



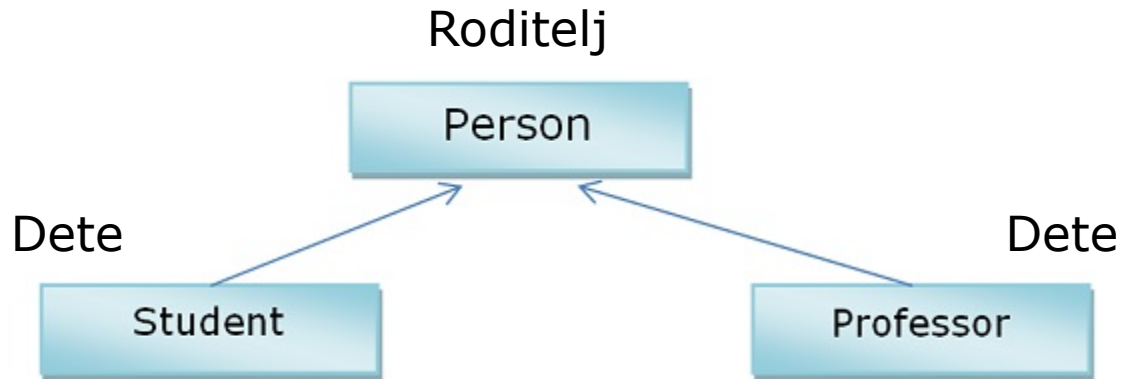
Distance Learning System

Nasleđivanje i polimorfizam

Object Oriented programming in Python

Nasleđivanje

- Nasleđivanje predstavlja osobinu klase da prihvati osobine neke druge klase, pri čemu klasa, koja biva nasleđena, ima status roditeljske, dok klasa koja nasleđuje ima status deteta klase



Nasleđivanje

- Prilikom nasleđivanja, sa roditeljske klase na dete klasu se prenose "samo" **public** i **protected** članovi

```
class Car:  
    def __init__(self):  
        self.model = ""  
        self.make = ""  
        self._wheels = 0  
        self.__doors = 0
```

Nasleđuju se

Ne može se naslediti

Nasleđivanje - primer (oopp-ex02 inheritance)

- Za dalje praćenje primera treba kreirati klasu Person:

```
class Person:
    def __init__(self,firstname,lastname):
        self.firstname = firstname
        self.lastname = lastname
    def show(self):
        print(f"Person: {self.firstname} {self.lastname}")
```

- Analizirajte ovu klasu

Nasleđivanje




- Prethodno kreiranu klasu je moguće naslediti. Na primer:

```
1
class Student(Person):
    def __init__(self, firstname, lastname, indexnumber):
        2 super().__init__(firstname, lastname)
        3 self.indexnumber = indexnumber
```

- Klasa ima zanimljive nove (i stare) komponente:
 1. **Nakon naziva klase, stoje parametri** (klase koje se nasleđuju)
 2. **super()** (ukazuje na roditelja)
 3. **self** (ukazuje na aktuelni objekat)

Nasleđivanje (instanciranje nasleđene klase)

- Klasu Student možemo koristiti, čak i bez znanja o postojanju klase Person

```
st = Student("Lucky", "Luke", "123/45")
st.  
   firstname  
   indexnumber  
   lastname
```

- Okruženje (Visual Studio Code) je detektovalo polja bazne i izvedene klase

Vežba 1 Dictionary (oopp-ex02 dictionary.py)

- Kreirati klasu Dictionary koja ima metod get_word
- Metod prihvata string parametar (tražena reč)
- Metod vraća kao rezultat prevod tražene reči
- Napraviti podklase klase Dictionary, za dva jezika po izboru

```
Enter word: house
English : house House
Deutsch : house Haus
Enter word: car
English : car Car
Deutsch : car Auto
Enter word: █
```

Prepisivanje metoda

- Prepisivanje je pojava u kojoj metoda roditeljske klase biva prepisana od strane dete klase
- Na primer, kada bi u klasi Student, dodali sledeći kod, mogli bi da kažemo da je došlo do prepisivanja

