In The Name of Allah

# «ترکیب و وراثت»

# فهرست مطالب

- مباحث
  - مقدمه
  - تركيب
  - وراثت
  - اعضاي حفاظت شد
  - غلبه كردن بر وراثت
  - اشاره‌گرها در وراثت
  - توابع مجازي و چندريختي
  - نابودكنندهٔ مجازي

# مقدمه

اغلب اوقات برای ایجاد یک کلاس جدید، نیازی نیست که همه چیز از اول طراحی شود. می‌توانیم برای ایجاد کلاس مورد نظر، از تعاریف کلاس‌هایی که قبلا ساخته‌ایم، استفاده نماییم.

این باعث صرفه‌جویی در وقت و استحکام منطق برنامه می‌شود.

در شی‌گرایی به دو شیوه می‌توان این کار را انجام داد: ***ترکیب*** ۱ و ***وراثت*** ۲. در این جلسه خواهیم دید که چگونه و چه مواقعی می‌توانیم از این دو شیوه بهره ببریم.

# ترکیب:

ترکیب کلاس‌ها (یا تجمیع کلاس‌ها) یعنی استفاده از یک یا چند کلاس دیگر در داخل تعریف یک کلاس جدید.
هنگامی که عضو داده‌ای کلاس جدید، شیئی از کلاس دیگر باشد، می‌گوییم که این کلاس جدید ترکیبی از سایر کلاس‌هاست.
به تعریف دو کلاس زیر نگاه کنید.

# کلاس Date

کد زیر، کلاس Date را نشان می‌دهد که اشیای این کلاس برای نگهداری تاریخ استفاده می‌شوند.

```
class Date
{  public:
     Date(int y=0, int m=0, int d=0) :
          year(y), month(m), day(d) {};
     void setDate(int y, int m, int d)
          {  year = y; month = m; day = d;  }
     void getDate()
          {  cin >> year >> month >> day ;  }
     void showDate()
          {  cout << year << '/' << month << '/' << day ;  }
   private:
     int year, month, day;
}
```

# کلاس Book

کد زیر، کلاس Book را نشان می‌دهد که اشیای این کلاس برخی از مشخصات یک کتاب را نگهداری می‌کنند:

```
class Book
{  public:
      Book(char* n = " ", int i = 0, int p = 0) :
         name(n), id(i), page(p) { }
      void printName() {  cout << name;  }
      void printId()   {  cout << id;    }
      void printPage() {  cout << page;  }
   private:
      string name, author;
      int id, page;
}
```

# بهبود دادن کلاس Book

```
"include "Date.h#
class Book
{  public:
      Book(char* n = " ", int i = 0, int p = 0) :
           name(n), id(i), page(p) { }
      void printName() {  cout << name;  }
      void printId()   {  cout << id;    }
      void printPage() {  cout << page;  }
      void setDOP(int y, int m, int d)
                        { publish.setDate(y, m, d) ; }
      void showDOP() { publish.showDate(); }
   private:
      string name, author;
      int id, page;
      Date publish;
}
```

# وراثت:

وراثت روش دیگری برای ایجاد کلاس جدید از روی کلاس قبلی است. گاهی به وراثت «اشتقاق» نیز می‌گویند.

اگر از قبل با برنامه‌نویسی مبتنی بر پنجره‌ها آشنایی مختصر داشته باشید، احتمالا عبارت «کلاس مشتق‌شده» را فراوان دیده‌اید. این موضوع به خوبی اهمیت وراثت را آشکار می‌نماید.

# اعضای حفاظت شده :

گرچه کلاس Ebook در مثال قبل نمی‌تواند مستقیما به اعضای خصوصی کلاس والدش دسترسی داشته باشد، اما با استفاده از توابع عضو عمومی که از کلاس والد به ارث برده، می‌تواند به اعضای خصوصی آن کلاس دستیابی کند. این محدودیت بزرگی محسوب می‌شود. اگر توابع عضو عمومی کلاس والد انتظارات کلاس فرزند را برآورده نسازند، کلاس فرزند ناکارآمد می‌شود. اوضاع زمانی وخیم‌تر می‌شود که هیچ تابع عمومی برای دسترسی به یک دادهٔ خصوصی در کلاس والد وجود نداشته باشد.

