به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی

برنامه نویسی شیءگرا

انتزاع داده

- برنامه نویسی شیءگرا [1] یک پارادایم (الگوواره) برنامه نویسی بر مبنای مفهوم شیء [2] است. یک شیء در واقع یک نمونه از یک نوع داده است که ویژگی‌ها و رفتارهایی دارد. در برنامه نویسی شیءگرا نوع داده‌ای که دارای ویژگی‌ها و رفتارهای تعریف شده است، کلاس نامیده می‌شود. ویژگی‌های یک کلاس خصوصیاتی هستند که با نوع‌های دیگر داده‌ای قابل توصیف است و رفتارهای یک کلاس توابعی هستند که برای آن کلاس تعریف شده‌اند.
- یک کلاس در واقع یک نوع دادهٔ انتزاعی است و کلاس‌ها می‌توانند در سطوح متفاوت انتزاعی قرار بگیرند.

[1] object-oriented programming
[2] object

انتزاع داده

- انتزاع داده [1] در واقع روشی است برای نمایش یک موجودیت به طوری که تنها ویژگی‌های مهم آن موجودیت نمایش داده شود.
- به عبارت دیگر با استفاده از انتزاع داده می‌توان نمونه‌های متفاوت از موجودات را بر اساس ویژگی‌های مشترک آنها در یک دسته قرار داد.

[1] data abstraction

انتزاع داده

- برای مثال وقتی از کلمهٔ پرنده استفاده می‌کنیم در واقع موجودات بسیار زیادی را بر اساس ویژگی‌هایشان در یک دسته قرار داده‌ایم. همهٔ جانداران که دو بال دارند و دو پا دارند و بدنشان از پر پوشیده شده است را با کلمهٔ پرنده در یک گروه قرار داده‌ایم. حال در یک زبان برنامه نویسی می‌توانیم این کلمه را با ویژگی‌های آن توصیف کنیم که در واقع یک نوع داده‌ای انتزاعی جدید تعریف کرده‌ایم. حال اگر بگوییم باز یک پرنده است، می‌دانیم که باز هم ویژگی‌های همه پرندگان را داراست. حال ممکن است در یک سطح پایین‌تر انتزاع، دسته پرندگانی را تعریف کنیم که پرواز می‌کنند و همچنین دسته پرندگانی که پرواز نمی‌کنند. پرندگانی که پرواز نمی‌کنند همه ویژگی‌های پرندگان را دارا هستند و علاوه بر آن ویژگی‌های دیگری نیز دارند. در یک سطح پایین‌تر انتزاع می‌توانیم کلاسی برای همهٔ پنگوئن‌ها تعریف تعریف کنیم. یک پنگوئن همه ویژگی‌های پرندگانی که پرواز نمی‌کنند و همهٔ ویژگی‌های پرندگان را داراست و علاوه بر آن ویژگی‌های دیگر هم دارد مثلاً اینکه از ماهی تغذیه می‌کند.
- حال یک پنگوئن واقعی را در نظر بگیرید این پنگوئن قد و وزن و ویژگی‌های خاص خود را دارد و از کلاس پنگوئن‌هاست.

انتزاع داده

- در زبان‌های برنامه نویسی دو نوع انتزاع وجود دارد : انتزاع پروسه یا فرایند [1] و انتزاع داده. انتزاع فرایند بدین معنا است که فرایندی با محاسبات مشخص را دسته‌بندی کرده و با یک نام فرایند که زیر برنامه نیز نامیده می‌شود نامگذاری کنیم و پس از آن، آن فرایند را با استفاده از نام و پارامترهای آن استفاده کنیم.
- وقتی می‌خواهیم یک لیست را مرتب کنیم از هر دو نوع انتزاع استفاده می‌کنیم. لیست یک نوع انتزاعی با ویژگی‌های مشخص و تعریف شده است و مرتب سازی، یک عملیات مشخص و تعریف شده است. در زبان‌های طبیعی اسم‌ها داده انتزاعی هستند و فعل‌ها فرایند انتزاعی چرا که یک عملیات مشخص را نامگذاری می‌کنند.

[1] process abstraction

انتزاع داده

- اولین نسخه دادهٔ انتزاعی در زبان کوبول در سال ۱۹۶۰ برای تعریف ساختار داده‌ها به وجود آمد و بعدها مفهوم ساختمان یا استراکت [1] در زبان سی نیز مورد استفاده قرار گرفت.
- یک نوع داده تعریف شده توسط کاربر به وسیلهٔ ساختمان در واقع ویژگی‌هایی را تجمیع می‌کرد که بدین ترتیب متغیرهایی از آن نوع داده انتزاعی می‌توانستند به توابع نیز ارسال شوند.
- یک نمونه از یک نوع انتزاعی در برنامه نویسی شیءگرا یک شیء نامیده می‌شود.

[1] struct

انتزاع داده

- دقت کنید که حتی نوع داده‌های اصلی در یک زبان برنامه نویسی هم انتزاعی هستند. مثلا نوع دادهٔ اعشاری همهٔ اعدادی را که به نحو معینی بر روی حافظه ذخیره می‌شوند و ویژگی‌های مشترکی دارند را با عدد اعشاری یا ممیز شناور [1] تعریف می‌کند.

[1] floating point

انتزاع داده

- یک نوع دادهٔ انتزاعی [1] نوع داده‌ای است که در تعریف آن از تعدادی ویژگی‌ها با نوع داده‌های دیگر استفاده شده و دارای رفتارهایی است. ویژگی‌های خصوصی یک نمونه از آن نوع داده‌ای که شیء نامیده می‌شود از استفاده کنندهٔ شیء پنهان است و استفاده کنندهٔ شیء تنها به رفتارهای عمومی شیء دسترسی دارد.
- یک کاربر شیء انتزاعی نیازی به دانستن جزئیات پیاده سازی آن نوع داده انتزاعی ندارد بلکه کافی است نام نوع داده و توابع (رفتارهای) عمومی که برای آن تعریف شده را بشناسد تا بتواند از آن استفاده کند.
- در زبان سی از ساختمان‌ها برای تعریف داده‌های انتزاعی استفاده می‌شد و توابعی برای استفاده از آن نوع داده‌ها تعریف می‌شد.

[1] abstract data type

انتزاع داده

- در برنامه نویسی شیءگرا نوع دادههای انتزاعی کلاس نامیده می‌شوند و هر نمونه از یک کلاس یک شیء نامیده می‌شود.
- یک کلاس علاوه بر تعریف ویژگیهای آن کلاس، تعدادی رفتار نیز برای کلاس تعریف می‌کند. یک رفتار در واقع یک تابع است که بر روی یک شیء عملیاتی انجام می‌دهد.
- ویژگی‌های یک کلاس معمولاً به صورت خصوصی تعریف می‌شوند و از استفاده کننده کلاس پنهان هستند و تنها با استفاده از توابع (رفتارهای) تعریف شده بر روی کلاس و رابطهای کاربری تعریف شده می‌توان ویژگی‌های کلاس را تغییر داد.
- بر خلاف زبان‌های غیر شیءگرا که در آنها نوع‌های داده‌ای انتزاعی جدا از توابعی هستند که بر روی آنها اعمال می‌شوند، در زبان‌های شیءگرا ویژگی‌ها و رفتارها در کنار یکدیگر در نوع داده‌ای مورد نظر تعریف می‌شوند.

انتزاع داده

- این مخفی سازی اطلاعات [1] در کلاس‌ها چندین مزیت دارد.
- مزیت اول این مخفی سازی اطلاعات بالا بردن قابلیت اطمینان برنامه است. اگر یک کاربر بتواند اطلاعاتی را در یک نوع داده تغییر دهد، این تغییر ممکن است باعث ایجاد اختلال در منطق داده مورد نظر شود. فرض کنید یک نوع دادهٔ جدید برای تعریف یک موجودیت ایجاد کرده‌ایم. فرض کنید این موجودیت یک صف باشد. نوع دادهٔ مورد نظر باید تعدادی ویژگی به عنوان اطلاعات مورد نیاز این صف نگهداری کند. برای مثال اندیس ابتدای صف در یک آرایه می‌تواند یکی از این اطلاعات باشد. حال اگر کاربر بتواند این اطلاعات را تغییر دهد، عملیات درج و برداشت از صف مورد نظر به درستی عمل نخواهند کرد. بنابراین در برنامه نویسی شیءگرا معمولا ویژگی‌ها پنهان می‌شوند و اطلاعاتی که باید در دسترس قرار بگیرند توسط تعدادی تابع به عنوان واسط کلاس در دسترس کاربران قرار می‌گیرند.
- مزیت دوم ساده شدن برنامه نویسی با کاهش جزئیات برای کاربر است.

[1] information hiding

انتزاع داده

- مزیت سوم بهبود قابلیت تغییر برنامه است به نحوی که برنامه کاربران تحت تاثیر قرار نگیرد. به عبارت دیگر جزئیات پیاده سازی یک نوع داده‌ای از کاربر پنهان می‌ماند و کاربر تنها توسط واسط‌هایی از کلاس‌های مورد نیاز خود استفاده می‌کند. حال اگر تغییر در جزئیات پیاده سازی یک کلاس (برای بهبود عملکرد آن) به وجود بیاید، کاربر کلاس نیازی به تغییر برنامهٔ خود ندارد و با استفاده از همان واسط‌های قبلی از کلاس مورد نیاز خود استفاده می‌کند. برای مثال فرض کنید در پیاده سازی یک پشته به جای آرایه از لیست پیوندی استفاده شود. کاربر نیازی به اطلاع از آن ندارد و می‌تواند به صورتی که قبلا از کلاس استفاده می‌کرد، از کلاس استفاده کند.

انتزاع داده

- در مواقعی که یک کاربر نیاز به تغییر ویژگی‌های یک شیء دارد، معمولاً در کلاس مربوطه توابعی به نام دریافت کننده [1] و تنظیم کننده [2] برای دریافت و تنظیم ویژگی‌های یک شیء از یک کلاس تعریف می‌شوند.

- دریافت کننده‌ها و تنظیم کننده چندین مزیت به همراه دارند. اول آنکه دسترسی به ویژگی‌ها غیر مستقیم می‌شود و کاربر با دستکاری مکان حافظه نمی‌توانند مقدار ویژگی‌های یک شیء را تغییر دهند. دوم آنکه ممکن است تنظیم کننده‌ها نیاز داشته باشند محدودیت‌هایی را برای تنظیم مقادیر اعمال کنند، برای مثال تنظیم‌کننده‌ها می‌توانند محدودهٔ مقدار را بررسی کنند. سوم آنکه ممکن است پیاده سازی ویژگی‌ها تغییر کند اما همچنان تنظیم کننده‌ها و دریافت کننده‌ها به عنوان واسط کاربر یکسان باقی خواهند ماند.

[1] getter
[2] setter

انتزاع داده

- برای ایجاد چنین قابلیت‌هایی در زبان شیءگرا برای ویژگی‌ها و رفتارها سطح دسترسی تعریف می‌شود. تنها ویژگی‌ها و رفتارها با سطح دسترسی عمومی [1] توسط کاربر قابل استفاده هستند و ویژگی‌ها و رفتارها با دسترسی خصوصی [2] از کاربران پنهان می‌شوند.
- همچنین در تعریف یک کلاس می‌توان تعیین کرد در هنگام ساخته شدن یک شیء یا تخریب یک شیء یا انتساب یک شیء به شیء دیگر چه اتفاقی بیافتد. در زبان سی++ می‌توان عملگرها را برای کلاس‌ها سربارگذاری کرد و بدین ترتیب با اعمال عملگرها بر روی اشیاء عملیات مورد نظر اعمال خواهد شد.

[1] public
[2] private

انتزاع داده

- مفهوم انتزاع داده برای اولین بار در زبان سیمولا به وجود آمد.
- زبان سی++ در سال ۱۹۸۵ برای افزودن شیءگرایی به زبان سی به وجود آمد. نوع‌های داده‌ای انتزاعی در سی++ کلاس نامیده می‌شوند. یک کلاس حاوی تعدادی ویژگی است که اعضای داده‌ای [1] نامیده می‌شوند و تعدادی رفتار به نام توابع عضو [2] دارد.
- اعضای یک کلاس مقدار نمی‌گیرند بلکه این اعضای اشیاء کلاس‌ها هستند که دارای مقدار هستند. یک نمونه از یک کلاس شیء نامیده می‌شود.
- اشیاء همچون متغیرهای دیگر می‌توانند ثابت یا غیر ثابت، ایستا، پویا بر روی پشته یا پویا بر روی هیپ باشند.

[1] data members
[2] member function

انتزاع داده

- اشیائی که به صورت پویا بر روی هیپ هستند توسط عملگر `new` ساخته و تخصیص می‌شوند و توسط عملگر `delete` تخریب شده و فضای حافظه آنها آزاد می‌شود. همچنین عملگر `new` تابع سازنده کلاس را و عملگر `delete` تابع مخرب کلای را برای شیء ساخته شده فراخوانی می‌کنند.
- اعضای داده‌ای کلاس نیز می‌توانند متغیرهای ثابت یا غیر ثابت، ایستا یا اشاره‌گر باشند.
- هر یک از اعضای کلاس‌ها می‌توانند خصوصی یا عمومی باشند. اعضای خصوصی با کلمهٔ private مشخص می‌شود و دسترسی کاربر شیء را به آن عضو غیر ممکن می‌سازند. اعضای عمومی یک کلاس با کلمه public مشخص می‌شوند و دسترسی کاربران شیء به آن عضو امکان پذیر است. معمولاً ویژگی‌ها با سطح دسترسی خصوصی و توابعی که کاربران نیاز دارند با سطح دسترسی عمومی تعریف می‌شوند. همچنین یک سطح دسترسی حفاظت شده نیز وجود دارد که در ارث بری از آنها استفاده می‌شود. کلاس‌هایی که از یک کلاس به ارث می‌برند به اعضای حفاظت شده دسترسی دارند، ولی کاربران به اعضای حفاظت شده دسترسی ندارند. اعضای حفاظت شده با کلمهٔ protected مشخص می‌شوند.

انتزاع داده

- در سی++ همچنین می‌توان یک کلاس را از یک نوع پارامتری تعریف کرد. بدین معنی که یک کلاس می‌تواند یک پارامتر به عنوان نوع برای نوع داده‌ای ویژگی‌های خود دریافت کند.
- برای مثال یک پشته که بتواند انواع داده‌ای متنوع را ذخیره کند به صورت زیر تعریف می‌شود.

```
۱ #include <iostream.h>
۲ template <typename Type> // Type is the template parameter
۳ class Stack {
۴   private:
۵     Type *stackPtr;
۶     int maxLen;
۷     int topSub;
```

انتزاع داده

```
 8   public:
 9     // A constructor for 100 element stacks
10     Stack() {
11         stackPtr = new Type [100];
12         maxLen = 99;
13         topSub = -1;
14     }
15     // A constructor for a given number of elements
16     Stack(int size) {
17         stackPtr = new Type [size];
18         maxLen = size - 1;
19         topSub = -1;
20     }
21     ~Stack() {
22         delete stackPtr;
23     } // A destructor
```

انتزاع داده

```
۲۴      void push(Type number) {
۲۵          if (topSub == maxLen)
۲۶              cout << "Error in push - stack is full\n";
۲۷          else stackPtr[++topSub] = number;
۲۸      }
۲۹      void pop() {
۳۰          if (empty())
۳۱              cout << "Error in pop - stack is empty\n";
۳۲          else topSub--;
۳۳      }
```

انتزاع داده

```
۳۴      Type top() {
۳۵          if (empty())
۳۶              cerr << "Error in top - stack is empty\n";
۳۷          else
۳۸              return (stackPtr[topSub]);
۳۹      }
۴۰      int empty() {
۴۱          return (topSub == -1);
۴۲      }
۴۳  }
```

انتزاع داده

- سپس برای تعریف یک پشته حاوی مقادیر صحیح از پارامتر int در تعریف شیء استفاده می‌کنیم :

```
۱  Stack <int> istack;
۲  Stack <float> fstack;
۳  Stack <Student> stustack;
۴
۵  istack.push(1);
۶  istack.push(2);
۷  while (!istack.empty()) {
۸      int i = istack.pop();
۹      cout << i << endl;
۱۰ }
۱۱ istack.maxLen++; // error
```

انتزاع داده

- در زبان پایتون اولین پارامتر توابع یک کلاس متغیر self است که مشخص می‌کند تابع متعلق به کلاس مربوطه است و می‌تواند به توابع و اعضای داده‌ای دیگر از طریق self دسترسی پیدا کند.
- تابع سازنده نیز با نام __init__ تعریف می‌شود.
- برای مثال :

```
۱ class Bag :
۲       def  __init__ (self) :
۳            self.data = [ ]
۴       def add (self,x) :
۵            self.data.append (x)
```

- اعضای داده‌ای که با (__) شروع می‌شوند اعضای خصوصی هستند.
- برای پیاده سازی مخرب یک کلاس باید از تابع __del__ استفاده کرد.

وراثت

- مفهوم شیءگرایی اولین بار در زبان سیمولا به وجود آمد، اما برای اولین بار در سال ۱۹۸۰ به طور کامل در زبان اسمالتاک پیاده شد. اسمالتاک یک زبان شیءگرای خالص است بدین معنی که همهٔ متغیرها در واقع شیء هستند. در یک زبان شیءگرا علاوه بر انتزاع داده، دو مفهوم اساسی دیگر به نام وراثت و چند ریختی نیز وجود دارند که در اینجا به مطالعهٔ آن می‌پردازیم.

- یکی از معیارهای مهم سنجش یک زبان، قابلیت آن زبان برای تسهیل استفاده مجدد کدها و نرم‌افزارها [1] است.

- انتزاع داده و تعریف کلاس‌ها امکان استفاده مجدد را فراهم می‌سازد، اما در بسیاری مواقع یک کلاس برای استفاده مجدد نیاز به تغییراتی دارد. علاوه بر این در بسیاری مواقع در یک برنامه نیاز به تعریف دو کلاس متفاوت است که با وجود تفاوت ماهیتی، اشتراکات زیادی نیز با یکدیگر دارند و برای صرفه جویی در کد و همچنین زمان برنامه نویسی بهتر است این جنبه‌های مشترک تنها یک بار پیاده سازی شوند.

[1] software reuse

وراثت

- وراثت [1] در برنامه نویسی شیءگرا راه حلی برای مشکلات مطرح شده است. وقتی یک نوع دادهٔ جدید شبیه یک نوع داده قبلی با اندکی تفاوت است، نوع جدید می‌تواند تمام ویژگی‌ها و رفتارهای نوع قبلی را به ارث ببرد. همچنین وقتی دو کلاس با یکدیگر اشتراکات زیادی دارند می‌توان حدس زد که این ویژگی‌ها و رفتارهای مشترک می‌توانند یک موجودیت جدید را در مدل نرم‌افزار تشکیل دهند به طوری که موجودیت جدید پدر و کلاس‌های موجود فرزند آن کلاس هستند. به طور مثال در یک سیستم نرم‌افزاری ممکن است دو کلاس به نام‌های وکتور و صف برای دو ساختار داده داشته باشیم که این دو ساختار در عین تفاوت، اشتراکات زیادی دارند. وجود این اشتراکات نشان دهندهٔ این است که احتمالاً یک ساختار انتزاعی دیگر وجود دارد. می‌توانیم این ساختار انتزاعی را ظرف [2] بنامیم و برای آن یک کلاس تعریف کنیم. سپس دو کلاس وکتور و صف می‌توانند از کلاس ظرف به ارث ببرند. علاوه بر این که ویژگی‌ها و رفتارهای مشترک تنها یک بار تعریف می‌شوند، برنامه نیز ساختار منسجم‌تری خواهد یافت.

[1] inheritance
[2] container

وراثت

- یک کلاس که از کلاس دیگر به ارث می‌برد زیر کلاس [1] یا کلاس فرزند [2] یا کلاس مشتق شده [3] نامیده می‌شود و کلاسی که از آن به ارث برده شده است، کلاس پایه [4] ، کلاس مافوق [5] ، یا کلاس پدر [6] نامیده می‌شود.
- در برنامه نویسی شیءگرا توابع کلاس گاهی متود [7] نیز نامیده می‌شوند.

[1] subclass
[2] child class
[3] derived class
[4] base class
[5] super class
[6] parent class
[7] method

وراثت

- با استفاده از وراثت کلاس فرزند هم می‌تواند ویژگی‌ها و رفتارهایی را اضافه کند و بدین ترتیب علاوه بر ویژگی‌ها و رفتارهای به ارث برده از پدر، تعدادی را نیز خود تعریف می‌کند و هم اینکه می‌تواند تعدادی از رفتارهای پدر را به گونه‌ای دیگر بازتعریف (دوباره تعریف) کند. همچنین کلاس پدر می‌تواند ویژگی‌ها و رفتارهایی را از کلاس‌های فرزند پنهان کند.
- تابعی که بازتعریف یک تابع پدر است، در واقع تابع پدر را لغو [1] می‌کند.
- برای مثال فرض کنید چندین پرنده از کلاس پرنده به ارث می‌برند. برای کلاس پرنده تابعی به نام `draw` برای رسم پرنده در کلاس پدر تعریف شده است که شمایل پرنده را به صورت کلی بدون جزئیات رسم می‌کند. با وجود این که هر پرنده‌ای ویژگی‌های کلاس پرنده را به ارث می‌برد، اما هر کلاس فرزند پرنده تابع رسم را به گونه‌ای متفاوت پیاده سازی می‌کند. پس تابع `draw` در کلاس پدر توسط کلاس‌های فرزند لغو می‌شود.