Dodatak klasi Student

```
def show(self):  
    print(f"Student: {self.firstname} {self.lastname} {self.indexnumber}")
```

Metod već postoji u klasi Person i biće prepisan

```
def show(self):  
    print(f"Person: {self.firstname} {self.lastname}")
```


Prepisivanje metoda

- Iako smo prepisali roditeljski metod, to ne znači da je on bespovratno izgubljen, odnosno, da je stvarno prepisan
- Ukoliko nam ustreba, možemo doći do njega pomoću funkcije `super`:

```
def show(self):  
    print(f"Student: {self.firstname} {self.lastname} {self.indexnumber}")  
    super().show()
```

- Rezultat koda je, nakon startovanja sledeći:

```
Student: Lucky Luke 123/45  
Person: Lucky Luke
```

- Očigledno je da su aktivirani i prvi i drugi metod

Magični članovi

<https://rszalski.github.io/magicmethods/>

- Magični članovi su članovi čijim prepisivanjem ili dodavanjem možemo modifikovati ponašanje objekta
- Magični članovi se imenuju sa dve donje crte ispred i nakon identifikatora. Na primer: `__init__` kojim se “presreće” trenutak nakon kreiranja objekta
- Osim metode `__init__`, ima još nekih članova koje često prepisujemo

Prepisivanje `__str__` metode

- Jedan od čestih slučajeva prepisivanja jeste prepisivanje magične metode `__str__`.
- Ako pokušamo da prikazemo objekat klase Student na izlazu, dobićemo rezultat poput sledećeg

```
st = Student("Lucky", "Luke", "123/45")  
print(st)
```



```
<__main__.Student object at 0x10ecd67b8>
```

- To je zato što Python automatski poziva roditeljski `__str__` metod (i to ne od Klase Person, već od prve roditeljske klase koja ima implementaciju ovog metoda, a to je klasa Object).
- Ako prepíšemo metod `__str__` u klasi Student:

```
def __str__(self):  
    return f"Hello from {self.firstname}"
```

- Na izlazu ćemo kao rezultat dobiti:

```
Hello from Lucky
```

Prepisivanje operatora

- Python podržava prepisivanje operatora nad objektima
- Ovo je korisna opcija, jer kompleksni tipovi ne moraju i ne mogu uvek isto reagovati na aritmetičke operacije ili operacije poređenja
- Na primer, znamo da je $2 + 3 = 5$, ali šta ako, na primer postoje dve instance klase `Point`, i obe imaju polja `x` i `y`? Koliko bi tada bilo: `pt1 + pt2`?

```
class Point:  
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
pt1 = Point(2,3)  
pt2 = Point(3,4)  
print(pt1+pt2)
```

```
TypeError: unsupported operand type(s)  
for +: 'Point' and 'Point'
```

- Prepisivanje operatora nam omogućava da korigujemo ovo ponašanje prema svojim potrebama

Prepisivanje operatora

- Python prilikom aktivacije operatora ne poziva istoimenu metodu, već svaki operator ima odgovarajuću magičnu metodu koju treba prepisati. Na primer, za operator +, metoda je `__add__`, za ==, metoda je `__eq__`

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x+other.x,self.y+other.y)

pt1 = Point(2,3)
pt2 = Point(3,4)
pt3 = pt1 + pt2
print(pt3.x, pt3.y)
```

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

pt1 = Point(2,3)
pt2 = Point(2,3)
print(pt1 == pt2)
```

Metode operatora

Aritmetički operatori

+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)

Operatori poređenja

<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Skraćeni operatori dodele

-=	__isub__(self, other)
+=	__iadd__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)

Unarni operatori

-	__neg__(self, other)
+	__pos__(self, other)
~	__invert__(self, other)

Polimorfizam

- Polimorfizam u Objektno Orjentisanom programiranju znači da će se istoimeni metod u jednoj klasi različito ponašati od istoimenog metoda u drugoj klasi, pri čemu, obe klase imaju istog roditelja
- Na primer, ako bi rekli da su pas i ptica podklase klase životinja, te da oboje imaju metod: kreći se, ova dva metoda, iako se isto zovu, imala bi potpuno drugačiju realizaciju, jer ptica leti, a pas hoda
- Da bi vršili polimorfovanje klase, moramo poznavati pojmove **prepisivanja** i **preopterećenja** (override i overload).