# غلبه کردن بر وراثت:

اگر Y زیر کلاسی از X باشد، آنگاه اشیای Y همهٔ اعضای عمومی و حفاظت شدهٔ کلاس X را ارث می‌برند.

مثلا تمامی اشیای Ebook تابع دستیابی printName() از کلاس Book را به ارث می‌برند. به تابع printName() یک «عضو موروثی» می‌گوییم. گاهی لازم است یک نسخهٔ محلی از عضو موروثی داشته باشیم.

یعنی کلاس فرزند، عضوی هم نام با عضو موروثی داشته باشد که مخصوص به خودش باشد و ارثی نباشد. برای مثال فرض کنید کلاس X یک عضو عمومی به نام p داشته باشد و کلاس Y زیر کلاس X باشد.

در این حالت اشیای کلاس Y عضو موروثی p را خواهند داشت. حال اگر یک عضو به همان نام p در زیرکلاس Y به شکل صریح اعلان کنیم، این عضو جدید، عضو موروثی هم‌نامش را مغلوب می‌کند.

به این عضو جدید، «عضو غالب» می‌گوییم. بنابراین اگر y1 یک شی از کلاس Y باشد، y1.p به عضو p غالب اشاره دارد نه به p موروثی.

البته هنوز هم می‌توان به p موروثی دسترسی داشت. عبارت y1.X::p به p موروثی دستیابی دارد.

هم می‌توان اعضای داده‌ای موروثی را مغلوب کرد و هم اعضای تابعی موروثی را.

یعنی اگر کلاس X دارای یک عضو تابعی عمومی به نام f() باشد و در زیرکلاس Y نیز تابع f() را به شکل صریح اعلان کنیم، آنگاه y1.f() به تابع غالب اشاره دارد و y1.X::f() به تابع موروثی اشاره دارد. در برخی از مراجع به توابع غالب override می‌گویند و داده‌های غالب را dominate می‌نامند. ما هر دو مفهوم را به عنوان اعضای غالب به کار می‌بریم. به مثال زیر نگاه کنید.

# اعضای داده‌ای و تابعی غالب:

```
class X
{  public:
      void f() { cout << "Now X::f() is running\n"; }
      int a;
};
class Y : public X
{  public:
      void f() { cout << "Now Y::f() is running\n"; }
      // this f() overrides X::f()
      int a;
      // this a dominates X::a
};
```

# سازنده‌ها و نابودکننده‌های والد:

```
class X
{  public:
      X() { cout << "X::X() constructor executing\n"; }
      ~X() { cout << "X::X() destructor executing\n"; }
};
clas Y : public X
{  public:
      Y() { cout << "Y::Y() constructor executing\n"; }
      ~Y() { cout << "Y::Y() destructor executing\n"; }
};
clas Z : public Y
{  public:
     Z(int n) {cout << "Z::Z(int) constructor executing\n";}
     ~Z() { cout << "Z::Z() destructor executing\n"; }
};
int main()
{  Z z(44);
}
```

# اشاره‌گرها در وراثت:

در شی‌گرایی خاصیت جالبی وجود دارد و آن این است که اگر p اشاره‌گری از نوع کلاس والد باشد، آنگاه p را می‌توان به هر فرزندی از آن کلاس نیز اشاره داد. به کد زیر نگاه کنید:

```
class X
{  public:
      void f();
}
class Y : public X              // Y is a subclass of X
{  public:
      void f();
}
int main()
{  X* p;         // p is a pointer to objects of base class X
   Y y;
   p = &y;       // p can also point to objects of subclass Y
}
```

