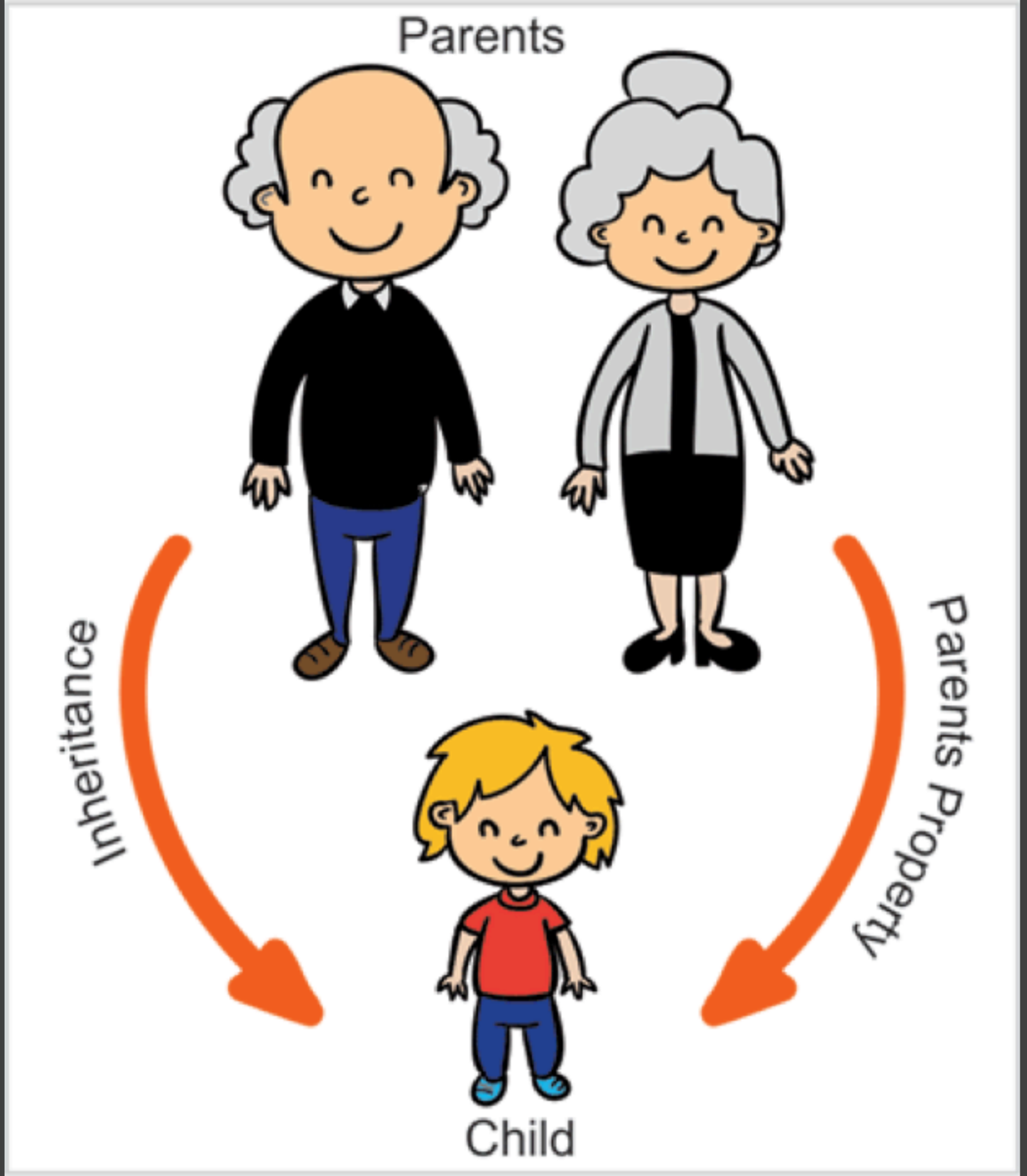
Objektinis Programavimas

Paveldėjimas (*Inheritance*)



Turinys

Motyvacija (1)

Kelios kodo eilutės, kelios sugaištos minutės ir 4-a praktinė užduotis "*atlikta*"! 😄

```
#include <iostream>
#include <vector>

template<typename T>
class Vector : public std::vector<T> {
public:
    using std::vector<T>::vector; // naudoti c-tor'ius iš std::vector
};

int main() {
    Vector<int> v(10,1);
    std::cout << "v[9] = " << v[9] << std::endl;
    v.push_back(2);
    std::cout << "v[10] = " << v[10] << std::endl;
    std::cout << "v[11] = " << v[11] << std::endl; // Ką gausime čia?
}
```

Motyvacija (2)

