Single, Dynamic, Multiple and Double Dispatching

**What is Dispatching?**

Method dispatching is basically an algorithm used to decide which method should be invoked in response to a certain message.

* Single/Dynamic Dispatching
* Multiple Dispatching
* Double Dispatching

**Single/Dynamic Dispatching**

class Messenger {

public void send(String message) { }

}

Messenger messenger = new Messenger();

messenger.send("Hello World");

These are all single dispatch examples. No dynamic dispatching needed so far. However, if you have a polymorphic structure as in:

**interface** Messenger {  
 **void** send(String message);  
}

**public class** TextMessenger **implements** Messenger {  
 **public void** send(String message) {  
 System.***out***.println(**"TEXT: "** + message);  
 }  
}

**public class** XMLMessenger **implements** Messenger {  
 **public void** send(String message) {  
 System.***out***.println(**"<message>"** + message + **"</message>"**);  
 }  
}

**class** Demo {

**public void** useMessenger(Messenger messenger) {  
 messenger.send(**"Which messenger am I using?"**);  
 }  
}

As you may quickly notice, this requires dynamic dispatching (run-time decision) since compiler cannot decide the actual type of the messenger which Demo.useMessenger uses. Lots of known design patterns are already based on this mechanism (Strategy, Bridge, etc..).

### Multiple Dispatching

**interface** Messenger {  
 **void** send(String message);  
}

**public class** TextMessenger **implements** Messenger {  
 **public void** send(String message) {  
 System.***out***.println(**"TEXT: "** + message);  
 }  
}

**public class** XMLMessenger **implements** Messenger {  
 **public void** send(String message) {  
 System.***out***.println(**"<message>"** + message + **"</message>"**);  
 }  
}

**class** Demo {  
 **public void** useMessenger(Messenger messenger) {  
 messenger.send(**"Which messenger am I using?"**);  
 }  
 **public void** useMessenger(TextMessenger messenger) {  
 messenger.send(**"I'm TextMessenger for sure, but?"**);  
 }  
  
 **public static void** main(String[] args) {  
 Messenger msgr = **new** TextMessenger();  
 TextMessenger txtmsgr = **new** TextMessenger();  
 **new** Demo().useMessenger(msgr); 🡺 TEXT: Which messenger am I using?  
 **new** Demo().useMessenger(txtmsgr); 🡺 TEXT: I'm TextMessenger for sure, but?

}  
}

**It happens in java.** BTW, Java doesn't support true multiple dispatching, so don't cause any bugs while creating overloaded methods.

### Double Dispatching

Visitor pattern is actually good way of explaining this topic. Let's look at the following example:

// messenger

interface Messenger {

void send(Message message);

}

class TextMessenger implements Messenger {

public void send(Message message) {

// some custom operations

message.print(this);

}

}

class XMLMessenger implements Messenger {

public void send(Message message) {

// some other custom operations

message.print(this);

}

}

// and message

class Message {

public void print(TextMessenger messenger) {

// do something

}

public void print(XMLMessenger messenger) {

// do some other thing

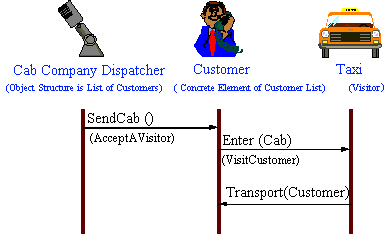
}

}

As you can easily see, objects are related to each-other both ways. Message is deciding to do something based on other types. Again, for more information you can refer **Double Dispatching** definition.

