadth
   end
    ...
end
```



`Rectangle` overrides **`area`** in **`Polygon`** with a faster implementation.

Abstract Methods

Overriding is useful when we want to define default behaviour for a set of classes but allow each class to redefine this behaviour if needs be.

Sometimes we simply want to state that any objects of a class's subclasses must be able to respond to a particular message, without giving any default behaviour.

This is achieved in Java using an **abstract method with no body**. It is achieved in C++ using a **pure virtual function**.

In Ruby, you don't need abstract classes and you don't need abstract methods, so we look at a Java example...

Abstract Methods in Java

For example, a **`Shape`** class might want to insist that all **Shape** objects can respond to the **`draw`** message.

At the same time, there's no point in trying to define default **`draw`** behaviour for the **`Shape`** class.

In Java we would define:

```
abstract class Shape{
    ...
    abstract void draw();
    ...
}
```

Abstract Methods in Ruby

OK, I said Ruby doesn't support this concept, but sometimes people implement this by throwing an exception if the method is called:

```
class Shape{
    ...
    def draw
        raise 'Called abstract method draw'
    end
    ...
}
```

A: Don't do this.
B: If you really want to do this, use Java.

Template Method

A **Template Method** is a polymorphic method can truly work with objects of different types. We haven't quite achieved this yet.

In the overriding examples, we still have to write a completely new method whenever we wanted to override a method in the superclass.

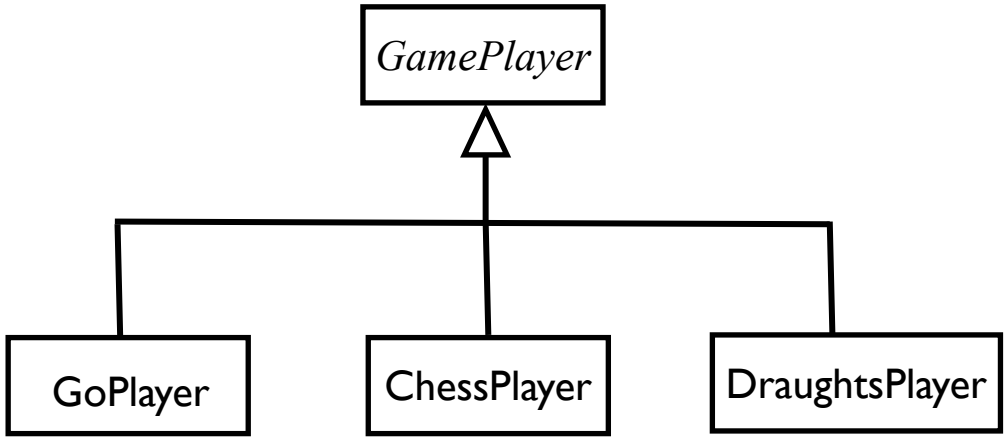
We aren't yet able to write a method that can work with objects of different types and at the same time do the appropriate thing with each type of object.

P.S. Template Method is also an example of a ***Design Pattern****. These are covered further in Comp 30160 Object-Oriented Design.*

Modelling 2-Player Boardgames

Consider building a simulation of various 2-player games such as chess, draughts, go etc. It's natural to model the players as objects, so we can identify immediately the classes **`ChessPlayer`**, **`DraughtsPlayer`** and **`GoPlayer`**.

It's also natural to generalise these classes to produce a superclass, **`GamePlayer`**. One obvious method is **`play`**, which plays whatever game this object specialises in.



Defining the `play` method

What technique do we use to define the method **`play`**? Consider these two possibilities:

> **Abstract Method**: Here we are stating that the ways these different players play have absolutely nothing in common with each other, and that's not true.
>
> **Overriding**: The question here is this: what default implementation of play do we provide? There's isn't really a suitable default implementation; each game is played in its own particular way.

Neither of these techniques quite matches the situation we face.

Consider the algorithm on the next slide…

An algorithm for `play`

```
class GamePlayer
   ...
   def play
      set_up_game
      if play_first
         make_best_move
      end
      while !game_over
         await_opponents_move
         if !game_over
            make_best_move
         end
      end
      decide_winner
   end
   ...
end
```

Looks like an algorithm, but it's real Ruby code; it's a **Template Method**.

Defining **`ChessPlayer`** is easy