Norint prisitaikyti egzistuojančių klasių funkcionalumą savo poreikiams

```
#include <iostream>
#include <vector>

template<typename T>
class Vector : public std::vector<T> {
public:
    using std::vector<T>::vector;                          // naudoti c-tor'ius iš std::vector
    T& operator[](int i) { return std::vector<T>::at(i); } // patikrina range
    const T& operator[](int i) const { return std::vector<T>::at(i); } // patikrina range const objektams
};

int main() {
    Vector<int> v(10,1);
    std::cout << "v[9] = " << v[9] << std::endl;
    v.push_back(2);
    std::cout << "v[10] = " << v[10] << std::endl;
    std::cout << "v[11] = " << v[11] << std::endl; // O ką dabar gausime?
}
```

Motyvacija (3)

Norint išplėsti egzistuojančių klasių funkcionalumą

```
/* Vector kodas iš ankstesnės skaidrės */
// Vector'ių sudėtis
template<typename T>
Vector<T> operator+(const Vector<T>& a, const Vector<T>& b) {
  if (a.size() != b.size())
    throw std::runtime_error("Vektorių dydžio neatitikimas!");
  auto size = a.size();
  Vector<T> c(size);
  for (size_t i = 0; i != a.size(); ++i)
    c[i] = a[i] + b[i];
  return c;
}

int main() {
    Vector<int> v1(10,1);
    Vector<int> v2(10,2);
    Vector<int> v3 = v1 + v2; // sudėdame Vector'ius
}
```

Paveldėjimas (*inheritance*) (1)

— Objektiškai orientuotame programavime (OOP) **paveldėjimas** (*inheritance*) yra vienas svarbiausių ir naudingiausių principų/mechanizmų.

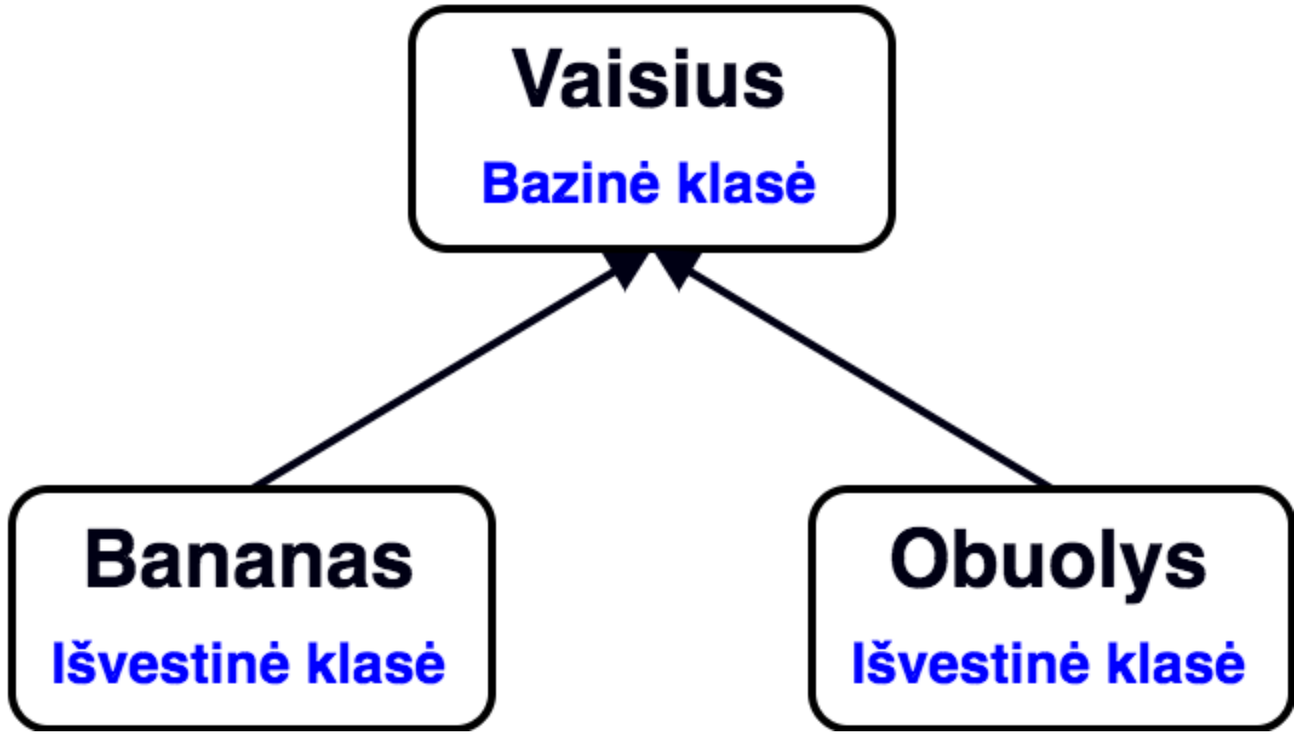
— C++ kalboje klasės gali paveldėti kitos klasės ar net kelių klasių duomenis (*member variables*) ir metodus (*member functions*).

— Tokiu būdu galime kurti naujas klases išplečiant egzistuojančių funkcionalumą.