Polimorfovanje

- Da bi došlo do polimorfizma, mora postojati bar dve dete klase
- Druga dete klasa može biti klasa **Profesor**

```
class Professor(Person):  
    def __init__(self,firstname,lastname,subject):  
        super().__init__(firstname,lastname)  
        self.subject = subject  
    def show(self):  
        print(f"Professor: {self.firstname} {self.lastname} {self.subject}")
```

- Analizirajte metode show u sve tri klase

Polimorfovanje

- Ako sada instanciramo sve tri klase, a zatim aktiviramo njihove metode show, dobićemo tri različita rezultata na izlazu:

```
person      = Person("John","Davidson")
student     = Student("John","Smith","10/2014")
professor   = Professor("Edward","Owen","Python Programming")
person.show()
student.show()
professor.show()
```

```
Person: John Davidson
Student: John Smith 10/2014
Professor: Edward Owen Python Programming
```

- Rezultat je različit za istoimene nasleđene metode, te možemo reći da je došlo do **polimorfovanja**

Polimorfovanje i nasleđivanje u praksi?

- Polimorfovanje je prirodna posledica nasleđivanja a u realnom sistemu, pojavljuje se veoma često:



Polimorfovanje i nasleđivanje u praksi?

```
class SceneObject:  
    position = []  
    texture = ""  
    model = ""
```

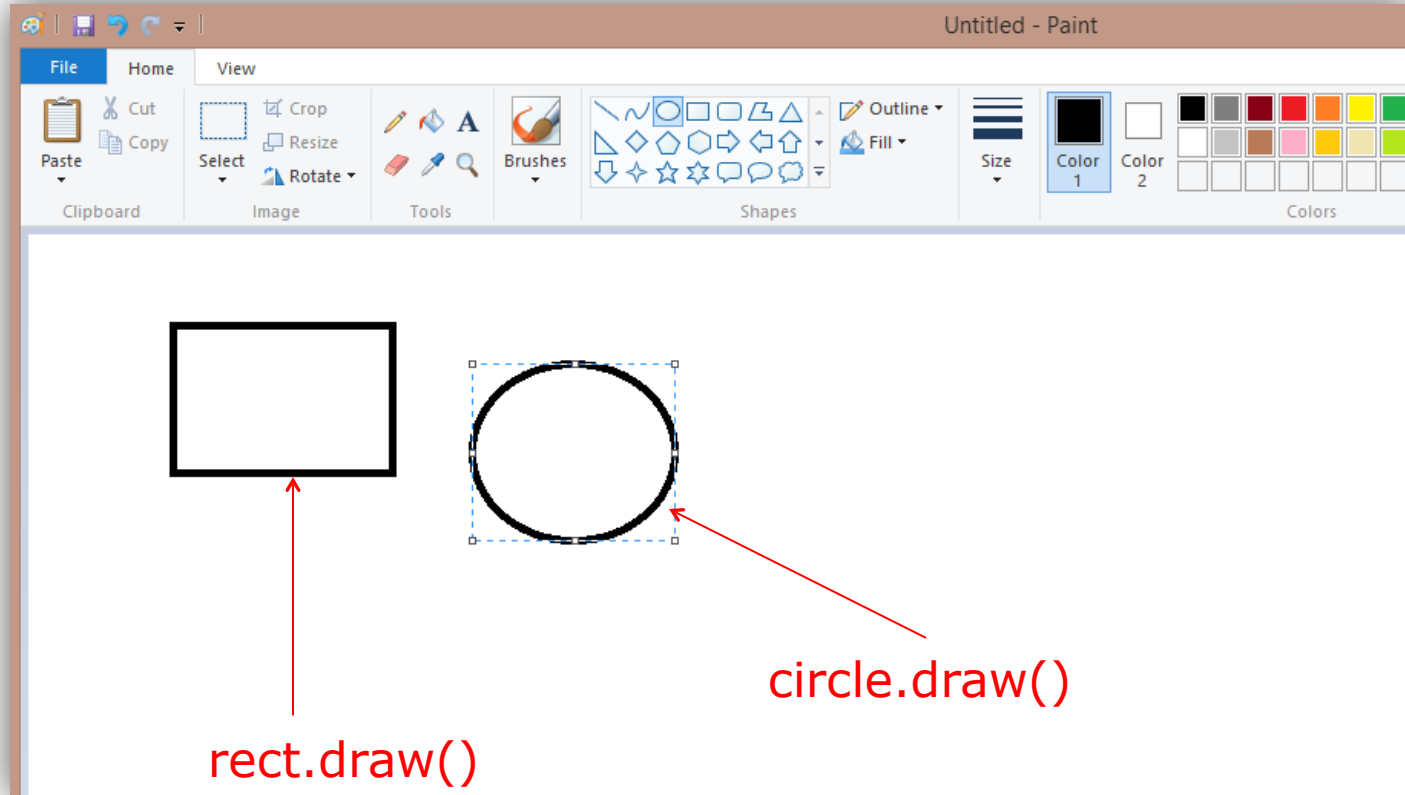
```
class Soldier(SceneObject):  
    ...
```

```
class Block(SceneObject):  
    ...
```

```
class Gun(SceneObject):  
    ...
```