```
class ChessPlayer < GamePlayer

   def set_up_game
     Place white queen on d1, etc.
   end

   def game_over
      stalemate || checkmate || mate_impossible
   end

   def play_first
      # Toss a coin
   end

   ...
   ...
end
```

More on the **`ChessPlayer`** subclass

The methods referred to in the **`play`** method of the **`GamePlayer`** class must be implemented. The method **`play`** then operates correctly for a **`ChessPlayer`**.

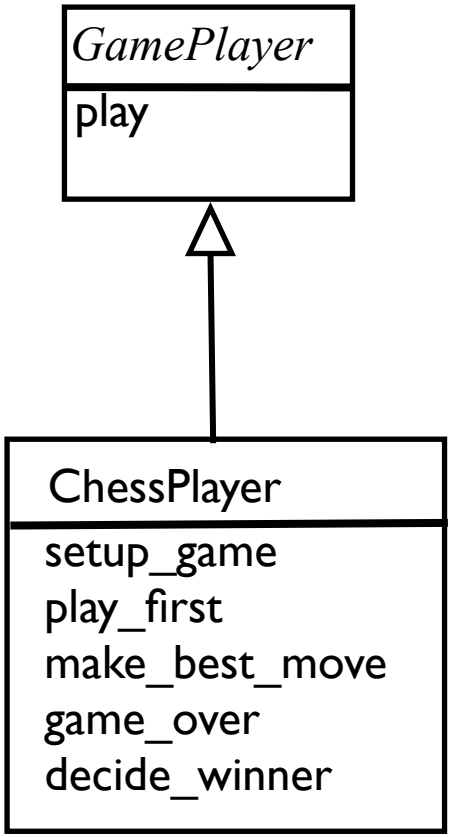
Consider the template method **`play`** in **`GamePlayer`**. It can be used in all of its subclasses without being overridden.

Each subclass simply overrides the **`set_up_game`**, **`game_over`**, **`make_best_move`** and **`decide_winner`** methods to define their own idiosyncratic game playing behaviour.

A Template Method in action

Consider how the play method will execute for an object of the **`ChessPlayer`** class...

```
def play
   set_up_game
   if play_first
      make_best_move
   end
   while !game_over
      await_opponents_move
      if !game_over
         make_best_move
      end
   end
   decide_winner
end
```



Features of Template Method

The superclass has a skeletal method, or algorithm, that invokes several other methods. This skeleton is the **Template Method**.

The subclasses inherit the Template Method as is, and implement each of the abstract methods it invokes.

When **`template_method`** is invoked on an instance of a concrete subclass:

1. **`template_method`** is not found in the subclass, it is found in the superclass so that method is executed.

2. Whenever **`template_method`** invokes a method on itself, the search for the method to execute starts in the concrete subclass.

3. As a template method executes, control jumps between the template method defined in superclass and the overridden methods defined in the subclass.

Hook Methods

A hook method is one that is invoked in a Template Method and has an implementation in the superclass, but that is intended to be overridden in some subclasses.

In the **`GamePlayer`** class, **`await_opponents_move`** could be implemented as a hook method, one that does nothing except wait.

Hook methods are often implemented to do nothing. However, unlike abstract methods, they do not have to overridden. They can usually be private.

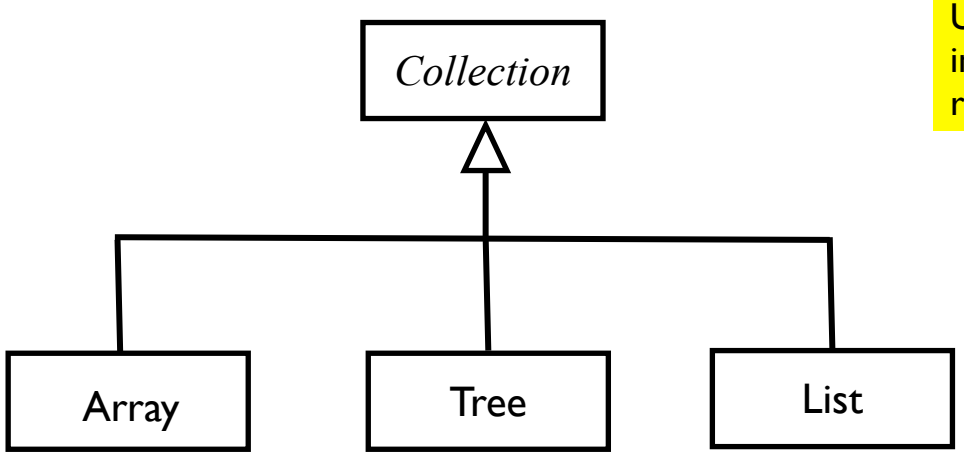
As the programmer writing the concrete subclass, you need to find out what hook methods are provided by the superclass -- these are the points of variability in the Template Method that you **may change** if you wish.

A Search Example

Now a more typical software example.

Consider a class **`Collection`**, that represents a collection of records. Such a collection could be organised in many ways, as a tree, a list (sorted or unsorted), an array etc.

Each of these will be represented by a subclass of **`Collection`**.



Using inheritance to reflect implementation differences is not recommended. See later in module.

Defining the **Search** method

Consider the **`search`** method defined in the class **`Collection`**. If we make this an abstract method, we miss an opportunity to capture at a high level what searching routines have in common.

Consider this:

```
class Collection
   def search? record
      pos = initialPosition(record)
      while !exhausted(pos) && !found(pos,record)
         pos = next(pos,record)
      end
     !exhausted(pos)
   end
end
```

Questions about this `Search` method

Three questions:

Is this search routine correct? Is it as general as it should be? Should you make it more flexible? If so, how?

How would you define the abstract methods to create a search for an *unsorted* array?

How would you define the abstract methods to create an improved search for a *sorted* array?

Summary

Polymorphism is a vital tool in current programming practice. It enable us to reuse high-level abstractions, rather than just low-level ones.

Use **overriding** when a method can be given a default implementation that can overridden by subclasses if required.

Use an **abstract method** when there is no sensible default implementation for a method, but you want all concrete subclasses to implement it.

Use a **template method** when methods have a shared structure, i.e., the same algorithm and you to model this.

Writing polymorphic code can be difficult, and it takes experience to spot when to do so.