Paveldėjimas (*inheritance*) (2)

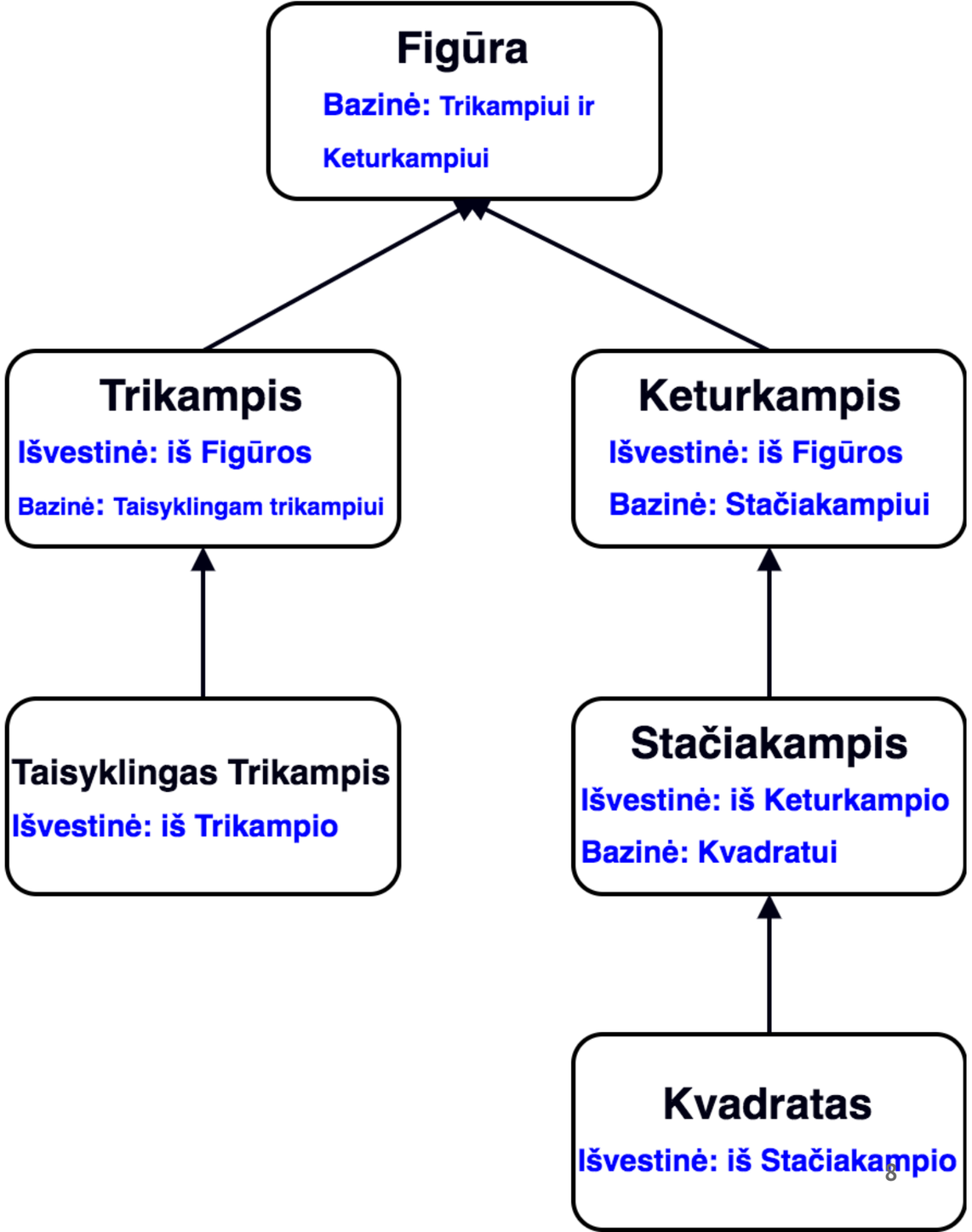
— Klasė, iš kurios paveldimos visos savybes vadinama **bazinė klasė** (*base class*) ar (*parent class*), o ją papildanti - **išvestinė klasė** (*derived class*) ar (*child class*).

— Jei bazinėje klasėje ištaisomos klaidos ar realizuojamos naujos funkcijos, visa tai automatiškai paveldi išvestinės klasės!