**What is the difference between Multiple Dispatch and Overloading?**

The Visitor pattern represents an operation to be performed on the elements of an object structure without changing the classes on which it operates. This pattern can be observed in the operation of a taxi company. When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]



<http://stackoverflow.com/questions/1801216/what-is-the-difference-between-multiple-dispatch-and-method-overloading>

**Method overloading is resolved at compile time.**

**Multiple dispatch is resolved at runtime.**

When using double dispatch the called method depends on the actual type of receiver and arguments. Method overloading however, only allows the called method to depends on the declared type of the parameters. Why? Java binds method calls at compile time with their full signature (early binding). The full signature includes all parameter types, hence when the actual type of an argument differs at runtime (polymoprhism), overloading does not work as you might expect!

void add(Foo o) { ... }

void add(Bar o) { ... }

void client() {

Foo o = new Bar();

add(o); // calls add(Foo) not add(Bar)!

}

using multiple dispatch however

void add(Foo o) { o.dispatch(this); }

void add(Bar o) { o.dispatch(this); }

void client() {

Foo o = new Bar();

add(o); // calls #dispatch as defined in Bar!

}

**Double Dispatch**

<http://c2.com/cgi/wiki?DoubleDispatchExample>

**Problem:**

Assume we are writing a print library. We want the client code of our library to be able to do something like this:

class Client {

/\*\* Prints all figures on each of the printers. \*/

void printAllEverywhere( Figure[] figures, Printer[] printers ) {

for ( int i = 0; i < figures.length; i++ ) {

Figure figure = figures[ i ];

for ( int j = 0; j < printers.length; j++ ) {

Printer printer = printers[ j ];

figure.printOn( printer );

// must work for any printer or figure !

}

}

}

}

Of course, when we add new printers or figures to our library, we want client code such as the above to immediately work with these new printers and figures, without a change in their code or even a recompile. How can we implement our library to allow this?

Another way to state this is that we need a [CartesianProduct](http://c2.com/cgi/wiki?CartesianProduct) of drivers or methods of printer kinds and shape kinds.

**Solution:**

First, we define two interfaces for our figures and printers:

interface Figure {

void printOn( Printer printer );

}

interface Printer {

void printCircle( Circle circle );

void printRectangle( Rectangle rectangle );

}

Next, we write our two printers:

class InkjetPrinter implements Printer {

public void printCircle( Circle circle ) {

// ... rasterizing logic for inkjet printing of circles here ...

System.out.println( "Inkjet printer prints a cirlce." );

}

public void printRectangle( Rectangle rectangle ) {

// ... rasterizing logic for inkjet printing of rectangles here ...

System.out.println( "Inkjet printer prints a rectangle." );

}

}

class PostscriptPrinter implements Printer {

public void printCircle( Circle circle ) {

// ... postscript preprocessing logic for circles here ...

System.out.println( "PostScript printer prints a cirlce." );

}

public void printRectangle( Rectangle rectangle ) {

// ... postscript preprocessing logic for rectangles here ...

System.out.println( "PostScript printer prints a rectangle." );

}

}

Now, all we have to make sure is that calling the *figure.printOn( printer )* method results in the correct *printXyz* implementation being executed, such as *postscriptPrinter.printRectangle( rectangle )*.

This can be achieved through a simple indirection in the implementation of *printOn* in the individual figure classes:

class Circle implements Figure {

public void printOn( Printer printer ) {

printer.printCircle( this ); // <-- the "trick" !

}

}

class Rectangle implements Figure {

public void printOn( Printer printer ) {

printer.printRectangle( this );

}

}

That's it!

To test the above code, just add the following class:

public class Main {

public static void main( String[] args ) {

Figure [] figures = new Figure [] {

new Circle(), new Rectangle() };

Printer [] printers = new Printer [] {

new PostscriptPrinter(), new InkjetPrinter() };

new Client().printAllEverywhere( figures, printers );

}

}

**Explanation:**

What happens when *figure.printOn( printer )* is called at runtime? This depends on the types of the objects referenced by the *figure* and *printer* variables at that time. Let's assume that at the moment, *figure* points to an instance of class Circle, and *printer* to an InkjetPrinter. So, the implementation of *printOn( printer )* being called will be that of the circle instance, defined in the Circle class (first dispatch). This method just contains one line: *printer.printCircle( this )*, which delegates the work to the printer object. As this object is of the class InkjetPrinter, the*printCircle( circle )* method of the InkjetPrinter class will be executed (second dispatch), which is just what we wanted.