Polimorfovanje i nasleđivanje u praksi?



Vežba 2 CreditCard (oopp-ex02 creditcard.py)

- Kreirati klasu Card koja ima polja: broj, stanje i metod pay
- Metod pay skida određenu sumu sa stanja kartice
- Kreirati klase Visa i Master, koje nasleđuju metod card, ali prilikom naplate, naplaćuju i porez koji za Viza karticu iznosi 5 posto, a za Master karticu 8 posto

Preopterećenje (overload) metoda

- Preopterećenje funkcija i metoda je u Python-u moguće pomoću podrazumevanih parametara:

```
def show(self, firstname = None, lastname = None, indexnumber = None):  
    if firstname:  
        print(f"Student: {self.firstname}")  
        if lastname:  
            print(f"Student: {self.firstname} {self.lastname}")  
            if indexnumber:  
                print(f"Student: {self.firstname} {self.lastname} {self.indexnumber}")  
    else:  
        super().show()
```

- Jedini način da specificiramo na koji ćemo metod pozvati, jeste da prosledimo broj parametara koji mu odgovara

```
student.show("John")  
student.show("John","Smith")  
student.show("John","Smith","10/2014")
```

Vežba 3 Shapes (oopp-ex02 shapes.py)

- Potrebno je napraviti jednu klasu Shape koja bi sadržala neke podatke o geometrijskom obliku:
 - poziciju (x,y)
 - boju i naziv.
- Potrebno je napraviti tri klase koje nasleđuju ovu klasu, jednu za krug, jednu za kvadrat i jednu za pravougaonik.
- Svaka treba da sadrži metodu za izračunavanje površine, kao i sopstvene attribute koji su neophodni za ovo izračunavanje (strane za pravougaonik i kvadrat i poluprečnik za krug).
- Krug, takođe, treba da poseduje i statičko polje PI.

```
Circle area: 12.56  
Rectangle area: 6  
Square area: 9
```


Apstraktne klase

- Apstraktne klase su šabloni za kreiranje drugih klasa
- Njihova osnovna osobina je da se ne mogu instancirati, već jedino naslediti ili koristiti u statičkom kontekstu
- Python u osnovi ne poznaje apstraktne klase, ali se one mogu simulirati paketom abc (**abstract base classes**)
- Da bi klasa postala apstraktna, nasleđuje klasu **ABC** iz paketa **abc**
- Dekorator **abstractmethod** iz istog paketa, označava dekorisani metod apstraktnim

```
import abc
class MyClass(abc.ABC):
    @abc.abstractmethod
    def f(self):
        pass
```


Instanciranje apstraktnih klasa

- Apstraktne klase NIJE moguće instancirati

Sledeći kod funkcioniše

```
import abc
class Person:
    def get_name(self):
        pass
```

```
person = Person()
```

Ali sledeći ne

```
import abc
class Person(abc.ABC):
    @abc.abstractmethod
    def get_name(self):
        pass
```

```
person = Person()
```

```
TypeError: Can't instantiate abstract class
Person with abstract methods get_name
```

Nasleđivanje osobina od apstraktnih klasa

- Apstraktna klasa može imati sve funkcionalnosti kao i konkretna klasa, uključujući **metode** i **polja**
- Svaka klasa koja nasledi apstraktnu klasu i implementira njene apstraktne metode može se normalno upotrebljavati, i imaće na raspolaganju sve elemente klase koju je nasledila

```
class Student(Person):  
    def get_name(self):  
        pass
```

```
import abc  
class Person(abc.ABC):  
    def __init__(self, first_name, last_name):  
        self.first_name = first_name  
        self.last_name = last_name  
    @abc.abstractmethod  
    def get_name(self):  
        pass  
    def show(self):  
        print(f"{self.first_name} {self.last_name}")
```

- I naravno, može se instancirati

```
person = Student("Peter", "Jackson")  
person.show()
```

Nasleđivanje apstraktne klase apstraktnom klasom

- Klasa koja nasleđuje apstraktnu klasu, takođe može biti apstraktna, i tada za nju takođe važe pravila apstraktne klase

```
class Student(Person):  
    pass
```

```
person = Student("Peter", "Jackson")  
person.show()
```

```
TypeError: Can't instantiate abstract class Student  
with abstract methods get_name
```

Apstraktne metode

- Apstraktna klasa sadrži apstraktne metode
- Ovo su metode koje nemaju telo, već sadrže samo potpis:

```
import abc
class Person(abc.ABC):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    @abc.abstractmethod
    def show(self):
        pass
```

Apstraktan metod **show**

- Čemu služi metod koji ne može da uradi ništa, i koji u stvari i ne postoji?

Apstraktne metode

- Apstraktan metod predstavlja neku vrstu ugovora između apstraktne klase i klase koja je nasleđuje
- On garantuje da će klasa koja nasleđuje ispoštovati uslove za učešće u ostatku sistema

Na primer:

Svaka benzinska pumpa ima isti sistem za ubacivanje goriva u automobil. Prilikom izgradnje pumpe, kreator pumpe mora ispoštovati standarde. Krajnji cilj je taj, da kada se pumpa jednom nađe u sistemu, vozači ne razmišljaju o tome da li će crevo odgovarati ulazu rezervoara



Primer apstraktne metode (oopp-ex02 gasstation.py)

- Recimo da postoji šablon za kreiranje benzinskih stanica, i da on zahteva metod **fillCar**
- Ako bismo nasledili ovu klasu sopstvenom praznom klasom (za sopstvenu stanicu), prilikom instanciranja bi se pojavio problem, jer nismo ispoštovali ugovor/dogovor i nismo implementirali apstraktan metod fillCar

```
import abc
class BaseStation(abc.ABC):
    @abc.abstractmethod
    def fillCar(self): pass
```

```
class ShellGasStation(BaseStation):
    pass
```

```
shell = ShellGasStation()
```



Primer apstraktne metode (oopp-ex02 gasstation.py)

- Kada implementiramo metod fillCar, insanciranje više nije problem

```
class ShellGasStation(BaseStation):  
    def fillCar(self):  
        print("Filling car with water")
```

- Primećujemo da je sve u redu, iako će automobil zapravo biti napunjen vodom (što svakako nije u redu)
- Ovaj deo sistema nije „zainteresovan“ za taj problem. Njemu je jedino važno da je ispoštovan ugovor, te da je metod obezbeđen.

Primer apstraktne metode (jcex082014 RpgHeroes)

- Recimo da kreiramo rpg igru i da u igru treba periodično dodavati nove heroje
- Koncept igre je takav, da svaki heroj ima određeni set akcija, koje se aktiviraju određenim metodama: **firePrimary** i **fireSecondary**
- Takođe, heroji trebaju da imaju mogućnost da prime određeni udarac i ta da na osnovu toga izvrše neku reakciju (izračunaju štetu, prekinu da postoje, izvrše automatski kontra udarac...). Ovo bi mogao biti metod: **receiveHit**
- Takođe, želimo da postoje situacije u kojoj svi heroji reaguju primanjem udarca. Na primer, prilikom neprijateljskog area of effect napada.



Primer apstraktne metode (oopp-ex02 rpgheroes)

Prilikom postavke sistema, svakako ćemo znati da će svaki heroj moći da ima neke opšte karakteristike. Na primer, ima određenu količinu zdravlja i mane, može biti mrtav ili živ. Ali ne možemo znati na koji način će se do tih podataka doći, jer to zavisi od nekih karakteristika koje su specifične za heroja. Dakle, osnovna klasa kojom bi mogli predstaviti heroja, mogla bi u osnovi biti:

```
from abc import ABC, abstractmethod
class BaseHero(ABC):
    health = 0
    mana = 0
    dead = False
    def areaOfEffectHit(self):
        pass
```

Kreirana klasa je u stanju da nosi podatke koji su nam potrebni u igri, ali imamo problem sa metodom **areaOfEffectHit**. Znamo da je to situacija kada heroji koji su u zoni napada, trpe štetu. Ali šta ta šteta znači za te heroje? Koliko će im zdravlja oduzeti? I koje će druge posledice doneti? Ne znamo, jer još uvek nemamo ni jednog heroja, niti znamo kako će se on ponašati u datoj situaciji. Takođe, čak i da imamo heroje, svi oni će imati različito ponašanje.

Primer apstraktne metode (oopp-ex02 rpgheroes)

- Da bi rešili problem opisan na prethodnom slajdu, možemo u metodi `areaOfEffectHit`, pozvati metod **`receiveHit`** i kompletnu odgovornost prebaciti na autora klase **konkretnog** heroja.
- Nije nas briga da li će autor odlučiti da heroj preživi ovakav udarac, to je više stvar pravila igre, nego sistema
- Ono što nas jeste briga to je da ne dođemo u situaciju da metod `receiveHit` uopšte ne postoji, i zbog toga ga označavamo apstraktno, kako bi naterali autora heroja da ga implementira.
- *Da li smo mogli i sami da izvršimo neku generičku implementaciju ovih metoda (`receiveHit`, `primaryFire` i `secondaryFire`), a autoru heroja ponudimo da ih prepiše samo ukoliko želi? Jesmo, i to bi takođe bilo sasvim ispravno rešenje ovog problema.*

```
class BaseHero(ABC):  
    health = 0  
    mana = 0  
    dead = False  
    def areaOfEffectHit(self):  
        self.receiveHit()  
    @abstractmethod  
    def receiveHit(self): pass  
    @abstractmethod  
    def primaryFire(self): pass  
    @abstractmethod  
    def secondaryFire(self): pass
```

Primer apstraktne metode (oopp-ex02 rpgheroes)

- Sada autor heroja ima slobodu prilikom kreiranja, a mi (kao eventualni autori sistema) znamo da će se taj deo uklopiti u ostatak. U suprotnom, program neće moći da bude ni preveden, zbog čega će biti lako identifikovati eventualni problem
- Jedino što moramo da uradimo, jeste da kažemo autoru da će heroj morati da nasledi klasu **BaseHero**. Bez ovoga, naravno, kompletan koncept ne bi imao smisla
- Možemo takođe autoru napraviti i „sheet“ za heroja. Na primer:

Tip heroja DwarfWarrior

Pri udarcu
gubi 10 zdravlja

Primarno oružje
troši 5 mane

Sekundarno oružje
troši 10 mane

```
class DwarfWarrior(BaseHero):
    def receiveHit(self):
        self.health -= 10
        self.dead = this.health <= 0
    def primaryFire(self):
        self.mana -= 5
        print("Firing primary!!!")
    def secondaryFire(self):
        self.mana -= 10
        print("Firing secondary!!!")
    def show(self):
        print(f"Health: {self.health} Mana: {self.mana} Dead: {self.dead}")
```

Primer apstraktne metode (oopp-ex02 rpgheroes)

- Nakon kreirane klase, posao autora klase je završen. Pokušajte da napišete i startujete sledeći kod
- Samo su prve četiri linije bitne, ostatak može biti proizvoljan i njime samo simuliramo dinamiku igre

```
import dwarf

dwarfWarrior = dwarf.DwarfWarrior()
dwarfWarrior.dead = False
dwarfWarrior.health = 25
dwarfWarrior.mana = 25
dwarfWarrior.show()
dwarfWarrior.receiveHit()
dwarfWarrior.show()
dwarfWarrior.receiveHit()
dwarfWarrior.show()
dwarfWarrior.primaryFire()
dwarfWarrior.show()
dwarfWarrior.secondaryFire()
dwarfWarrior.show()
dwarfWarrior.receiveHit()
dwarfWarrior.show()
```

Ne mora da postoji.
Simulira tok igre

Zadatak:

- Pokušajte samostalno da kreirate heroja po sledećem sheet-u:
 - 1. Tip heroja ElfMage***
 - 2. Pri udarcu gubi 30 zdravlja***
 - 3. Primarno oružje troši 15 mane***
 - 4. Sekundarno oružje troši 50 mane***
- Pokušajte da kreirate konstruktor za ovog heroja, tako da se osnovne karakteristike mogu dodati prilikom instanciranja

Višestruko nasleđivanje

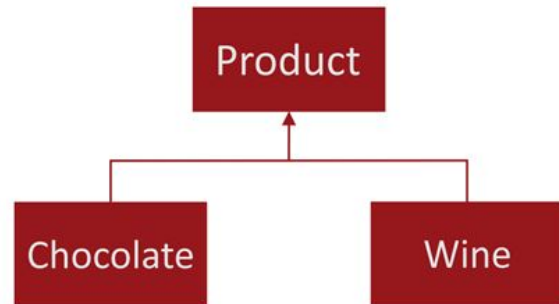
- Python, za razliku od mnogih drugih objektnih jezika, ima mogućnost višestrukog nasleđivanja
- Na ovaj način, funkcionalnosti više klasa, možemo spojiti u jednu
- Sličan efekat, postiže se na različite načine na nekim drugim platformama (Threat, Extension, Interface)

```
class A:
    def f1(self):
        print("Hello from A")
class B:
    def f2(self):
        print("Hello from B")
class C(A,B):
    pass

c = C()
c.f1()
c.f2()
```

Zadatak

Zamislite da je potrebno da modelujete informacijski sistem jedne trgovine. Potrebno je da napravite klasu **Product** koja će predstavljati osnovu za dalje nasleđivanje i neće se moći instancirati. Ovu klasu nasleđuju dve klase koje predstavljaju konkretne grupe proizvoda: **Chocolate** i **Wine**. Ova hijerarhija prikazana grafikom izgleda ovako:



Svaki proizvod poseduje određene osobine:
naziv proizvoda,
bar-kod (obična numerička vrednost),
osnovnu cenu,
porez.

Takođe, svaki proizvod poseduje i metodu za računanje cene, koja se izračunava kada se osnovna cena i uveća za iznos poreza. Porez (PDV) za svaki proizvod je 20% i ovo je podatak koji je konstantan i neće se menjati. Ipak, proizvodi iz grupe vina, imaju i dodatni porez, pošto spadaju u grupu alkoholnih pića i on iznosi 10% od cene već uvećane za iznos PDV-a. I ovo je podatak koji je konstantan i neće se menjati. Zbog ovoga je potrebno redefinisati metodu za računanje cene u okviru klase *Wine*. Pored ovoga, klasa *Wine* treba da poseduje atribut koji definiše zapreminu boce, a klasa *Chocolate* atribut koji definiše težinu. U klasama *Chocolate* i *Wine*, potrebno je napraviti parametrizovane konstruktore za kreiranje objekata. Potrebno je, takođe, u klasama redefinisati metodu `__str__` za prikaz informacija o objektu.