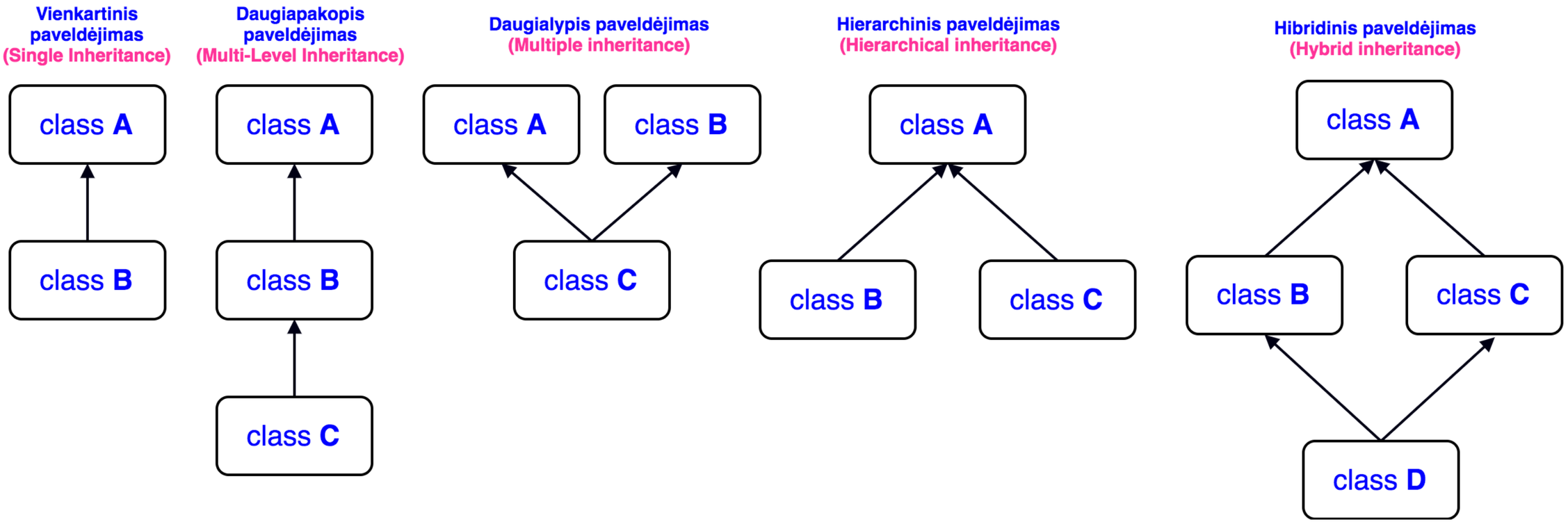


Paveldėjimas (*inheritance*) (3)

— Tam tikros klasės tuo pat metu gali būti išvestinės (iš ankstesnės bazinės klasės) ir bazinės klasės iš jų naujai išvedamoms klasėms.



Paveldėjimo formos



Bazinės (base) klasės kūrimas

```cpp
#include<iostream>
#include<string>

class Base {
public:
    std::string vardas;
    Base(std::string v = "") : vardas{v} { }
    std::string getVardas() const { return vardas; }
};

int main() {
    Base b{"Remigijus"}; // C++ išskiria atmintį ir iškviečia konstruktorių ją įnicializuoti.
    std::cout << b.getVardas() << std::endl;
    return 0;
}
```

— `Base` yra tradicinė **bazinė** klasė.

Išvestinės (derived) klasės kūrimas (1)

```
class DerivedClass : accessSpecifier BaseClass { ... };
```

— Prieigos specifikatorius (`accessSpecifier`) gali būti: **public**, **protected** ir **private** (numatytasis).

— Specifikatorius nustato išvestinės klasės paveldimumo lygį (*apie tai netrukus*).

— Bazinė klasė neturi priėjimo prie išvestinės klasės duomenų ir funkcijų!

Išvestinės (derived) klasės kūrimas (2)

```
#include<iostream>
#include<string>
#include "Base.h" // Base klasės realizacija

class Derived : public Base {
public:
    int amzius;
    Derived(int a = 0) : amzius{a} {  }
    int getAmzius() const { return amzius; }
};

int main() {
    Base b{"Remigijus"};
    std::cout << b.getVardas() << std::endl;
    Derived d{36};
    std::cout << d.getAmzius() << std::endl;
    std::cout << d.getVardas() << std::endl; // Ką gausime čia?
    return 0;
}
```

Išvestinės (derived) klasės kūrimas (3)

— Išvestinė klasės objektų konstravimas vyksta dvejomis pakopomis: pirmiausia sukonstruojama **bazinė** dalis (aukščiausiai esanti paveldimumo medyje), o tuomet **išvestinė(s)** dalis(-ys), iki žemiausiai paveldimumo medyje esančio palikuonio.

— Taigi, kai mes sukuriame išvestinį objektą, pirmiausia iškviečiamas bazinis (numatytasis) konstruktorius ir tik po to išvestinis konstruktorius.

Išvestinės (derived) klasės kūrimas (4)

— Papildžius **Base** ir **Derived** konstruktorius:

```
// Konstruktoriai
Base(std::string v = "") : vardas{v} { std::cout << "Base c-tor\n"; }
Derived(int a = 0) : amzius{a} { std::cout << "Derived c-tor\n"; }

int main() {
    Derived d{36};
    std::cout << d.getAmzius() << std::endl;
    return 0;
}

/* Gauname:
 Base c-tor
 Derived c-tor
 36
```

Išvestinės (derived) klasės kūrimas (5)

```
// Includinam viską ko reikia

int main() {
    Base b{"Remigijus"};
    std::cout << b.getVardas() << std::endl;
    Derived d{36};
    std::cout << d.getAmzius() << std::endl;
    std::cout << d.getVardas() << std::endl; // Ką čia gausime?
    return 0;
}
```

— Kaip inicializuoti **d.vardas** reikšmę?

