در ادامه سری مقالات الگوی های طراحی نرم افزار های شی گرا به الگوی iterator خواهیم پرداخت.

**نقش**

این الگو راهی برای دست یافتن به عناصر موجود در یک مجموعه را فراهم می کند. این دست یابی بدون توجه به ساختار مجموعه به صورت ترتیبی است. همچنین این الگو می تواند عناصر را به شکل های گوناگونی فیلتر نماید.

**تشریح**

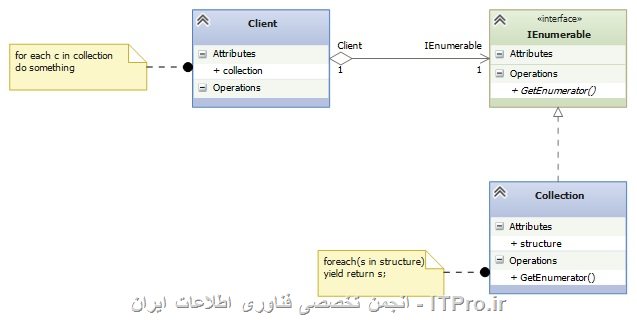
برای ذخیره داده های مرتبط در زبان های [برنامه نویسی](http://itpro.ir/gotorel/technichaltext/32786) از مجموعه عناصر استفاده می شود. این مجموعه عناصر می توانند مجموعه ها ، لیست ها و آرایه ها  و... باشند. هر کدام از این مجموعه ها با توجه به شرایط استفاده می تواند بسیار مفید باشد. هر کدام از این مجموعه ها عناصر خود را به گونه ای متفاوت ذخیره می کنند. حال تصور کنید که در برنامه خود از آرایه استفاده می کرده اید و بنابر تغییرات برنامه و نیاز به وجود آمده ناگزیر به استفاده از لیست باشید باید همه ی قسمت های برنامه که از آرایه مذکور استفاده می کرده اند را تغییر داده و بازنویسی کنید که این بار از لیست به جای آرایه استفاده کنند. خود این عمل بسیار مشکل است و می تواند باعث به وجود آمدن ناسازگاری های بسیاری شود. راه حل این است که روش استفاده از مجموعه عناصر به شکل یکتا پیاده سازی شود.   
الگوی iterator یکی از ساده ترین الگوها و در عین حال یکی از الگوهای پرکاربرد در بین الگوهای طراحی می باشد. این الگو یک واسط استاندارد برای استفاده کنندگان فراهم می کند که بدون توجه به چگونگی پیاده سازی یک مجموعه از عناصر، استفاده کنندگان بتوانند به اعضای مجموعه دسترسی داشته باشند. بسیاری از زبان های برنامه نویسی شی گرا مانند #C و Java در مجموعه های خود از این الگو استفاده کرده اند.   
در استفاده از این الگو باید دو مفهوم iterator و enumerator را تعریف کنیم. مفاهیم iterator و enumerator دارای قدمت طولانی می باشد. Enumerator مسئول یافتن شی بعدی است که شرط خاصی را دارد. به طور مثال اگر بخواهیم عدد اول کوچکتر از هزار بعدی را پیدا کنیم از enumerator استفاده می کنیم و این مجموعه اعداد 1 تا 1000 را مجموعه enumerable می نامیم. مفهوم iterator به این معنی است که توسط حلقه ای یک مجموعه از اشیا را از اول تا آخر پیمایش نماییم. الگوی iterator باید هر دوی این مفاهیم را پیاده سازی کند.

**موارد استفاده**

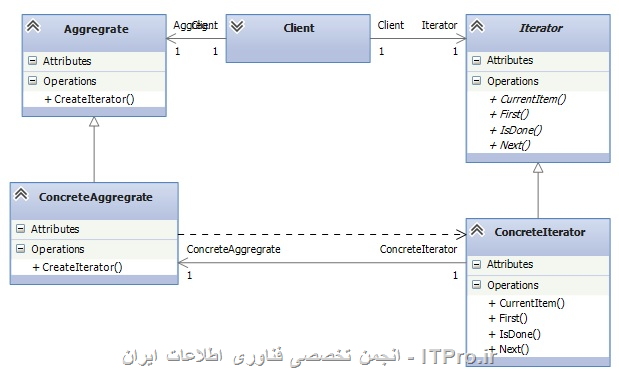
از این الگو در موارد زیر استفاده می شود:   
برای دسترسی به محتوای یک شئ تجمعی بدون در نظر گرفتن ساختار داخلی آن.   
برای پیمایش در اشیا تجمعی(هر گاه هر شی خود دارای زیر گروه هایی از اشیا باشد و بخواهیم به همه آن ها دسترسی داشته باشیم.)   
برای فراهم کردن یک واسط یکتا برای پیمایش ساختار های تنوعی متفاوت.

