**Single, Dynamic, Multiple and Double Dispatching - 2022**

**What is Dispatching?**

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

**Late Binding**

type is unknown until the variable is used at run-time

**Early Binding**

type is known before the variable is exercised during run-time

* Single/Dynamic Dispatching
* Multiple Dispatching
* Double Dispatching

**Single/Dynamic Dispatching**

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

**public class** Test1 {  
 **public static void** main(String[] args) {  
 Account act = new Account();  
 act.open();  
 }  
}

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

**public interface** Account {  
 **void** open();  
}

**public class** SavingAccount **implements** Account {  
 @Override  
 **public void** open() {  
 System.***out***.println(**"Saving account"**);   
 }  
}

**public static void** main(String[] args) {  
 *//Account act = new LoanAccount(); //***Loan**Account act = **new** SavingAccount(); //**Saving**  
 Bank bank = **new** Bank();  
 bank.openAccount(act);  
}

**public class** LoanAccount **implements** Account {  
 @Override  
 **public void** open() {  
 System.***out***.println(**"Loan Account"**);  
 }  
}

**public class** Bank {  
 **void** openAccount(Account account) { 🡸 **Dynamic Dispatch**  
 account.open();  
 }  
}

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

**Multiple Dispatching**

**Multiple dispatch is resolved at runtime.** 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/)

**public class** Account {  
  
}

**public class** SavingAccount **extends** Account {  
  
}

**public class** LoanAccount **implements** Account {  
  
}

**public class** Bank {  
 **void** openAccount(Account account) {  
 System.***out***.println(**"...Normal..."**);  
 }  
 **void** openAccount(SavingAccount account) {  
 System.***out***.println(**"...Savings..."**);  
 }  
  
 **void** openAccount(LoanAccount account) {  
 System.***out***.println(**"...Loan..."**);  
 }  
}

**public static void** main(String[] args) {  
 List<Account> actList = **new** ArrayList<Account>();  
 actList.add( **new** SavingAccount());  
 actList.add( **new** LoanAccount());  
 Bank bank = **new** Bank();  
 **for**( Account act : actList)  
 bank.openAccount(act);  
}

**OUTPUT**

...Normal...

...Normal...

**It happens in java.** BTW, Java doesn't support true multiple dispatching, so don't cause any bugs while creating overloaded methods. You may be thinking that it may be because of Account is a class, let us change it to an interface. Again the output will be same. **It means the method parameter checks the exact class or interface type in case of overloading**.

**What is Binding?**

**Binding" is the idea that the method is "bound" to a particular instance (or class of instances)**

a method that's bound to an object with dynamic binding might use dynamic dispatch when you call it. I**t's the mechanism behind polymorphism, in which a reference to an object might point to one of multiple implementations.** **Dynamic dispatch decides which method to invoke at runtime.**

**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

**Double Dispatching**

**More simply, Double Dispatch is used to invoke an overloaded method where the parameters vary among an inheritance hierarchy**. When using double dispatch, the called method depends on the actual type of receiver and arguments. Visitor pattern is actually good way of explaining this topic. **Method overloading however, only allows the called method to depends on the declared type of the parameters**. It is the language’s mechanism to decide which method to invoke based on the actual runtime type of the passed arguments.

Let us consider a problem and then we will go to the solution.

**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 ...."**);}  
}

**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**);  
 }  
}

**OUTPUT**

Basic Account opened ...

Basic account ....

Savings account opened ..

Savings account ....

Demat account opened ..

Demat account ....

Loan account opened ..

Loan account ....

**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 ...."**); 🡸 **This is called Double Dispatch**.  
 }  
  
 **public void** openAccount(DematAccount act) {  
 System.***out***.println(**"Demat account ...."**);  
 }  
  
 **public void** openAccount(LoanAccount act) {  
 System.***out***.println(**"Loan account ...."**);  
 }  
}

**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);  
 }  
 }

**Why Double Dispatch**

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.

Actual Difference between Dispatch and Binding

Binding is always related to an object either at compile time or run time. For example

Parent p = new Child() or new Parent();

p.invokeMethod(); 🡸 This is binding.

Dispatch is always related to method invocation where parameter may vary.