— Ok, ignoruojame, kad `vardas` yra **public** 😄 Tuoj jis bus `private`, kaip ir turėtų būti 😉

Išvestinės (derived) klasės kūrimas (6)

```
// Pirmas bandymas: per member-initializer list'a
Derived(int a = 0, std::string v = "") : amzius(a), vardas{v} { }

int main() {
    Derived d{36, "Remis"}; // Ar viskas gerai?
    return 0;
}
```

— Tik nepaveldėti kintamieji gali būti (member) inicializuoti išvestinio konstruktoriaus! Kodėl?

— Kas jei **Base** klasės kintamieji yra `const` ar & tipo?

Išvestinės (derived) klasės kūrimas (7)

```
// Antras bandymas: konstruktoriaus viduje
Derived(int a = 0, std::string v = "") : amzius(a) { vardas = v; }

int main() {
    Derived d{36, "Remis"}; // Ar viskas gerai?
    return 0;
}
```

— Paveldėtiems kintamiesiems galima pakeisti jų reikšmes konstruktoriaus viduje!

Išvestinės (derived) klasės kūrimas (8)

- Tačiau tokiu būdu `vardas` reikšmė yra priskiriama du kartus: 1-ą kartą per **Base** initializer-list'ą, 2-ą kartą **Derived** konstruktoriaus viduje.
- Be to, šios reikšmės nemato **Base** konstruktorius.
- Galiausiai kaip ir prieš tai atveju, kas jei **Base** klasės kintamieji yra `const` ar & tipo?
- Kas būtų, jei `Base::vardas` būtų (kaip ir turėtų būti) `private`?

Išvestinės (derived) klasės kūrimas (9)

```
// Trečias bandymas: inicializuojame per reikiamą Base c-tor'ių,
// nes konstruktoriai nepaveldimi
Derived(int a = 0, std::string v = "") : Base{v}, amzius(a) { }

int main() {
    Derived d{36, "Remis"}; // Ar viskas gerai?
}
```

— Kaip dabar su `const` ir & tipo kintamaisiais?

— Kaip dėl `std::string vardas` ir `int amzius` tapimo `private` tipo kintamaisiais?

Nepaveldimi bazinės klasės metodai

— Konstruktorius

```
Derived d{36, "Remis"};
d.Base("Remigijus"); // Kažkas tokio tikrai neveiks :)
```

— Kopijavimo konstruktorius (*copy constructor*)

— Priskyrimo operatorius (`operator=`)

