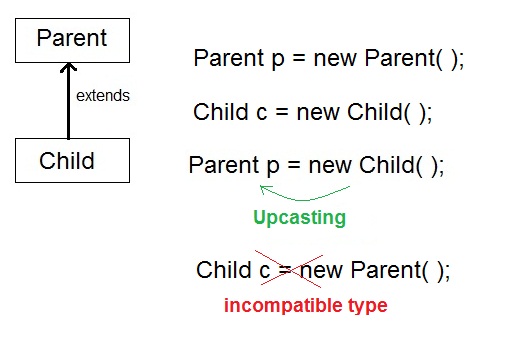
Dynamic Dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.

  
**Example**

class **Game**

{

public void type()

{

System.out.println("Indoor & outdoor");

}

}

Class **Cricket** extends **Game**

{

public void type()

{

System.out.println("outdoor game");

}

public static void main(String[] args)

{

Game gm = new Game();

Cricket ck = new Cricket();

gm.type();

ck.type();

**gm=ck;**  **//gm refers to Cricket object**

gm.type(); **//calls Cricket's version of type**

}

}

**Output:**

Indoor & outdoor

Outdoor game

Outdoor game

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **dynamic dispatch** is the process of selecting which implementation of a polymorphic operation ([method](https://en.wikipedia.org/wiki/Method_(computer_programming)) or function) to call at [run time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)). Dynamic dispatch contrasts with static dispatch in which the implementation of a polymorphic operation is selected at [compile-time](https://en.wikipedia.org/wiki/Compile-time). The purpose of dynamic dispatch is to support cases where the appropriate implementation of a polymorphic operation can't be determined at compile time because it depends on the runtime type of one or more actual parameters to the operation.

Dynamic dispatch is different from [late binding](https://en.wikipedia.org/wiki/Late_binding) (also known as dynamic binding).

**Late Binding**: type is unknown until the variable is exercised during run-time; usually through assignment but there are other means to coerce a type; dynamically typed languages call this an underlying feature, but many statically typed languages have some method of achieving late binding

Implemented often using [special] dynamic types, introspection/reflection, flags and compiler options, or through virtual methods by borrowing and extending dynamic dispatch

**Early Binding**: type is known before the variable is exercised during run-time, usually through a static, declarative means

Implemented often using standard primitive types

## **Functions**

**Static Dispatch**: known, specific function or subroutine at compile time; it is unambiguous and matched by the signature

Implemented as static functions; no method can have the same signature

**Dynamic Dispatch**: not a specific function or subroutine at compile time; determined by the context during execution, in other words the input parameter type or types helps determine which function to call; few languages are multiple [dynamic] dispatch, where more than one parameter can differ; overloading is possible because of this mechanism; see also virtual method table

Implemented as virtual or abstract functions; other clues include overriddes, hidden, or shadowed methods

## **Values**

**Lazy Loading**: object initialization strategy that defers value assignment until needed; allows an object to be in an essentially valid but knowingly incomplete state and waiting until the data is needed before loading it; often found particularly useful for loading large datasets or waiting on external resources

Implemented often by purposefully not loading a collection or list into a composite object during the constructor or initialization calls until some downstream caller asks to see the contents of that collection (eg. get\_value\_at, get\_all\_as, etc). Variations include loading meta information about the collection (like size or keys), but omitting the actual data; also provides a mechanism to some runtimes to provide developers with a fairly safe and efficient singleton implementation scheme

**Eager Loading**: object initialization strategy that immediately performs all value assignments in order to have all the data needed to be complete before considering itself to be in a valid state.

Implemented often by providing a composite objects with all their known data as soon as possible, like during a constructor call or initialization

**Data Binding**: often involves creating an active link or map between two compatible information streams so that changes to one are reflected back into the other and vice versa; in order to be compatible they often have to have a common base type, or interface

Implemented often as an attempt to provide cleaner, consistent synchronization between different application aspects (eg view-model to view, model to controller, etc.) and talks about concepts like source and target, endpoints, bind/unbind, update, and events like on\_bind, on\_property\_change, on\_explicit, on\_out\_of\_scope

**In short, late binding refers to the object-side of an eval, dynamic dispatch refers to the functional-side. In late binding the type of a variable is the variant at runtime. In dynamic-dispatch, the function or subroutine being executed is the variant.**

Single Dispatch

In short, single dispatch is when a method is polymorphic on the type of one parameter (including the implicit this). Double dispatch is polymorphism on two parameters.

# Single, Double And Multiple Dispatch

This post has already been published on [**code::gallery**](http://codegallery.blogsome.com/2006/07/15/overcautions-coding/) blog which now has been[**merged**](http://ifacethoughts.net/2006/07/23/merging-blogs/) into this blog.

These are mechanisms in object oriented programming languages to identify the function/method to be invoked. The dispatch in the nomenclature is about dispatching messages to objects, as it is said in Smalltalk. It is equivalent of saying invoking methods of an object.

### Single Dispatch

Typically, multiple methods or functions are given the same name, because the represent the same purpose. In the single dispatch mechanism, the method to be invoked is determined using the object, usually type of the object, on which it is invoked. It also includes the parameters, but the parameter types are identified at the compile time whereas dynamic binding or dynamic dispatch can be used for object on which method is invoked. This object is also syntatically highlighed, like obj.behave(the, arguments).

Most of the conventional and popular languages, like C++, Java or Smallatalk inherently support single dispatch mechanism.

### Double Dispatch

Why is anything more than single dispatch is being considered? Because in the real world it is required. In the real world, the behavior between two objects is dependent on both of them and not just one. Lets consider an example.

Your behavior would change when you face other humans, the domestic cat or the tiger. This means that your actions are dependent not only on you but also on whom you face. This cannot be incorporated using the single dispatch mechanism.

So we come up with double dispatch. It is in fact a simulation using the single dispatch mechanism, and hence is not completely extensible.

The following code might not work, as pointed out by a commenter. Instead you can refer to [**Dr. Carlo Pescio’s explanation**](http://www.eptacom.net/pubblicazioni/pub_eng/mdisp.html) for more on this. It also goes a step further to improve the solution using templates.

I will try to update the following code to get it working.

Consider the following example code:

class Human;

class Cat;

class Animal

{

public:

virtual void face (Animal& animal);

virtual void face (Human& human);

virtual void face (Cat& cat);

}

class Human : public Animal

{

virtual void face (Animal& animal)

{

animal.face(\*this);

}

virtual void face (Human& human);

{

// shakehand

}

virtual void face (Cat& cat);

{

cat.face(\*this);

}

}

class Cat : public Animal

{

virtual void face (Animal& animal)

{

animal.face(\*this);

}

virtual void face (Human& human);

{

human.face(\*this);

}

virtual void face (Cat& cat);

{

// run

}

}

This code works. What is done here is that two calls are used to identify both the types involved. Consider this code:

Animal& acat = new Cat();

Animal& ahuman = new Human();

ahuman.face(acat);

Here, when ahuman.face(acat) is invoked, it in turn invokes Cat::face(Human&) at which point both the types are determined. This is the double dispatch mechanism.

However, as you can see, the biggest disadvantage is that the base class Animal has to know all the derived classes. Everytime a new animal is added, the interface of Animal has to change making it impractical, exactly what the [**Dependency Inverstion Principle**](http://ifacethoughts.net/2006/03/16/dependency-inversion-principle-and-interface/) advises us to avoid.

### Multiple Dispatch

So we need multi dispatch, also called multimethods. The multidispatch mechanism considers all parameters equally and hence can provide easier and more extensible implementations. Some of the languages that support multiple dispatch are [**Common Lisp**](http://clisp.cons.org/), [**Dylan**](http://www.opendylan.org/), [**Nice**](http://nice.sourceforge.net/), [**Scheme**](http://www-swiss.ai.mit.edu/projects/scheme/) and [**Slate**](http://slate.tunes.org/).

One of the common designs of multiple dispatch is to separate the methods from the class (which contains the structure). This allows for treating all the parameters equally.