**Visitor Design Pattern Example**

<https://sourcemaking.com/design_patterns/visitor/java/2>

**public** **class** **VisitorSingle** {

**interface** Base {

**void** **process1**( Base secondObject );

**void** **process2**( A firstObject );

**void** **process2**( B firstObject );

**void** **process2**( C firstObject );

}

**static** **class** **A** **implements** Base {

**public** **void** **process1**( Base second ) { second.process2( **this** ); }

**public** **void** **process2**( A first ) {

System.out.println( "first is A, second is A" ); }

**public** **void** **process2**( B first ) {

System.out.println( "first is B, second is A" ); }

**public** **void** **process2**( C first ) {

System.out.println( "first is C, second is A" ); }

}

**static** **class** **B** **implements** Base {

**public** **void** **process1**( Base second ) { second.process2( **this** ); }

**public** **void** **process2**( A first ) {

System.out.println( "first is A, second is B" ); }

**public** **void** **process2**( B first ) {

System.out.println( "first is B, second is B" ); }

**public** **void** **process2**( C first ) {

System.out.println( "first is C, second is B" ); }

}

**static** **class** **C** **implements** Base {

**public** **void** **process1**( Base second ) { second.process2( **this** ); }

**public** **void** **process2**( A first ) {

System.out.println( "first is A, second is C" ); }

**public** **void** **process2**( B first ) {

System.out.println( "first is B, second is C" ); }

**public** **void** **process2**( C first ) {

System.out.println( "first is C, second is C" ); }

}

**public** **static** **void** **main**( String[] args ) {

Base array[] = { **new** A(), **new** B(), **new** C() };

**for** (**int** i=0; i < array.length; i++)

**for** (**int** j=0; j < 3; j++)

array[i].process1( array[j] );

}}

#### Output

first is A, second is A

first is A, second is B

first is A, second is C

first is B, second is A

first is B, second is B

first is B, second is C

first is C, second is A

first is C, second is B

first is C, second is C

[Differences between Dynamic Dispatch and Dynamic Binding](http://programmers.stackexchange.com/questions/235772/differences-between-dynamic-dispatch-and-dynamic-binding)

<http://programmers.stackexchange.com/questions/235772/differences-between-dynamic-dispatch-and-dynamic-binding>

Under the covers, Dynamic Dispatch and Dynamic Binding may work out the same. But the idea in dynamic dispatch is following some function pointer to see which method to actually invoke, or object to invoke it on. **"Binding" is the idea that the method is "bound" to a particular instance (or class of instances)** & that's how you identify it.

So they could work together -- a method that's bound to an object with dynamic binding might use dynamic dispatch when you call it.

**Also dynamic dispatch has more of an OO flavor to it... it's the mechanism behind polymorphism, in which a reference to an object might point to one of multiple implementations.** Dynamic dispatch decides at runtime which one to actually run. By contrast, late binding would be dropping in whole new methods that weren't there at compile time.

**Dynamic dispatch is different from late binding (also known as dynamic binding).** In the context of selecting an operation, binding refers to the process of associating a name with an operation. Dispatching refers to choosing an implementation for the operation after you have decided which operation a name refers to. With dynamic dispatch, the name may be bound to a polymorphic operation at compile time, but the implementation not be chosen until runtime (this is how dynamic dispatch works in C++). However, late binding does imply dynamic dispatching since you cannot choose which implementation of a polymorphic operation to select until you have selected the operation that the name refers to.

<http://stackoverflow.com/questions/533330/dynamic-dispatch-and-binding>

Dynamic Dispatch - The actual method group/signature/override chain is bound at compile time. The method called is dependent upon the actual runtime type of the object but no actual interpretation occurs. It will still be a version of the statically bound method.

Dynamic dispatch or dynamic binding means that when calling a method, choosing the actual method implementation to execute happens while the program is running, because statically there is not enough information available. It will based on the method name, the actual receiver type (subtype polymorphism), and/or the actual argument types (overloading), or even more fancy pattern matching.

**Double Dispatch is a Code Smell**

<https://lostechies.com/derekgreer/2010/04/19/double-dispatch-is-a-code-smell/>

If you’re using [Double Dispatch](http://en.wikipedia.org/wiki/Double_dispatch) in your code, this may be a symptom of an underlying design issue which may impact the maintainability of your application.  Due to the fact that Double Dispatch is at times confused with a form of the [Strategy Pattern](http://en.wikipedia.org/wiki/Strategy_Pattern), an overview may be in order to elaborate on this assertion further.

**What is Double Dispatch?**

Technically, Double Dispatch refers to a technique used in the context of a polymorphic method call for mitigating the lack of [multimethod](http://en.wikipedia.org/wiki/Multimethods) support in programming languages.  **More simply, Double Dispatch is used to invoke an overloaded method where the parameters vary among an inheritance hierarchy**.  To explain fully, let’s start with a review of [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming).

**Polymorphism**

In the following example, a hierarchy of shapes are defined with each of the derived types overloading a base virtual Draw() method.  Next, a console application is used to define a list of each of the shapes and iterate over each shape in the collection calling the Draw() method of each item in the list:

import java.util.ArrayList;  
import java.util.List;  
class Shape  
 {  
 public void Draw()  
 {  
 System.*out*.println("A shape is drawn.");  
 }  
 }  
  
 class Polygon extends Shape  
 {  
 public void Draw()  
 {  
 System.*out*.println("A polygon is drawn.");  
 }  
 }  
  
 class Quadrilateral extends Polygon  
 {  
 public void Draw()  
 {  
 System.*out*.println("A quadrilateral is drawn.");  
 }  
 }  
  
 class Parallelogram extends Quadrilateral  
 {  
 public void Draw()  
 {  
 System.*out*.println("A parallelogram is drawn.");  
 }  
 }  
  
 class Rectangle extends Parallelogram  
 {  
 public void Draw()  
 {  
 System.*out*.println("A rectangle is drawn.");  
 }  
 }  
  
 class Program  
 {  
 public static void main(String[] args)  
 {  
 List<Shape> shapes = new ArrayList<Shape>();  
 shapes.add( new Shape() );  
 shapes.add( new Polygon() );  
 shapes.add( new Quadrilateral() );  
 shapes.add( new Parallelogram());  
 shapes.add( new Rectangle());  
  
  
 for(Shape shape : shapes)  
 {  
 shape.Draw();  
 }  
 System.*out*.println("................................");  
 }  
 }

The following lines are printed to the console upon running the application:

A shape is drawn.

A polygon is drawn.

A quadrilateral is drawn.

A parallelogram is drawn.

A rectangle is drawn.

Note that the proper Draw() method is called for each item in the collection.  In most object-oriented languages, this polymorphic behavior is achieved through the use of a [virtual table](http://en.wikipedia.org/wiki/Virtual_table) consulted at run-time to derive the proper offset address for an object’s method.  **This behavior is referred to as "**[**Dynamic Dispatch**](http://en.wikipedia.org/wiki/Dynamic_dispatch)**" or "Single Dispatch**".  So, how does this relate to Double Dispatch?  To answer this question, let’s next review [method overloading](http://en.wikipedia.org/wiki/Function_overloading).

**Method Overloading**

In the following example, our Shape class is redefined to have two overloaded Drawmethods: one with a parameter of type Surface and one with a parameter of typeEtchASketch:  
class Surface {  
}  
  
class EtchASketch extends Surface {  
}  
  
class Shape {  
 public void Draw(Surface surface) {  
 System.*out*.println("A shape is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the shape.");  
 }  
}  
  
class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 shape.Draw(new Surface());  
 shape.Draw(new EtchASketch());  
 }  
}

When executed, the following lines are printed to the console:

A shape is drawn on the surface with ink.

The knobs are moved in attempt to draw the shape.