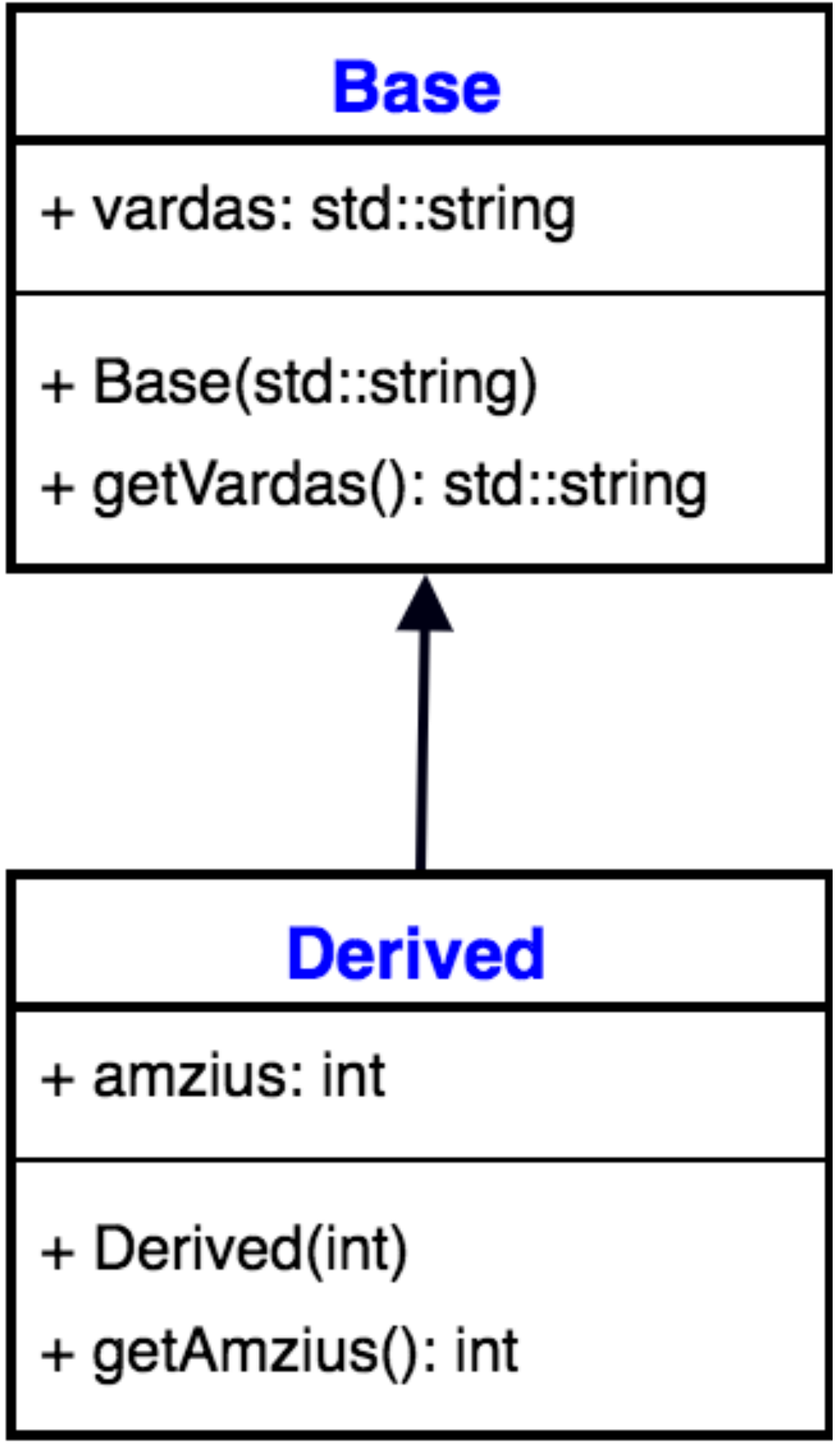
— Destruktorius

— Draugiškos klasės

UML diagramos

Klasių struktūrą ir paveldimumo schemas patogu pateikti UML (*Unified Modeling Language*) diagrama.

- Nurodomi klasių pavadinimai, duomenys, konstruktoriai, destruktoriai ir metodai.
- Paveldimumas žymimas rodykle link bazinės klasės.
- Išvestinės klasės pateikiama tik pavadinimas ir papildomi laukai.



Prieigos specifikatoriai (1)

— Iki šiol pavyzdžiuose naudojome **public** paveldėjimą.

— Mes jau esame susipažinę su **private** ir **public**:

```
class Base {
private:
    std::string vardas; // pasiekia Base nariai, friend'ai, bet ne išvestinės klasės
public:
    std::string pavarde; // pasiekia be kas
};
```

— Paveldėjime naudojama ir **protected**, kurios narius pasiekia klasės nariai, draugai ir išvestinės klasės.

Prieigos specifikatoriai (2)

```
class Base {
private:
    std::string vardas; // pasiekia Base nariai, friend'ai, bet ne išvestinės klasės
public:
    std::string pavarde; // pasiekia be kas
protected:
    std::string pravarde; // pasiekia Base nariai, friend'ai ir išvestinės klasės
};

class Derived: public Base {
public:
    Derived() {
        vardas   = "Remigijus";   // negalima pasiekti private iš išvestinės klasės
        pavarde  = "Paulavičius"; // galima pasiekti public iš išvestinės klasės
        pravarde = "Hmm?";        // galima pasiekti protected iš išvestinės klasės
    }
};

int main() {
    Base b;
    b.vardas   = "Neremigijus";   // Ar galima?
    b.pavarde  = "Nepaulavičius"; // Ar galima?
    b.pravarde = "Nehmm";         // Ar galima?
}
```

Pasiekiamumas priklausomai nuo prieigos specifikatoriaus

Klasės viduje (realizacijoje) ir kuriant klasės objektus ("išorėje") pasiekiamumai skiriasi:

Prieigos specifikatorius bazinėje klasėje	Pasiekiamumas bazinėje klasėje	Pasiekiamumas išvestinėje klasėje	Bazinės klasės objekto pasiekiamumas
`Public`	👍	👍	👍
`Protected`	👍	👍	👎
`Private`	👍	👎	👎

Prieigos specifikatoriai ir paveldėjimo tipai

Išvestinės klasės prieigos specifikatoriai (pasiekiamumas) priklausomai nuo **paveldėjimo tipo**:

Prieigos specifikatorius bazinėje klasėje	**Public paveldėjimas**	**Protected paveldėjimas**	**Private paveldėjimas**
`Public`	`Public`	`Protected`	`Private`
`Protected`	`Protected`	`Protected`	`Private`
`Private`	Nepasiekiama	Nepasiekiama	Nepasiekiama

Paveldėtų funkcijų naudojimas (1)

```
#include<iostream>
#include<string>

class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    std::string getVardas() const { return vardas; }
    void whoAmI() { std::cout << "Aš esu Base klasė\n"; }
};
class Derived : public Base {
protected:
    int amzius;
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    int getAmzius() const { return amzius; }
};
int main() {
    Base b{"Remigijus"};
    std::cout << b.getVardas() << std::endl; // Ką čia gausime?
    Derived d{36, "Remis"};
    std::cout << d.getVardas() << std::endl; // Ką čia gausime?
    d.whoAmI();                              // Ką čia gausime?
}
```

Paveldėtų funkcijų naudojimas (2)

— Kai nario funkcija yra iškviečiama iš paveldėtosios klasės objekto, kompiliatorius pirmiausia ieško šios funkcijos paveldėtoje klasėje.

