Dziedziczenie

Klasy i dziedziczenie to jedne z najważniejszych (jeśli nie najważniejsze) mechanizmów współczesnych języków programowania wysokiego poziomu. Są podstawą programowania zorientowanego obiektowo bez którego tworzenie rozbudowanych i skomplikowanych systemów byłoby zdecydowanie trudniejsze i bardziej czasochłonne.

Dziedziczenie, jak sama nazwa wskazuje, pozwala odziedziczyć zmienne i metody innej klasy. Klasę która dziedziczy nazywamy klasą pochodną, natomiast tę po której klasa pochodna dziedziczy – klasą bazową. Pozwala to wyciągnąć wspólne elementy wielu klas do klas bazowych i dzięki temu uniknąć redundancji (powielania) kodu, a w efekcie otrzymać dobrze zaprojektowany system. Z jednej klasy bazowej można uzyskać dowolną liczbę klas pochodnych. Po klasie pochodnej również można dziedziczyć – w efekcie klasa dziedzicząca posiada metody i pola zarówno klasy po której bezpośrednio dziedziczy, jak i klasy którą stoi na najwyższym poziomie hierarchii dziedziczenia (dziedziczy cechy wszystkich klas z hierarchii dziedziczenia). Przykład:

Klasa A stanowi klasę bazową.

Klasa B dziedziczy po klasie A.

Klasa C dziedziczy po klasie B.

Klasa C w efekcie ma dostęp zarówno do metod i pól klasy B jak i klasy A.

Dla zainteresowanych: Dziedziczenie obejmuje metody i pola publiczne (w większości języków programowania także `protected`). Oznaczenie zmiennej lub metody jak `private` uniemożliwia dostęp do niej w klasie pochodnej.

Przykład:

Wyobraźmy sobie, że mamy do zrealizowania system, który przechowuje informacje o uczniach i nauczycielach jakiejś szkoły. Pewne informacje takie jak: imię i nazwisko czy wiek są wspólne dla obu grup, ale każda z nich ma także cechy indywidualne: przy nauczycielach mogą znaleźć się chociażby informacje o zarobkach a przy uczniach o ich ocenach. Idealnie będzie więc wyciągnąć cechy wspólne do klasy bazowej `SchoolMember`, a następnie stworzyć klasy Student i `Teacher` rozszerzające klasę bazową o potrzebne pola i metody:

```
class SchoolMember:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")


class Teacher(SchoolMember):
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))
```

```
    def tell(self):
        SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))


class Student(SchoolMember):
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Initialized Student: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Marks: "{:d}"'.format(self.marks))
```

W kodzie możemy zauważyć następujący fragment:

```
SchoolMember.__init__(self, name)
```

Kod ten odpowiada za wywołanie konstruktora klasy bazowej – po to, aby pola które są po niej dziedziczone również zostały zainicjalizowane. Analogicznie możemy wywołać dowolne, inne metody klasy bazowej.

Możemy także zauważyć, że zarówno klasa bazowa jak i klasy pochodne implementują metodę `tell`(). Należy wiedzieć, że implementacja w klasie pochodnej „przesłoni" tę z klasy bazowej.

Polimorfizm

Polimorfizm jest to mechanizm który powala traktować różne typy danych w taki sam sposób. Znaczy to mniej więcej tyle, że jeśli pewna grupa różnych obiektów różnych klas ma pewną wspólną część, na przykład dziedziczy po jednej klasie bazowej to możemy operować na wszystkich tych obiektach traktując je jako instancje klasy bazowej – nie potrzebujemy wiedzieć jakie to są konkretnie typy, bo wiemy, że jako klasy pochodne mają wszystkie cechy klasy bazowej.

Przykład:

Załóżmy, że mamy klasę `Figura`, która ma zadeklarowaną metodę `pole`(), oraz klasy `Kwadrat` i `Trójkąt` które dziedziczą po klasie `Figura` i we właściwy dla siebie sposób implementują metodę pole(). Mając następnie kolekcję obiektów typu Kwadrat i `Trójkąt` i chcąc dowiedzieć się jakie pole mają, możemy to zrobić tylko za pomocą interfejsu klasy bazowej – nie interesuje nas czy pod spodem są to kwadraty czy trójkąty bo wiemy, że zarówno jedne jak i drugie mają metodę `pole`(), która zwróci nam potrzebą wartość. Co więcej: możemy je przechowywać we wspólnej kolekcji (np. liście) jako obiekty typu `Figura`. Interpreter (dla innych języków kompilator), sam rozpozna jaki typ obiektu znajduję się tam w rzeczywistości i wywoła jego implementację metody `pole`().

Dziedziczenie pozwala nam wyodrębnić pewne wspólne cechy obiektów, natomiast polimorfizm ułatwia operowanie na nich.

Klasy i metody abstrakcyjne

Klasy abstrakcyjne są to klasy, które nie mają swoich reprezentantów. W przykładzie z Figurami i polem klasą abstrakcyjną jest `Figura`. Klasa ta pozwala nam wyciągnąć wspólną logikę wszystkich figur i udostępnia pewien wspólny interfejs(metoda `pole()`) dla wszystkich klas pochodnych, natomiast nie powinno dojść do sytuacji, że tworzony jest obiekt typu `Figura`. Jest to bez sensu z punktu widzenia matematyki: nie ma takiej konkretnej figury jak Figura: to tylko określenie na pewną rodzinę obiektów posiadających pewne wspólne cechy.

Analogicznie: metoda abstrakcyjna jest to metoda, która dostarcza interfejs ale sama nie powinna zostać wywołana (w wielu językach metoda oznaczona jako abstrakcyjna nie ma swojego ciała i musi zostać koniecznie zaimplementowana w klasach pochodnych). Dla powyższego przykładu, metodą abstrakcyjną jest metoda pole().

Niemniej jednak nie każda metoda w klasie abstrakcyjnej jest metodą abstrakcyjną – to od nas zależy czy implementacja metody ma sens w klasie bazowej czy nie (tak jak w przykładzie ze szkołą, gdzie metoda w klasie abstrakcyjnej posiada swoją implementację).

UWAGA:

Sam Python nie posiada wbudowanych mechanizmów pozwalających bezpośrednio oznaczyć klasę czy metodę jako abstrakcyjną. Dawniej przyjęło się natomiast, że metody abstrakcyjne w swojej implementacji po prostu rzucają wyjątek. Np.:

```
raise NotImplementedError("Please Implement this method")
```

Dla zainteresowanych: dostępny jest moduł abc, który umożliwia korzystanie z adnotacji pozwalających oznaczać klasy i metody jako abstrakcyjne: https://docs.python.org/3/library/abc.html.