[1] override

وراثت

- سطح دسترسی حفاظت شده [1] در وراثت استفاده می‌شود. وقتی یک عضو با این سطح دسترسی تعریف شود، کلاس‌های فرزند به آن عضو دسترسی دارند، اما کاربران کلاس به آن عضو دسترسی ندارند.
- ویژگی‌ها و رفتارها معمولاً متعلق به اشیا یا نمونه‌های کلاس هستند، اما گاهی ممکن است نیاز به ویژگی یا رفتاری داشته باشیم که متعلق به کلاس داشته باشد. برای مثال وقتی می‌خواهیم تعداد اشیاء ساخته شده از یک کلاس را بشماریم، متغیر شمارند باید متعلق به کلاس باشد نه اشیاء. اگر ویژگی‌ها و رفتارهای یک کلاس به صورت ایستا تعریف شوند، آن ویژگی‌ها متعلق به کلاس هستند و با تغییر آنها ویژگی همه اشیای کلاس تغییر می‌کند.
- یک کلاس می‌تواند از چندین کلاس به ارث ببرد که به آن وراثت چندگانه [1] گفته می‌شود.

[1] protected
[1] multiple inheritance

چندریختی

- یکی دیگر از قابلیت‌ها در برنامه نویسی شیءگرا چندریختی [1] است که انقیاد پویای توابع [2] نیز نامیده می‌شود. چندریختی در واقع به معنای قابلیت استفاده از یک نام واحد برای توابع متفاوت است.
- توضیح خواهیم داد که چرا به چندریختی انقیاد پویای توابع کلاس نیز گفته می‌شود.
- فرض کنید کلاسی به نام `shape` داشته باشیم که نمایندهٔ همه اشکال هندسی است. حال کلاس‌های متفاوتی از جمله کلاس دایره و مستطیل از این کلاس به ارث می‌برند. همه این کلاس‌ها تابعی به نام `draw` برای رسم شکل دارند که این تابع را از کلاس شکل به ارث می‌برند.

[1] polymorphism
[2] dynamic function binding

چندریختی

- حال فرض کنید می‌خواهیم لیستی از تعداد اشکال متفاوت تشکیل دهیم که همهٔ آنها توسط اشاره‌گری از نوع شکل مشخص شده‌اند. اگر تابع رسم را برای همه اعضا این لیست فراخوانی کنیم در واقع نیاز داریم تابع `draw` را از کلاس‌های متفاوت فراخوانی کنیم. پس در زمان کامپایل مشخص نیست چه تابعی فراخوانی می‌شود، اما در زمان اجرا با توجه به این که یک عضو لیست به چه شکلی اشاره می‌کند باید تابع رسم برای شکل مربوطه فراخوانی شود. بنابراین زمان انقیاد تابع چند ریخت در زمان اجرا است یعنی به طور پویا صورت می‌گیرد و به همین دلیل چند ریختی را انقیاد پویای توابع کلاس نیز می‌گویند.

- وقتی یک کلاس پدر یک رفتار را تعریف می‌کند اما آن را پیاده سازی نمی‌کند به آن رفتار یک رفتار انتزاعی [1] گفته می‌شود. در سی++ به این رفتارها، توابع مجازی خالص [2] گفته می‌شود. کلاسی که حداقل یک رفتار انتزاعی داشته باشد، کلاس انتزاعی گفته می‌شود. از یک کلاس انتزاعی نمی‌توان نمونه ساخت. کلاسی که از یک کلاس انتزاعی به ارث می‌برد باید رفتارهای انتزاعی آن کلاس را پیاده سازی کند، در غیر اینصورت انتزاعی باقی می‌ماند.

[1] abstract
[2] pure virtual function

پیاده سازی زبان شیءگرا

- در زبان سی++، برای تخصیص حافظه به اشیاء در زمان اجرا، برای هر شی یک رکورد نمونهٔ کلاس [1] در نظر گرفته می‌شود که اندازهٔ آن به اندازهٔ مجموع اندازهٔ اعضای داده‌ای کلاس است و دسترسی به اعضای داده‌ای توسط یک آفست یا فاصله از ابتدای رکورد است.
- وقتی یک کلاس از یک کلاس دیگر به ارث می‌برد، در واقع یک کپی از رکورد نمونهٔ کلاس پدر گرفته می‌شود و اندازه‌ای برای اعضای داده‌ای کلاس فرزند به آن اضافه می‌شود.

[1] class instance record

پیاده سازی ساختار شیءگرا

- وقتی یک تابع چندریخت متعلق به یک شیء فراخوانی می‌شود، باید بدانیم تابع چندریخت متعلق به کدام کلاس باید فراخوانی شود.
- برای پیاده‌سازی چندریختی، در رکورد نمونه ساختاری به نام جدول تابع مجازی [1] یا vtable وجود دارد که به ازای هر تابع مجازی در یک رکورد نمونه، آدرس تابع مشخص می‌کند.

[1] virtuel function table (vtable)

پیاده سازی زبان شیءگرا

- انتخاب توابع چندریخت به طور پویا در زمان اجرا صورت می‌گیرد. به عبارت دیگر تنها در زمان اجرا [1] مشخص می‌شود که یک اشاره‌گر به چه شیئی از اشیای چندریخت اشاره می‌کند.

- به طور مثال برنامه‌ای را در نظر بگیرید که در آن کاربر می‌تواند اشکالی را رسم کرده و از بین اشکال رسم شده، یک شکل را انتخاب می‌کند. حال بسته به این که چه شکلی توسط کاربر انتخاب شده است، مساحت توسط یک تابع چندریخت باید محاسبه شود. پس در جایی از برنامه داریم:

```
۱ shape * shp;
۲ if (/*user chooses circle*/) shp = new circle;
۳ else if (/*user chooses rectangle*/) shp = new rectangle;
۴ double area = shp->calcArea();
```

- در زمان کامپایل [2] مشخص نیست کاربر چه شکلی را انتخاب خواهد کرد و `shp` به چه شیئی اشاره می‌کند، پس کد ماشین تولید شده نمی‌تواند آدرس تابع `calcArea` مورد نظر را تعیین کند.

[1] run-time
[2] compile-time

پیاده سازی زبان شیءگرا

- به دلیل این که آدرس تابع مورد نظر برای اجرا در توابع چندریخت به طور پویا در زمان اجرا تعیین می‌شود، چندریختی از سازوکاری به نام انقیاد پویای توابع [1] استفاده می‌کند.
- در مقابل سازوکار انقیاد پویا، انقیاد ایستای توابع [2] وجود دارد. در سربارگذاری توابع از انقیاد ایستا استفاده می‌کنیم.
- به عبارت دیگر، در سربارگذاری توابع، در زمان کامپایل، کامپایلر همهٔ اطلاعات مورد نیاز برای قرار دادن آدرس توابع سربارگذاری شده در کد ماشین را دارد.