— Kadangi `Derived::getVardas()` nerado, tuomet "*lipa*" paveldėjimo medžiu aukštyn, tol kol suranda pirmąją tinkančią: `Base::getVardas()`.

— Išvestinėse klasėse paveldėtas funkcijas galima per naują apibrėžti!

Paveldėtų funkcijų perrašymas (1)

```
// Viskas kaip anksčiau
class Derived : public Base {
protected:
    int amzius;
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    int getAmzius() const { return amzius; }
    void whoAmI() { std::cout << "Aš esu Derived klasė\n"; }
};

int main() {
    Base b{"Remigijus"};
    b.whoAmI(); // Ką čia gausime?
    Derived d{36, "Remis"};
    d.whoAmI(); // Ką čia gausime?
}
```

Paveldėtų funkcijų perrašymas (2)

```
// Viskas kaip anksčiau
class Derived : public Base {
protected:
    int amzius;
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    int getAmzius() const { return amzius; }
    void whoAmI() { std::cout << "Aš esu Derived klasė\n"; }
    void fullHierarchy() { // apjungiame f-jas iš abiejų klasių
        Base::whoAmI();    // kviečiame iš Base klasės
        whoAmI();          // kviečiame iš Derived klasės
    }
};

int main() {
    Derived d{36, "Remis"};
    d.fullHierarchy(); // Ką čia gausime?
}
```

Paveldėtų funkcijų prieigos (pasiekiamumo) keitimas (1)

```
// čia: getVardas() yra ne public, o protected!
// Pasiekiamas tik Base ir Derived klasių viduje.
class Base {
protected:
    std::string vardas;
    std::string getVardas() const { return vardas; }
public:
    Base(std::string v = "") : vardas{v} { }
    void whoAmI() { std::cout << "Aš esu Base klasė\n"; }
};

int main() {
    Derived d{36, "Remis"};
    std::cout << d.getVardas() << std::endl; // Ką čia gausime?
}
```

Paveldėtų funkcijų prieigos (pasiekiamumo) keitimas (2)

C++ galima pakeisti paveldėtos klasės narių prieigos specifikatorių išvestinėje klasėje per **using** deklaraciją:

```
class Derived : public Base {
protected:
    int amzius;
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    // Base::getVardas() iš protected pakeičiame į public
    using Base::getVardas; // Svarbu: be skliaustų "()"!
};

int main() {
    Derived d{36, "Remis"};
    std::cout << d.getVardas() << std::endl; // Ką čia gausime?
    Base b{"Remigijus"};
    std::cout << b.getVardas() << std::endl; // O ką čia gausime?
}
```

Paveldėtų funkcijų prieigos (pasiekiamumo) keitimas (3)

— Galime pakeisti prieigos specifikatorius tik tiems **Base** nariams, kuriuos **Derived** klasė pasiekia, t.y., negalima iš `private` į `public` ar `protected`.

— Galima tą pati gauti ir be **using** *deklaracijos* (žodelio), tačiau šis būdas daugiau neberekomenduojamas.

— Galimas ir atvirkštinis būdas, t.y. sumažinti **Base** klasės narių prieigą **Derived** klasėje, t.y. paslėpti dalį funkcionalumo egzistuojančio **Base** klasėje.

Paveldėtų funkcijų prieigos (pasiekiamumo) keitimas (4)

```cpp
class Base {
protected:
    std::string vardas;
public:
    Base(std::string v = "") : vardas{v} { }
    std::string getVardas() const { return vardas; }
    void whoAmI() { std::cout << "Aš esu Base klasė\n"; }
};

class Derived : public Base {
protected:
    int amzius;
    using Base::whoAmI; // Padarėme, kad nebūtų pasiekiamas Derived objektams
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    int getAmzius() const { return amzius; }
};

int main() {
    Derived d{36, "Remis"};
    d.whoAmI(); // Ką čia gausime?
}
```

Paveldėtų funkcijų prieigos (pasiekiamumo) keitimas (5)

```
// Alternatyvus būdas: su `delete` irgi galima tą patį rezultatą gauti
class Derived : public Base {
protected:
    int amzius;
    //using Base::whoAmI; // Padarėme, kad nebūtų pasiekiamas Derived objektams
public:
    Derived(int a = 0, std::string v = "") : Base{v}, amzius{a} { }
    int getAmzius() const { return amzius; }
    void whoAmI() = delete; // pažymime šią funkciją neprieinamą
};
```

— Tokiu būdu galime prastai "suprojektuotas" bazines klases "ištaisyti" paveldėtose klasėse, pvz. `public` tipo duomenis paslepiant į `private`.

Daugialypis (*multiple*) paveldėjimas (1)

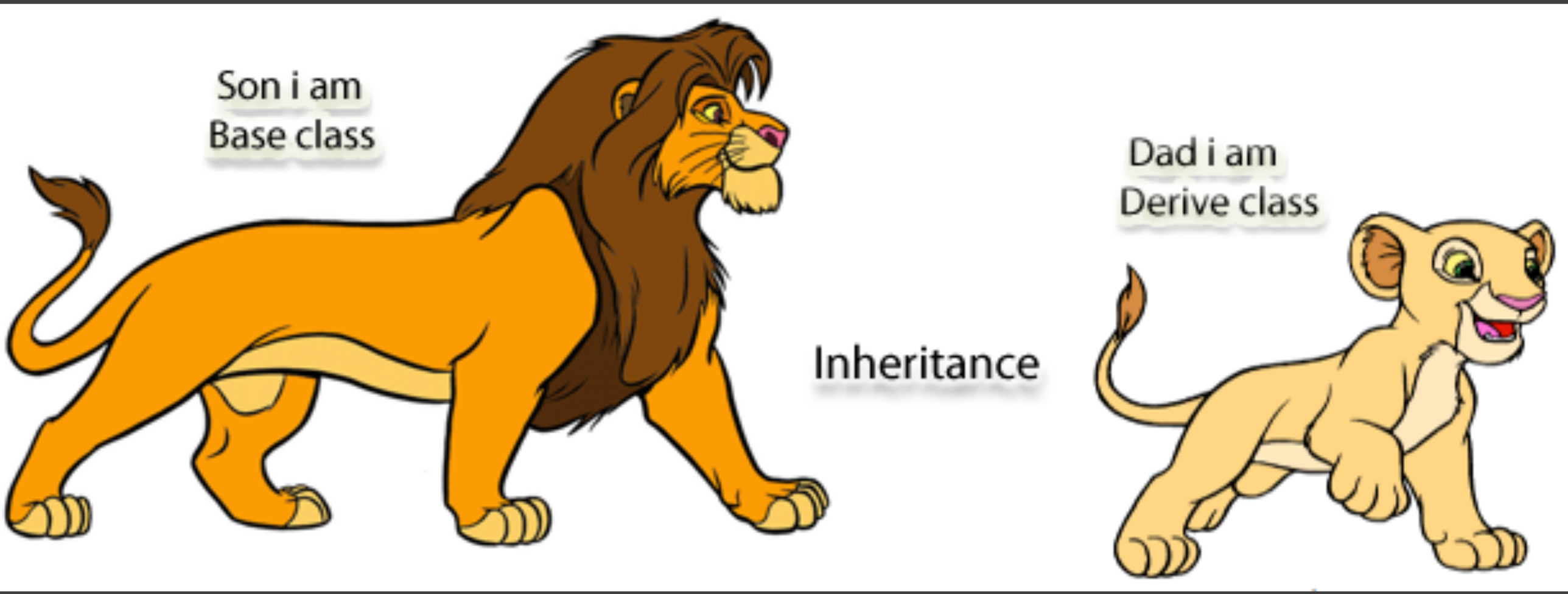
```
class Zmogus {
protected:
    int amzius;
    std::string vardas;
public:
    Zmogus(int amz = 0, std::string v = "") : amzius{amz}, vardas{v} { }
    int getAmzius() const { return amzius; }
    std::string getVardas() const { return vardas; }
};

class Darbuotojas {
protected:
    int alga;
public:
    Darbuotojas(int alg = 0) : alga{alg} { }
    int getAlga() const { return alga; }
};

// Studentas visgi yra žmogus :) ir tuo pat metu gali būti dirbantis!
class Studentas : public Zmogus, public  Darbuotojas {
private:
    double vidurkis;  // mokymosi vidurkis
public:
    Studentas(int amz = 0, std::string v = "", int alg = 0, double avg = 0)
        : Zmogus{amz, v}, Darbuotojas{alg}, vidurkis{avg} { }
    int getVidurkis() const { return vidurkis; }
};
```

Daugialypis (*multiple*) paveldėjimas (2)

```
// Includinam reikalingas klases ir kitus dependenc'us
int main() {
    Studentas s1{20, "Jonas", 1000, 7.5};
    Studentas s2{18, "Petras"};
    std::cout << s1.getVardas() << " uždirba: " << s1.getAlga() << std::endl;
    std::cout << s2.getVardas() << " uždirba: " << s2.getAlga() << std::endl;
}
```

— Daugiau žalos, negu naudos - geriau vengti.

— Pvz. jei kelios bazinės klasės turi funkciją tuo pačiu pavadinimu, kuri iškviečiama išvestinėje klasėje?


Son i am
Base class
Inheritance
Dad i am
Derive class