Note that the parameter type determines which Draw() method is invoked.

But what happens if we change the Main() method to the following?

class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 shape.Draw(surface);  
 shape.Draw(etchASketch);  
 }  
}

Executing this produces the following:

A shape is drawn on the surface with ink.

A shape is drawn on the surface with ink.

What happened?  The issue here is that the method to call was determined statically at compile time based upon the reference type, not at run-time based upon the object type.  To resolve this issue, another technique is needed … Polymorphic Static Binding.

**Polymorphic Static Binding**

Polymorphic static binding is a technique where static method invocations are determined at run-time through the use of polymorphism.  This can be demonstrated in our example by adding a new Draw(Shape shape) method to theSurface and EtchASketch types which call shape.Draw() with a reference to the current object:

class Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class EtchASketch extends Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}

To invoke the correct Shape.Draw() method, our console application needs to be modified to call the the method indirectly through a Surface reference:

class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 surface.Draw(shape);  
 etchASketch.Draw(shape);  
 }  
}

For double dispatch example for my understanding, the complete code is given below.

package com.ddlab.rnd.type2;  
  
class Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class EtchASketch extends Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class Shape {  
  
 public void Draw(Surface surface) {  
 System.*out*.println("A shape is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the shape.");  
 }  
}  
  
class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 surface.Draw(shape);  
 etchASketch.Draw(shape);  
 }  
}

Upon executing the application again, the following lines are now printed:

A shape is drawn on the surface with ink.

The knobs are moved in attempt to draw the shape.

This example achieves the desired result by effectively wrapping the static-dispatched method invocation (i.e. Shape.Draw()) within a virtual-dispatch method invocation (i.e. Surface.Draw() and EtchASketch.Draw()).  This causes the staticShape.Draw() method invocation to be determined by which virtual Surface.Draw()method invocation is executed.

Although the above example now contains a method invocation using a reference to the current object as the method parameter (often seen with Double Dispatch), it should be noted that Double Dispatch has yet to be demonstrated.  Thus far, only one level of virtual dispatching has been used.  To demonstrate Double Dispatch, the techniques from both the polymorphism example and the polymorphic static binding example need to be combined as seen in the next section.

**Double Dispatch**

The following example contains a hierarchy of Surface types and a hierarchy of Shapetypes.  Each Shape type contains an overloaded virtual Draw() method which contains the logic for how the shape is to be drawn on a particular surface.  The example console application uses the polymorphic static binding technique to ensure the proper overload is called for each surface type:

import java.util.ArrayList;  
import java.util.List;  
class Surface {  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class EtchASketch extends Surface {  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class Shape {  
 public void Draw(Surface surface) {  
 System.*out*.println("A shape is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the shape.");  
 }  
}  
  
class Polygon extends Shape {  
 public void Draw(Surface surface) {  
 System.*out*.println("A polygon is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the polygon.");  
 }  
}  
  
class Quadrilateral extends Polygon {  
 public void Draw(Surface surface) {  
 System.*out*.println("A quadrilateral is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the quadrilateral.");  
 }  
}  
  
class Parallelogram extends Quadrilateral {  
 public void Draw(Surface surface) {  
 System.*out*.println("A parallelogram is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the parallelogram.");  
 }  
}  
  
class Rectangle extends Parallelogram {  
 public void Draw(Surface surface) {  
 System.*out*.println("A rectangle is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the rectangle.");  
 }  
}

class Program {  
 public static void main(String[] args) {  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 List<Shape> shapes = new ArrayList<Shape>();  
 shapes.add(new Shape());  
 shapes.add(new Polygon());  
 shapes.add(new Quadrilateral());  
 shapes.add(new Parallelogram());  
 shapes.add(new Rectangle());  
  
 for (Shape shape : shapes) {  
 surface.Draw(shape);  
 etchASketch.Draw(shape);  
 }  
 }  
}

Executing this example produces the following:

A shape is drawn on the surface with ink.

The knobs are moved in attempt to draw the shape.

A polygon is drawn on the surface with ink.

The knobs are moved in attempt to draw the polygon.

A quadrilateral is drawn on the surface with ink.

The knobs are moved in attempt to draw the quadrilateral.

A parallelogram is drawn on the surface with ink.

The knobs are moved in attempt to draw the parallelogram.

A rectangle is drawn on the surface with ink.

The knobs are moved in attempt to draw the rectangle.

In the above example, virtual dispatch occurs twice for each call to one of the Surfacereferences: Once when the Surface.Draw() virtual method is called and again when either calls the Shape.Draw() overloaded virtual method.  Note again that while the second virtual dispatch is based on the type of Shape instance, the overloaded method called is still determined statically based upon the reference type.

**Consequences**

So, what’s wrong with Double Dispatch?  The problem isn’t so much in the technique, but what design choices might be leading to reliance upon the technique.  Consider for instance the hierarchy of shape types in our Double Dispatch example.  What happens if we want to add a new surface?  In this case, each of the shape types will need to be modified to add knowledge of the new Surface type.  This violates the[Open/Closed Principle](http://en.wikipedia.org/wiki/Open_Closed_Principle), and in this case in a particularly egregious way (i.e. It’s violation is multiplied by the number of shape types we have).   Additionally, it violates the [Single Responsibility Principle](http://en.wikipedia.org/wiki/Single_responsibility_principle).  Changes to how shapes are drawn on a particular surface are likely to differ from surface to surface, thereby leading our shape objects to change for different reasons.

The presence of Double Dispatch generally means that each type in a hierarchy has special handling code within another hierarchy of types.  This approach to representing variant behavior leads to code that is less resilient to future changes as well as being more difficult to extend.

**Conclusion**

Since Double Dispatch is a technique for calling virtual overloaded methods based upon parameter types which exist within an inheritance hierarchy, its use may be a symptom that the Open/Closed and/or Single responsibility principles are being violated, or that responsibilities may otherwise be misaligned.  This is not to say that every case of Double Dispatch means something is amiss, but only that its use should be a flag to reconsider your design in light of future maintenance needs.

import java.util.ArrayList;

import java.util.List;

class Circle {

}

class SmallCircle extends Circle {

}

class BigCircle extends Circle {

}

class Shape {

public void draw(Circle circle) {

System.out.println("... Circle ...");

}

public void draw(BigCircle circle) {

System.out.println("... Big Circle ...");

}

public void draw(SmallCircle circle) {

System.out.println("... Small Circle ...");

}

}

public class Test1 {

public static void main(String[] args) {

Shape shape = new Shape();

List<Circle> circles = new ArrayList<Circle>();

circles.add(new Circle());

circles.add( new BigCircle());

circles.add( new SmallCircle() );

for( Circle circle : circles )

shape.draw(circle);

}

}

----------------------------

Output

... Circle ...

... Circle ...

... Circle ...

**To resolve the above problem, make small change**

package com.ddlab.rnd.type3;

import java.util.ArrayList;

import java.util.List;

class Circle {

public void draw(Shape shape) {

shape.draw(this);

}

}

class SmallCircle extends Circle {

public void draw(Shape shape) {

shape.draw(this);

}

}

class BigCircle extends Circle {

public void draw(Shape shape) {

shape.draw(this);

}

}

class Shape {

public void draw(Circle circle) {

System.out.println("... Circle ...");

}

public void draw(BigCircle circle) {

System.out.println("... Big Circle ...");

}

public void draw(SmallCircle circle) {

System.out.println("... Small Circle ...");

}

}

public class Test1 {

public static void main(String[] args) {

Shape shape = new Shape();

List<Circle> circles = new ArrayList<Circle>();

circles.add(new Circle());

circles.add(new BigCircle());

circles.add(new SmallCircle());

for (Circle circle : circles) {

// shape.draw(circle); //Do not call like this

circle.draw(shape);

}

}

}

Let us take another example like banking to open different accounts like Saving, Demat, Loan etc.

**public class** Account {  
 **public void** open() {  
 System.***out***.println(**"Basic Account opened ..."**);  
 }  
}

**public class** DematAccount **extends** Account {  
}

**public class** LoanAccount **extends** Account {  
}

**public class** SavingsAccount **extends** Account {  
}

**public class** Bank {  
  
 **public void** openAccount(Account act) {  
 System.***out***.println(**"Basic account ...."**);  
}  
  
 **public void** openAccount(SavingsAccount act) {  
 System.***out***.println(**"Savings account ...."**);  
}  
  
 **public void** openAccount(DematAccount act) {  
 System.***out***.println(**"Demat account ...."**);  
}  
  
 **public void** openAccount(LoanAccount act) {  
 System.***out***.println(**"Loan account ...."**);}  
}

**import** java.util.ArrayList;  
**import** java.util.List;  
**public class** Test {  
  
 **public static void** main(String[] args) {  
 List<Account> actList = **new** ArrayList<Account>();  
  
 actList.add( **new** Account());  
 actList.add( **new** SavingsAccount());  
 actList.add( **new** DematAccount());  
 actList.add( **new** LoanAccount());  
  
 Bank bank = **new** Bank();  
 **for**( Account act : actList)  
 bank.openAccount(act);  
 }  
}

**OUTPUT**

Basic account ....

Basic account ....

Basic account ....

Basic account ....

Let us modify the above code using visitor pattern.

**public class** Account {  
  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Basic Account opened ..."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** DematAccount **extends** Account {  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Demat account opened .."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** LoanAccount **extends** Account {  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Loan account opened .."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** SavingsAccount **extends** Account {  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Savings account opened .."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** Bank {  
 **public void** openAccount(Account act) {  
 System.***out***.println(**"Basic account ...."**);  
 }  
  
 **public void** openAccount(SavingsAccount act) {  
 System.***out***.println(**"Savings account ...."**);  
 }  
  
 **public void** openAccount(DematAccount act) {  
 System.***out***.println(**"Demat account ...."**);  
 }  
  
 **public void** openAccount(LoanAccount act) {  
 System.***out***.println(**"Loan account ...."**);  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.List;  
**public class** Test {  
 **public static void** main(String[] args) {  
 List<Account> actList = **new** ArrayList<Account>();  
  
 actList.add( **new** Account());  
 actList.add( **new** SavingsAccount());  
 actList.add( **new** DematAccount());  
 actList.add( **new** LoanAccount());  
  
 Bank bank = **new** Bank();  
 **for**( Account act : actList) {  
*// bank.openAccount(act);//Do not call like this* act.open(bank);  
 }  
 }  
}

**OUTPUT**

Basic Account opened ...

Basic account ....

Savings account opened ..

Savings account ....

Demat account opened ..

Demat account ....

Loan account opened ..

Loan account ....

Let us see one peculiarity in the following example.

import java.util.ArrayList;

import java.util.List;

class SavingAccount {

}

class DematAccount extends SavingAccount {

}

class Bank {

public void open(SavingAccount act ) {

System.out.println("... Opening Saving Account ...");

}

public void open(DematAccount act ) {

System.out.println("... Opening Demat Account ...");

}

}

public class Test {

public static void main(String[] args) {

List<SavingAccount> actList = new ArrayList<SavingAccount>();

Bank bank = new Bank();

actList.add( new SavingAccount());

actList.add( new DematAccount());

for( SavingAccount act : actList ) {

bank.open(act);

}

}

}

Here the output is given below.

**... Opening Saving Account ...**

**... Opening Saving Account ...**

Now let me modify the code and see the output below.

import java.util.ArrayList;

import java.util.List;

class SavingAccount {

public void open() {

System.out.println("... Opened Saving Account Successfully ...");

}

}

class DematAccount extends SavingAccount {

public void open() {

System.out.println("... Opened Demat Account Successfully ...");

}

}

class Bank {

public void open(SavingAccount act ) {

System.out.println("... Opening Saving Account ...");

act.open();

}

public void open(DematAccount act ) {

System.out.println("... Opening Demat Account ...");

act.open();

}

}

public class Test {

public static void main(String[] args) {

List<SavingAccount> actList = new ArrayList<SavingAccount>();

Bank bank = new Bank();

actList.add( new SavingAccount());

actList.add( new DematAccount());

for( SavingAccount act : actList ) {

bank.open(act);

}

}

}

Here the output is ... Opening Saving Account ... ... Opened Saving Account Successfully ... ... Opening Saving Account ... ... Opened Demat Account Successfully ...

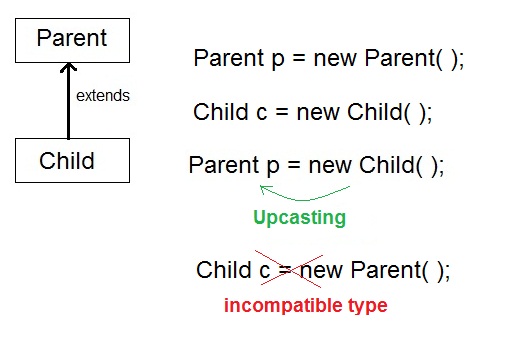
Now my question is I am getting the result what I am expecting, why should I go for Visitor pattern, in the above code even if it displays "Saving Account", but it properly executes "Demat Account Code" portion.

Answer by StackOverflow.

Java's calls are polymorphic on objects. act.open() can call the subclass method because the runtime has the object to call it on, but bank.open(act) can't call the method taking a subclass parameter because **bank** is the object on which polymorphism is decided not **act** and**bank** has an **open()** method for **SavingAccount** so that is what has to be called.

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.

Multiple Dispatch

It is the language’s mechanism to decide which method to invoke based on the actual runtime type of the passed arguments. Java in contrast links the method at compile time based on the static types.

In statically typed languages, including Java, the biggest difference between dispatch and overloading is that overloading is based on the *static* type of parameters (i.e. the choice of which method is actually called is decided compile-time), while dispatch is based on the *dynamic* types (i.e. the decision is made runtime). (Such languages usually don't support multiple dispatch.)

Let us take an example.

**class** Fruit {

@Override

**public** String toString() {

**return** "Fruit";

}

}

**class** Banana **extends** Fruit {

@Override

**public** String toString() {

**return** "Banana";

}

}

**public** **class** Test1 {

**public** **void** print(Fruit fruit) {

System.***out***.println("Plain fruit ...");

}

**public** **void** print(Banana banana) {

System.***out***.println("This is banana ...");

}

**public** **static** **void** main(String[] args) {

Test1 test1 = **new** Test1();

Fruit fruit = **new** Fruit();

**test1.print(fruit); //Output Plain fruit ...**

Banana banana = **new** Banana();

**test1.print((Fruit) banana); //Output Plain fruit ...**

}

}

Output will be

**Plain fruit ...**

**Plain fruit ...**

To resolve the above the above problem , we rewrite the above program in the following manner.

**package** com.ddlab.rnd.type1;

**class** Vegetable {

@Override

**public** String toString() {

**return** "Vegetable";

}

}

**class** Tomato **extends** Vegetable {

@Override

**public** String toString() {

**return** "Tomato";

}

}

**public** **class** Test2 {

**public** **void** print(Vegetable vegetable ) {

**if**( vegetable **instanceof** Tomato ) {

System.***out***.println("It is Tomato ...");

}

**else** {

System.***out***.println("It is vegetable");

}

}

**public** **static** **void** main(String[] args) {

Test2 test2 = **new** Test2();

Vegetable vegetable = **new** Vegetable();

**test2.print(vegetable); // Output It is vegetable**

Tomato tomato = **new** Tomato();

**test2.print((Vegetable) tomato); // Output It is Tomato ...**

}

}

**Output will be**

**It is vegetable**

**It is Tomato ...**