[1] dynamic binding
[2] static binding

پیاده سازی زبان شیءگرا

- فرض کنید در یک برنامه، بسته به این که کاربر می‌خواهد با استفاده از نام دانشجو یا شمارهٔ دانشجویی، اطلاعات دانشجو را بیابد، تابع سربارگذاری شدهٔ `getinfo` فراخوانی می‌شود.

```
۱ if (/*user chooses selection by name*/) {
۲     cin >> name; getinfo(name);
۳ } else if (/*user chooses selection by id*/) {
۴     cin >> id; getinfo(id);
۵ }
```

- در زمان کامپایل، کامپایلر می‌تواند دقیقا کد ماشین معادل کد بالا را تولید کرده و آدرس توابع سربارگذاری شده را جایگزین نام توابع کند. پس در زمان اجرا هیچ تصمیم‌گیری صورت نمی‌گیرد.

پیاده سازی زبان شیءگرا

- از آنجایی که انتخاب تابع چند ریخت در زمان اجرا صورت می‌گیرد، چندریختی با استفاده از یک جدول توابع مجازی به نام vtable و یک اشاره‌گر به جدول توابع مجازی به نام vptr پیاده‌سازی می‌شود.
- نحوهٔ پیاده‌سازی چندریختی توسط کامپایلر بدین صورت است که کامپایلر به هر کلاس چندریخت که توابع مجازی را تعریف می‌کند، یک اشاره‌گر vptr اضافه می‌کند که این اشاره‌گر به یک جدول vtable اشاره می‌کند. در این جدول آدرس توابع مجازی که پیاده‌سازی شده‌اند قرار می‌گیرد.
- حال هر تابعی که از یک کلاس چندریخت ارث‌بری کند، طبق قوانین وراثت اشاره‌گر vptr را نیز به ارث می‌برد. اشاره‌گر در کلاس فرزند به جدولی اشاره می‌کند که در آن جدول آدرس توابع کلاس پیاده‌سازی شده در کلاس فرزند ذکر شده است. اگر تابعی چندریخت در کلاس فرزند تعریف نشده باشد، برای آن تابع آدرس تابعی قرار می‌گیرد که نزدیک‌ترین پدر آن را پیاده‌سازی کرده باشد.
- حال در زمان اجرا با استفاده از vptr کامپایلر می‌تواند تصمیم بگیرد چه توابعی را اجرا کند.

پیاده سازی زبان شیءگرا

- برای مثال فرض کنید کلاس A توابع چندریخت f و g را تعریف کرده است. پس کلاس A یک اشارهگر مجازی vptr دارد که به جدول توابع مجازی vtable از کلاس A اشاره میکند. در این جدول آدرس پیادهسازی توابع f و g ذکر شده است.
- حال اگر کلاس B از کلاس A به ارث ببرد، اشارهگر را نیز به ارث میبرد و اشارهگر vptr در کلاس B به جدولی مجازی مربوط به کلاس B اشاره میکند. حال اگر B هیچ کدام از توابع f و g را تعریف نکند، در جدول vtable کلاس B آدرس توابع f و g در کلاس پدر ذکر میشود. اما اگر B هر یک از این توابع را پیادهسازی کند، در جدول توابع مجازی آن، آدرس توابع پیادهسازی شده توسط خود کلاس B ذکر میشود.

پیاده سازی زبان شیءگرا

- حال برنامه زیر را در نظر بگیرید.

```
۱ A aobj; B bobj; A * aptr;
۲ if (/*user chooses A*/) aptr = &aobj;
۳ else if (/*user chooses B*/) aptr = &bobj;
۴ aptr->f(); aptr->g();
```

- فرض کنید کلاس B تنها تابع f را پیاده‌سازی کند. کامپایلر این برنامه را به شکل زیر کامپایل خواهد کرد.

```
۱ A aobj; B bobj; A * aptr;
۲ // aobj.vptr and bobj.vptr are added.
۳ // aobj.vtable contains addresses of f and g implemented in A.
۴ // bobj.vtable contains the address of f implemented in B
۵ // and the address of g implemented in A.
۶ if (/*user chooses A*/) aptr = &aobj;
۷ else if (/*user chooses B*/) aptr = &bobj;
۸ aptr->f(); // => aptr->vptr->vtable[f]();
۹ aptr->g(); //=> aptr->vptr->vtable[g]();
```

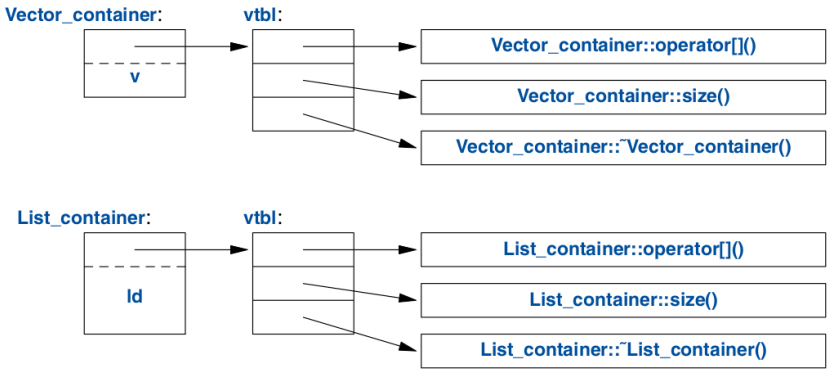
پیاده سازی زبان شیءگرا

```
A aobj; B bobj; A * aptr;
// aobj.vptr and bobj.vptr are added.
// aobj.vtable contains addresses of f and g implemented in A.
// bobj.vtable contains the address of f implemented in B
// and the address of g implemented in A.
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); // => aptr->vptr->vtable[f]();
aptr->g(); //=> aptr->vptr->vtable[g]();
```

- پس در زمان اجرا بسته به اینکه vptr به چه جدولی اشاره کند و در جدول مربوطه چه آدرسی ذکر شده است، تصمیم‌گیری مبنی بر اجرای تابع چندریخت مورد نظر صورت می‌گیرد.

پیاده سازی زبان شیءگرا

- پس اشیای چندریخت که از کلاس‌های انتزاعی به ارث برده‌اند، جدولی به نام جدول تابع مجازی [1] یا vtable در حافظه نگهداری می‌کنند که در آن جدول آدرس توابعی که باید فراخوانی شوند، یادداشت شده است.



[1] virtual function table

پیاده سازی زبان شیءگرا

- در دیباگر gdb می‌توان جدول vtable برای اشاره‌گر ptr را با دستور `-exec info vtbl ptr` مشاهده کرد.

```
-exec info vtbl aptr
vtable for 'A' @ 0x55555556b8f8 (subobject @ 0x7fffffffd0f0):
[0]: 0x55555556370c <A::f()>
[1]: 0x55555556371c <A::g()>
-exec info vtbl aobj
vtable for 'A' @ 0x55555556b8f8 (subobject @ 0x7fffffffd0f0):
[0]: 0x55555556370c <A::f()>
[1]: 0x55555556371c <A::g()>
-exec info vtbl bobj
vtable for 'B' @ 0x55555556b8d8 (subobject @ 0x7fffffffd100):
[0]: 0x55555556372c <B::f()>
[1]: 0x55555556371c <A::g()>
```

مسائل طراحی زبان شیءگرا

- تعدادی از مسائل در زمان طراحی یک زبان شیءگرا باید مورد بررسی قرار بگیرند.
- در طراحی یک زبان شیءگرا می‌توان همهٔ نوع‌های داده‌ای را به صورت کلاس تعریف کرد. بدین ترتیب از اعداد ساده گرفته تا ساختارهای پیچیده همگی کلاس هستند. این طراحی گرچه یکنواخت و یکپارچه است اما یک نقص نیز دارد. وقتی نوع‌های اساسی مانند عدد صحیح و اعشاری به صورت کلاس طراحی شوند تمام سربارهایی که تعریف یک کلاس دارد برای این نوع‌ها نیز وجود خواهد داشت. این سربار باعث کاهش راندمان و افزایش زمان اجرا می‌شود.
- راه حلی که برای این مسئله در نظر گرفته شده است این است که در بیشتر زبان‌های شیءگرا نوع‌های اساسی و ابتدایی [1] بدون شیءگرایی همانند برنامه نویسی رویه‌ای طراحی شده‌اند و نوع‌های دیگر به صورت کلاس.

[1] primitive type

مسائل طراحی زبان شیءگرا

- یکی دیگر از مسائل در طراحی زبان‌های شیءگرا این است که آیا یک زبان باید وراثت چندگانه را پشتیبانی کند یا خیر.
- برای روشن شدن این مسئله مثال زیر را در نظر بگیرید. فرض کنید کلاس A و B هر دو تابع f را پیاده سازی کنند. حال کلاس C از کلاس A و B به صورت وراثت چندگانه به ارث می‌برد. حال فرض کنید تابع f از کلاس C فراخوانی شود، در این حالت فراخوانی مبهم است، زیرا مشخص نیست کامپایلر باید تابع f از کلاس A را فراخوانی کند یا تابع f از کلاس B .
- زبان سی++ وراثت چندگانه را پشتیبانی می‌کند و در چنین مواقعی پیام خطا با محتوای ابهام در فراخوانی صادر می‌کند ولی زبان جاوا وراثت چندگانه را پشتیبانی نمی‌کند.

مسائل طراحی زبان شیءگرا

- زبان جاوا وراثت چندگانه را در کلاس‌های رابط [1] پشتیبانی می‌کند. یک کلاس رابط در واقع کلاسی است که در آن پیاده سازی وجود ندارد، بنابراین وراثت چندگانه ابهامی ایجاد نمی‌کند، زیرا هیچ پیاده سازی وجود ندارد که ابهام در فراخوانی به وجود آورد.

[1] interface

مسائل طراحی زبان شیءگرا

- وراثت چندگانه در زبان سی++ یک مشکل دیگر نیز ایجاد می‌کند. فرض کنید کلاس A متغیر x را تعریف می‌کند. سپس کلاس‌های B و C هر دو از A به ارث می‌برند. کلاس D از کلاس B و C به صورت وراثت چندگانه به ارث می‌برد. حال اگر D بخواهد به متغیر x دسترسی پیدا کند مشخص نیست آیا به متغیر x که از طریق A به ارث رسیده است دسترسی پیدا می‌کند یا به متغیر x که از B به ارث رسیده است. در این حالت نیز ابهام به وجود می‌آید زیرا هر کدام از کلاس‌های B و C ممکن است مقداری متفاوت برای متغیر x تعیین کرده باشند. این مشکل، مشکل لوزی [1] نامیده می‌شود، زیرا A و B و C و D یک لوزی می‌سازند. برای حل این مشکل لوزی، زبان سی++ وراثت مجازی را تعریف کرده است. وقتی B و C به صورت مجازی از A به ارث می‌برند، هر دو برای متغیر x یک خانه حافظه تخصیص می‌دهند، بنابراین در دسترسی D به متغیر x ابهامی به وجود نمی‌آید.
- طراحان زبان جاوا تصمیم گرفته‌اند وراثت مجازی را ممنوع کنند، زیرا عقیده داشته‌اند مشکلاتی که وراثت چندگانه به وجود می‌آورد بیشتر است از مشکلاتی که می‌تواند در طراحی حل کند.

[1] diamond problem

مسائل طراحی زبان شیءگرا

- در زبان پایتون نیز وراثت چندگانه پشتیبانی می‌شود ولی ابهامی به وجود نمی‌آورد چرا که اگر دو پدر یک تابع یکسان پیاده سازی کرده باشند، تابع فرزند در فراخوانی تابع مذکور اولویت را به پدر اول میدهد.

```
۱  class B :
۲      def f() :
۳          print ("B")
۴  class C :
۵      def f() :
۶          print ("C")
۷  class D(B,C) :
۸      pass
۹  d = D()
۱۰ d.f() # B
```

مسائل طراحی زبان شیءگرا

- مسئلهٔ دیگری که در طراحی شیءگرا وجود دارد، مسئلهٔ تخصیص حافظه و آزادسازی حافظه است. هر کلاس ممکن است تعداد زیادی ویژگی داشته باشد که هر کدام از این ویژگی‌ها ممکن است آرایه یا اشیایی از کلاس‌های دیگر باشد و بنابراین ممکن است تعداد زیادی از ویژگی‌ها بر روی هیپ ساخته شوند.
- وقتی یک شیء تخریب می‌شود، برنامه نویس باید مراقب باشد که همهٔ حافظه‌های تخصیص داده شده در هیپ را آزاد کند در غیر اینصورت نشست حافظه رخ می‌دهد. طراح زبان دو راه پیش رو دارد. اگر آزادسازی حافظه را به برنامه نویس محول کند قابلیت اطمینان برنامه پایین می‌آید ولی در عوض برنامه می‌تواند با سرعت بالاتری اجرا شود. اگر آزادسازی حافظه به طور خودکار انجام شود، قابلیت اطمینان برنامه بالا می‌رود ولی سربار اضافی تحمیل شده بابت آزادسازی حافظه به صورت خودکار سرعت اجرای برنامه را پایین می‌آورد.
- زبان سی++ روش اول و جاوا روش دوم را در پیش گرفته است. پایتون نیز دارای مکانیزم بازیافت حافظه خودکار است. زبان راست راه حل سومی در پیش گرفته است. برنامه‌نویس بر روی حافظه کنترل دارد ولی باید از قوانینی که زبان تعیین کرده است پیروی کند و در صورتی که آزادسازی حافظه به درستی صورت نگرفته باشد، کامپایلر پیام خطا صادر می‌کند.

جاوا

- در اینجا به پاره‌ای از تفاوت‌ها بین سی++ و جاوا اشاره می‌کنیم.
- کلاس‌ها در سی++ می‌توانند بدون کلاس پدر باشند، اما در جاوا همهٔ کلاس‌ها باید کلاس مافوق داشته باشند.
- کلاس `Object` در بالاترین رده در سلسله مراتب کلاس‌هاست.
- کلاس `Object` تعدادی متود از جمله `toString` برای تبدیل شیء به رشته و `equals` برای مقایسه برابری دو شیء دارد.
- همهٔ شیءها در جاوا بر روی هیپ ساخته می‌شوند برخلاف سی++ که در آن اشیاء می‌توانند بر روی پشته یا بر روی هیپ ساخته شوند.

- عملگر delete برای آزادسازی حافظه در جاوا وجود ندارد چرا که بازیافت حافظه به صورت خودکار انجام می‌شود. بازیافت کنندهٔ حافظه تنها بر روی حافظه کنترل دارد و بر روی منابع دیگر هیچ کنترلی ندارد. بنابراین اگر یک شیء در حال خواندن یک فایل باشد و شیء مورد نظر توسط بازیافت کنندهٔ حافظه از بین برود، بازیافت کننده هیچ عملیاتی بر روی فایل انجام نمی‌دهد. برای اینکه برنامه نویس بتواند در هنگام تخریب شیء عملیاتی را پیش‌بینی کند، متود `finalize` می‌تواند پیاده سازی شود. این متود در هنگام تخریب شیء فراخوانی می‌شود، اما مشخص نیست دقیقا چه زمانی یک شیء تخریب می‌شود.

- بر خلاف سی++، جاوا وراثت چندگانه را پشتیبانی نمی‌کند، اما می‌توان کلاس‌های واسط یا `interface` هایی تعریف کرد که هیچ پیاده سازی ندارند اما وراثت چندگانه را پشتیبانی می‌کنند.
- یک کلاس واسط متغیر متعلق به کلاس ندارد و همچنین هیچ سازنده‌ای برای آن تعریف نمی‌شود. در یک کلاس واسط تنها متودها اعلام می‌شوند اما تعریف نمی‌شوند.
- یک کلاس واسط می‌تواند از چند کلاس واسط به ارث ببرد و همچنین یک کلاس می‌تواند چندین کلاس واسط را پیاده سازی کند، اما یک کلاس تنها از یک کلاس به ارث می‌برد.
- وقتی یک کلاس، یک کلاس واسط را پیاده سازی می‌کند، باید همهٔ متودهای آن را تعریف کند.

جاوا

- در سی++ تنها توابعی که به صورت مجازی با کلمهٔ `virtual` تعریف شوند چند ریخت هستند، اما در جاوا همه متودهایی که به صورت عمومی تعریف شوند می‌توانند چندریخت باشند.
- قابلیت چند ریختی توسط کلاس‌های واسط نیز می‌تواند مورد استفاده قرار بگیرد. برای مثال یک متود می‌تواند به عنوان پارامتر یک کلاس واسط دریافت کند. سپس به عنوان آرگومان این متود، باید کلاسی ارسال شود که کلاس واسط مذکور را پیاده سازی کرده باشد.
- یک کلاس در جاوا می‌تواند با استفاده از کلمهٔ abstract به صورت انتزاعی تعریف شود. در یک کلاس انتزاعی حداقل یکی از متودها انتزاعی است یعنی هیچ تعریفی ندارد. از یک کلاس انتزاعی نمی‌توان شیء ساخت. کلاس انتزاعی جاوا معادل کلاس انتزاعی سی++ است که حداقل یکی از توابع آن مجازی خالص است.

راست : مخفی‌سازی اطلاعات

- گرچه راست یک زبان‌های شیءگرا نیست، اما مخفی‌سازی اطلاعات [1] در زبان راست توسط ماژول‌ها امکان‌پذیر است.

[1] information hiding

راست : مخفی‌سازی اطلاعات

- وقتی یک ساختمان (یا یک تابع) در یک ماژول تعریف شود، به طور پیش‌فرض سطح دسترسی آن از بیرون ماژول به صورت خصوصی [1] است، مگر اینکه به طور صریح سطح دسترسی عمومی [2] برای آن تعریف شود.

```
۱  mod module {
۲  pub struct A {
۳      pub x : i32, // x is a public field
۴      y : i32, // y is a private field
۵  }
۶  }
۷  fn main() {
۸      let var = module::A { x : 1, y : 2};
۹      println!("var.x = {}", var.x);
۱۰     println!("var.y = {}", var.y); // error : y is a private field
۱۱ }
```

[1] private
[2] public

راست : نوع عمومی

- در زبان راست یک ساختمان را می‌توانیم از نوع دادهٔ عمومی [1] تعریف کنیم و بدین ترتیب نوع اعضای ساختمان را به عنوان پارامتر به ساختمان ارسال کنیم.

```
۱ struct Point<T> {
۲     x: T,
۳     y: T,
۴ }
۵ fn main() {
۶     let ipoint = Point { x: 5, y: 10 };
۷     let fpoint = Point { x: 1.0, y: 4.0 };
۸ }
```

[1] generic data type

راست : نوع عمومی

- یک ساختمان عمومی می‌تواند بیش از یک پارامتر نوع نیز دریافت کند.

```
۱ struct Point<T, U> {
۲     x: T,
۳     y: U,
۴ }
۵ fn main() {
۶     let both_integer = Point { x: 5, y: 10 };
۷     let both_float = Point { x: 1.0, y: 4.0 };
۸     let integer_and_float = Point { x: 5, y: 4.0 };
۹ }
```

راست : نوع عمومی

- نوع دادهٔ شمارشی نیز می‌تواند به صورت عمومی تعریف شود.

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

راست : نوع عمومی

- در پیاده سازی متودهای یک ساختمان نیز می‌توانیم از نوع عمومی استفاده کنیم. قرار دادن نوع T بعد از impl به کامپایلر می‌گوید T یک نوع عمومی است نه یک نوع خاص.

```
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

راست : نوع عمومی

- ممکن است بخواهیم برای پیاده‌سازی یک نوع خاص یک تابع جداگانه تعریف کنیم. در اینصورت می‌توانیم پیاده سازی ساختمان را به صورت زیر انجام دهیم.

```
۱ impl Point<f32> {
۲     fn distance_from_origin(&self) -> f32 {
۳         (self.x.powi(2) + self.y.powi(2)).sqrt()
۴     }
۵ }
```

راست : نوع عمومی

- پارامترهای نوع یک ساختمان می‌توانند با پارامتر متودها تفاوت داشته باشند.

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}
impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
```

راست : نوع عمومی

```
۱ fn main() {
۲     let p1 = Point { x: 5, y: 10.4 };
۳     let p2 = Point { x: "Hello", y: 'c' };
۴     let p3 = p1.mixup(p2);
۵     println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
۶ }
```

راست : نوع عمومی

- استفاده از نوع‌های عمومی هیچ سرباری بر سرعت اجرای برنامه ندارد. در زمان کامپایل همهٔ کدهای عمومی توسط کامپایلر به نوع‌های اصلی و ساخته شده توسط کاربر تبدیل می‌شوند.

راست : ویژگی‌های مشترک

- از آنجایی که راست یک زبان شیءگرا نیست، نمی‌توان توسط وراثت از یک ساختمان دیگر، ویژگی‌های آن را به ارث برد.
- برای تعریف ویژگی‌های مشترک بین کلاس‌ها، می‌توان یک فیلد با نوع مشترک در چند ساختمان تعریف کرد.

```
۱ struct Person {
۲     pid : i64, firstname : String, lastname  : String,
۳ }
۴ struct Student {
۵     pinfo : Person, sid : i64, average : i64,
۶ }
۷ struct Teacher {
۸     pinfo : Person, tid : i64, salary : i64,
۹ }
```

راست : رفتار مشترک

- با استفاده از رابط‌ها [1] می‌توانیم رفتار مشترک [2] برای چند نوع متفاوت تعریف کنیم.
- برای این کار ابتدا باید آن رابط را تعریف کنیم و سپس رابط را برای نوع‌های متفاوت تعریف کنیم.

```
۱ trait Summary {
۲     fn summarize(&self) -> String;
۳ }
```

[1] trait
[2] shared behavior

راست : رفتار مشترک

- حال برای ساختمان‌های متفاوت می‌توانیم آن رابط را پیاده سازی کنیم.

```
۱  struct NewsArticle {
۲      headline: String,
۳      location: String,
۴      author: String,
۵      content: String,
۶  }
۷  impl Summary for NewsArticle {
۸      fn summarize(&self) -> String {
۹       format!("{}, by {} ({})", self.headline, self.author,
۱۰                                              self.location)
۱۱     }
۱۲ }
```

راست : رفتار مشترک

```rust
struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

راست : رفتار مشترک

- در زبان راست وراثت وجود ندارد. بدین معنی که یک ساختمان نمی‌تواند از یک ساختمان دیگر به ارث ببرد. اما چند ساختمان می‌توانند یک رفتار مشترک داشته باشند.
- همچنین چند ریختی به شکلی که در زبان‌های شیءگرا وجود دارد، در راست وجود ندارد اما ساختمان‌هایی که یک رفتار مشترک داشته باشند می‌توانند به عنوان پارامتر به توابع ارسال شوند.
- راست بدین دلیل از وراثت استفاده نمی‌کند که در وراثت معمولاً کدهایی که بین کلاس‌ها به اشتراک گذاشته می‌شوند، بیش از حد نیاز هستند. برای مثال کلاس‌های فرزند به همهٔ توابع پدر و اجداد نیاز ندارند، اما آنها را به ارث می‌برند. شیءگرایی اساساً برای پیاده سازی مفهوم رفتار مشترک تعریف شده است که این رفتارهای مشترک در راست، توسط رابط‌ها پیاده سازی می‌شوند.

راست : رفتار مشترک

- حال فرض کنید می‌خواهیم یک برنامه واسط کاربر بنویسیم. یک صفحه شامل چندین عنصر است که هر کدام از آنها باید در پنجره یا صفحه برنامه رسم شوند. یک عنصر می‌تواند یک دکمه یا یک جعبهٔ انتخاب یا اشیای دیگر باشد.
- بنابراین باید ابتدا این رفتار مشترک را تعریف کنیم.
- رفتار مشترک عناصر قابلیت رسم کردن آنها است که آن را به عنوان یک رابط به صورت زیر تعریف می‌کنیم.

```
۱ trait Draw {
۲     fn draw(&self);
۳ }
```

راست : رفتار مشترک

- یک صفحه یا Screen تشکیل شده است از تعدادی عناصر که هر کدام از آنها باید ساختمانی باشد که رابط رسم با Draw را پیاده سازی کرده باشد.

```
۱ struct Screen {
۲     components: Vec<Box<dyn Draw>>,
۳ }
```

- در اینجا Box یک اشارهگر هوشمند تعریف میکند که زمانی به کار میرود که اندازه یک نوع در زمان کامپایل نامشخص است. اشارهگر هوشمند فضایی بر روی هیپ تخصیص میدهد.
- همچنین dyn مشخص میکند که Draw یک رابط است، نه یک ساختمان خاص.
- با نوشتن عبارت dyn Trait کامپایلر یک اشارهگر مجازی به جدول مجازی برای نوع مورد نظر ایجاد میکند.

راست : رفتار مشترک

- حال برای صفحه تابعی پیاده سازی می‌کنیم که همه عناصر درون خود را رسم کند.

```
impl Screen {
    fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

راست : رفتار مشترک

- حال یک دکمه یا Button را تعریف می‌کنیم که رابط رسم یا Draw را پیاده سازی می‌کند.

```
struct Button {
    width: u32,
    height: u32,
    label: String,
}
impl Draw for Button {
    fn draw(&self) {
        // code to actually draw a button
    }
}
```

راست : رفتار مشترک

- می‌توانیم یک جعبهٔ انتخاب یا SelectBox و هر شیء دیگری را به همین شکل تعریف کنیم و رابط رسم را برای آنها پیاده سازی کنیم.

```
struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}
impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

راست : رفتار مشترک

- حال برنامه‌ای می‌نویسیم که شامل دو متغیر از نوع‌های دکمه و جعبهٔ انتخاب است و این برنامه دو تابع رسم متفاوت را برای این متغیرها فراخوانی می‌کند.

```
fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
```

راست : رفتار مشترک

```
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };
    screen.run();
}
```

راست : رفتار مشترک

- رفتارهای یک رابط می‌توانند پیاده سازی پیش فرض داشته باشد. بدین ترتیب یک ساختمان می‌تواند برخی رفتارهای یک رابط را پیاده سازی کند و برای برخی دیگر از پیاده سازی پیش فرض استفاده کند.
- برای مثال :

```
۱ trait Summary {
۲     fn summarize(&self) -> String {
۳         String::from("(Read more...)")
۴     }
۵ }
```

راست : رفتار مشترک

- همچنین رفتارهای پیش‌فرض یک رابط می‌توانند دیگر رفتارها را فراخوانی کنند.

```
۱ trait Summary {
۲     fn summarize_author(&self) -> String;
۳     fn summarize(&self) -> String {
۴         format!("(Read more from {}...)", self.summarize_author())
۵     }
۶ }
```

راست : رفتار مشترک

- رابط‌ها را می‌توان به عنوان پارامتر نیز به توابع دیگر ارسال کرد. بدین ترتیب قابلیت چندریختی زبان‌های شیءگرا در زبان راست قابل استفاده است.

```
۱ fn notify(item: &impl Summary) {
۲     println!("Breaking news! {}", item.summarize());
۳ }
```

راست : رفتار مشترک

- روش قبلی برای ارسال یک رابط به یک تابع درواقع معادل کد زیر است که یک نوع عمومی به تابع ارسال می‌کند.

```
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

راست : رفتار مشترک

- ممکن است یک تابع نیاز داشته باشد که پارامتر ورودی آن ساختمانی باشد که چند رابط را پیاده سازی کرده باشد. در این صورت می‌توانیم از عملگر + به صورت زیر استفاده کنیم.

```
fn notify(item: &(impl Summary + Display)) {
```

راست : رفتار مشترک

- کد قبل در واقع یک میانبر برای کد زیر است.

```
fn notify<T: Summary + Display>(item: &T) {
```

راست : رفتار مشترک

- وقتی تعداد زیادی رابط در تعریف توابع داشته باشیم، ممکن است کد پیچیده شود. در چنین مواقعی می‌توانیم از یک روش دیگر برای ساده‌سازی کد استفاده کنیم.
- برای مثال :

```
fn some_function<T: Display + Clone,
                 U: Debug + Clone>(t: &T, u: &U) -> i32
// it is equivalent to :
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Debug + Clone,
```