**ساختار**

ساختار این الگو در زبان C# اینگونه پیاده سازی شده است که برای پیاده سازی iterator از حلقه foreach استفاده می شود و برای هر مجموعه ای که بخواهد از این الگو استفاده کند باید اینترفیس Ienumerable را پیاده سازی کند. ساختار این الگو به این شکل است:



همانطور که در شکل دیده می شود مجموعه ها اینترفیس IEnumerable را پیاده سازی می کنند. این اینترفیس دارای یک تابع به نام GetEnumerator() می باشد که همان کار enumerator را پیاده سازی می کند. اگر این اینترفیس و این تابع در مجموعه ای پیاده سازی نشده باشد قسمت iterator یعنی حلقه foreach نمی تواند کار خود را انجام دهد.   
  
در زبان هایی که این مکانیزم در آن پیاده سازی نشده است مانند++C باید از ساختار زیر استفاده کرد:



در این نمودار برنامه برای پیمایش یک لیست در این ساختار از ListIterator استفاده می شود که در طول یک حلقه به کاوش لیست می پردازد.

**پیاده سازی**

اول پیاده سازی این الگو برای زبان ++C را توضیح می دهم که هیچگونه اینترفیس آماده ای ندارد و کاربر باید از اول خود به پیاده سازی این الگو بپردازد و سپس به پیاده سازی این الگو در زبان #C خواهیم پرداخت.   
برای پیمایش یک مجموعه از عناصر باید توابع داخل کلاس Iterator را بازنویسی نماییم. یعنی کلاس ConcreteIteratorاین توابع را با توجه به یک مجموعه خاص پیاده سازی می کند. قطعه کد زیر این مثال را برای یک لیست نشان می دهد. 

template <class Item>

class Iterator {

public:

virtual void First() = 0;

virtual void Next() = 0;

virtual bool IsDone() const = 0;

virtual Item CurrentItem() const = 0;

protected:

Iterator();

};

template <class Item>

class ListIterator : public Iterator<Item> {

public:

ListIterator(const List<Item>\* aList);

virtual void First(){

\_current = 0;

}

virtual void Next(){

\_current++;

}

virtual bool IsDone() const{

return \_current >= \_list->Count();

}

virtual Item CurrentItem() const{

if (IsDone()) {

throw IteratorOutOfBounds;

}

return \_list->Get(\_current);

}

private:

const List<Item>\* \_list;

long \_current;

};

void PrintNumbers(ListIterator<int> i) {

for (i.First(); !i.IsDone(); i.Next()) {

cout<<i.CurrentItem()<<endl;

}

}

همانگونه که در کد نشان داده شده است کلاسی به نام Iterator وجود دارد که فقط چند تابع virtual دارد و این توابع توسط کلاس ListIterator که از کلاس قبلی ارث بری کرده است پیاده سازی شده اند. در تابع PrintNumbers نحوه استفاده از این الگو را می توان دید که از اولین عنصر آن پیمایش شروع شده و تا آخرین عنصر پیش می رود و هرکدام از عناصر را در یک خط چاپ می نماید.   
در زبان های سطح بالا تر مانند Java و #C اینترفیس هایی وجود دارند که کار را آسان تر کرده و نیازی نیست که از ابتدا شروع به نوشتن این الگو نماییم.   
قطعه کد زیر پیاده سازی این الگو را برای ماه های سال در زبان#C نشان می دهد. 

using System;

using System.Collections;

public class Iterator

{

public class MonthCollection:IEnumerable

{

string[] months = { "January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December" };

public IEnumerator GetEnumerator()

{

foreach (var month in months)

{

yield return month;

}

}

}

static void main()

{

var collection = new MonthCollection();

foreach (var element in collection)

{

Console.WriteLine(element);

}

}

}

همان طور که در کد بالا دیده می شود مجموعه ای که قصد پیاده سازی این الگو در آن را داریم یک آرایه رشته ای است که ماه ها را در خود ذخیره کرده است. توسط کلمه کلیدی yield می توان به کامپایلر فهماند که مقداری که return می کنیم مقدار enumerator است یعنی عنصر بعدی است. البته می توان در داخل foreach موجود در تابع getenumerator شرط گذاشت که همه ماه ها را به خروجی نفرستد. با این کار هرکدام از عناصر که شرط مورد نظر را داشته باشند به foreach موجود در تابع main ارسال می شود.