# اشاره‌گری از کلاس والد به شیئی از کلاس فرزند :

در برنامهٔ زیر، کلاس Y زیرکلاسی از X است. هر دوی این کلاس‌ها دارای یک عضو تابعی به نام f() هستند و p اشاره‌گری از نوع X* تعریف شده:

```
class X
{  public:
      void f() { cout << "X::f() executing\n"; }
};
class Y : public X
{  public:
      void f() { cout << "Y::f() executing\n"; }
}
int main()
{  X x;
   Y y;
   X* p = &x;
   p->f();        // invokes X::f() because p has type X*
   p = &y;
   p->f();        // invokes X::f() because p has type X*
}
```

# توابع مجازی و چندریختی:

*تابع مجازی* تابعی است که با کلمهٔ کلیدی virtual مشخص می‌شود. وقتی یک تابع به شکل مجازی اعلان می‌شود، یعنی در حداقل یکی از کلاس‌های فرزند نیز تابعی با همین نام وجود دارد. توابع مجازی امکان می‌دهند که هنگام استفاده از اشاره‌گرها، بتوانیم بدون در نظر گرفتن نوع اشاره‌گر، به توابع شیء جاری دستیابی کنیم. به مثال زیر دقت کنید.

# استفاده از توابع مجازی:

```
class X
{  public:1 – Virtual function
      virtual void f() { cout << "X::f() executing\n"; }
};
class Y : public X
{  public:
      void f() { cout << "Y::f() executing\n"; }
}
int main()
{  X x;
   Y y;
   X* p = &x;
   p->f();        // invokes X::f()
   p = &y;
   p->f();        // invokes Y::f()
}
```

# چندریختی از طریق توابع مجازی:

سه کلاس زیر را در نظر بگیرید. بدون استفاده از توابع‌مجازی، برنامه آن طور که مورد انتظار است کار نمی‌کند:

```
class Book
{  public:
      Book(char* s) { name = new char[strlen(s+1)];
                      strcpy(name, s);
                    }
      void print() { cout << "Here is a book with name "
                          << name << ".\n";
                   }
   protected:
      char* name;
};
```

```
class Ebook : public Book
{  public:
      Ebook(char* s, float g) : Book(s), size(g) {}
      void print() { cout << "Here is an Ebook with name "
                          << name << " and size "
                          << size << " MB.\n";
                   }
   private:
      float size;
}
```

```
class Notebook : public Book
{  public:
      Notebook(char* s, int n) : Book(s) , pages(n) {}
      void print() { cout << "Here is a Notebook with name "
                          << name << " and " << pages
                          << " pages.\n";
                   }
   private:
      int pages;
};
```

# نابودکنندهٔ مجازی:

با توجه به تعریف توابع مجازی، به نظر می‌رسد که نمی‌توان توابع سازنده و نابودکننده را به شکل مجازی تعریف نمود زیرا سازنده‌ها و نابودگرها در کلاس‌های والد و فرزند، هم‌نام نیستند. در اصل، سازنده‌ها را نمی‌توان به شکل مجازی تعریف کرد اما نابودگرها قصهٔ دیگری دارند.

مثال بعدی ایراد مهلکی را نشان می‌دهد که با مجازی کردن نابودگر، برطرف می‌شود.

# حافظهٔ گم شده:

## به برنامهٔ زیر دقت کنید:

```
class X
{  public:
      x() { p = new int[2];  cout << "X().   "; }
      ~X() { delete [] p;  cout << "~X().\n" }
   private:
      int* p;
};
class Y : public X
{  public:
      Y() { q = new int[1023];  cout << "Y() : Y::q = " << q
    << ".   "; }
       ~Y() { delete [] q;  cout << "~Y().   "; }
   private:
      int* q;
};
int main()
{  for (int i=0; i<8; i++)
   {  X* r = new Y;
      delete r;
   }
}
```

# کلاس‌های پایهٔ انتزاعی:

در شی‌گرایی رسم بر این است که ساختار برنامه و کلاس‌ها را طوری طراحی کنند که بتوان آن‌ها را به شکل یک نمودار درختی شبیه زیر نشان داد: