Nesne Yönelimli Programlamaya Giriş

Bu hafta

- Kalıtım

Kalıtım (Inheritance)

Kalıtım Nedir?

Kalıtım, bir sınıfın (türetilmiş sınıf) başka bir sınıftan (temel sınıf) özellik ve davranışları devralmasını sağlar.

Kodun yeniden kullanılabilirliğini ve modülerliğini artırır.

Kalıtım Türleri:

1. **Public** Kalıtım: Taban sınıfın üyeleri aynı erişim düzeyiyle kalıtılır.
2. **Protected** Kalıtım: Taban sınıfın `public` üyeleri türetilmiş sınıfta `protected` olur.
3. **Private** Kalıtım: Taban sınıfın tüm üyeleri türetilmiş sınıfta `private` olur.

Nesne Yönelimli Programlama (OOP) – Kalıtım (Inheritance)

Kalıtım (Inheritance) Nedir?

Kalıtım, Nesne Yönelimli Programlama'da (OOP) bir sınıfın ('base class' veya 'parent class') özelliklerini ve davranışlarını (üyeler ve metotlar) bir başka sınıfa ('derived class' veya 'child class') aktarımını sağlayan bir yapıdır.

Temel Kavramlar:

- **Parent Class (Base Class, Super Class):** Özellikleri ve metotları kalıtım yoluyla aktarılan sınıftır.
- **Child Class (Derived Class, Subclass):** Parent class'tan kalıtım alarak oluşturulan yeni sınıftır.
- **Reusability (Yeniden Kullanılabilirlik):** Kod tekrardan kaçınılır, temel özellikler bir kez tanımlanıp alt sınıflar tarafından kullanılabilir.
- **Extensibility (Genişletilebilirlik):** Yeni özellikler kolayca eklenebilir.

Kalıtım Çeşitleri

a) Tek Seviyeli Kalıtım (Single Level Inheritance)

Bir sınıf, tek bir parent class'tan kalıtım alır.

```cpp
#include<iostream>
using namespace std;

// Parent Class
class Hayvan {
public:
    void nefesAl() {
        cout << "Hayvan nefes alıyor." << endl;
    }
};

// Child Class
class Kedi : public Hayvan {
public:
    void miyavla() {
        cout << "Kedi miyavlıyor." << endl;
    }
};
```

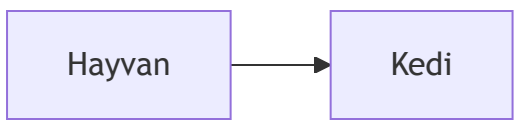
Örnek Kullanım:

```
int main() {
    Kedi k;
    k.nefesAl(); // Parent'tan alınan metod
    k.miyavla(); // Child'a ait metod
    return 0;
}
```

Çıktı:

```
Hayvan nefes alıyor.
Kedi miyavlıyor.
```

Çizim: Single Level Inheritance



b) Çok Seviyeli Kalıtım (Multi-Level Inheritance)

Bir sınıf, bir başka child class'ın parent class'ından kalıtım alır.

```cpp
#include<iostream>
using namespace std;

class Canli {
public:
    void hareketEt() {
        cout << "Canlı hareket ediyor." << endl;
    }
};
class Hayvan : public Canli {
public:
    void nefesAl() {
        cout << "Hayvan nefes alıyor." << endl;
    }
};
class Kedi : public Hayvan {
public:
    void miyavla() {
        cout << "Kedi miyavlıyor." << endl;
    }
};
```

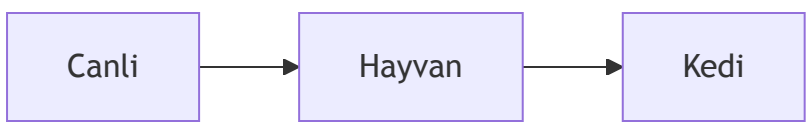
Örnek Kullanım:

```
int main() {
    Kedi k;
    k.hareketEt(); // Canli'dan miras
    k.nefesAl();   // Hayvan'dan miras
    k.miyavla();   // Kedi'ye ait metod
    return 0;
}
```

Çıktı:

```
Canlı hareket ediyor.
Hayvan nefes alıyor.
Kedi miyavlıyor.
```

Çizim: Multi-Level Inheritance



c) Çoklu Kalıtım (Multiple Inheritance)

Bir sınıf, birden fazla parent class'tan kalıtım alabilir.

```
#include<iostream>
using namespace std;
class Baba {
public:
    void arabaSur() {
        cout << "Baba araba sürüyor." << endl;
    }
};
class Anne {
public:
    void yemekYap() {
        cout << "Anne yemek yapıyor." << endl;
    }
};
class Cocuk : public Baba, public Anne {
public:
    void oyna() {
        cout << "Cocuk oyun oynuyor." << endl;
    }
};
```

Örnek Kullanım:

```
int main() {
    Cocuk c;
    c.arabaSur(); // Baba'dan miras
    c.yemekYap(); // Anne'den miras
    c.oyna();     // Cocuk'a ait metod
    return 0;
}
```

Çıktı:

```
Baba araba sürüyor.
Anne yemek yapıyor.
Cocuk oyun oynuyor.
```

Çizim: Multiple Inheritance

Kalıtımda Erişim Belirteçleri (Access Specifiers)

Kalıtımda `public`, `protected` ve `private` kullanılır.

Base Class	Public Inheritance	Protected Inheritance	Private Inheritance
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Erişilemez	Erişilemez	Erişilemez

Örnek:

```
class A {
public:
    int x = 10;
protected:
    int y = 20;
private:
    int z = 30;
};

class B : public A {
public:
    void yazdir() {
        cout << x; // Erişim var (Public)
        cout << y; // Erişim var (Protected)
        // cout << z; // Hata! (Private)
    }
};
```

Kalıtımda Yapıcı ve Yıkıcılar

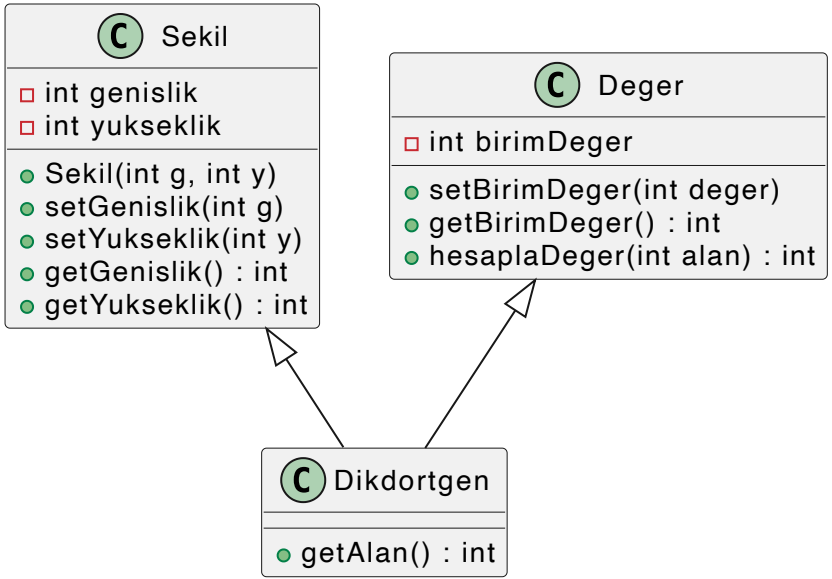
- Parent sınıfın yapıcısı (‘constructor’) ve yıkıcısı (‘destructor’) child sınıf tarafından kullanılabilir.

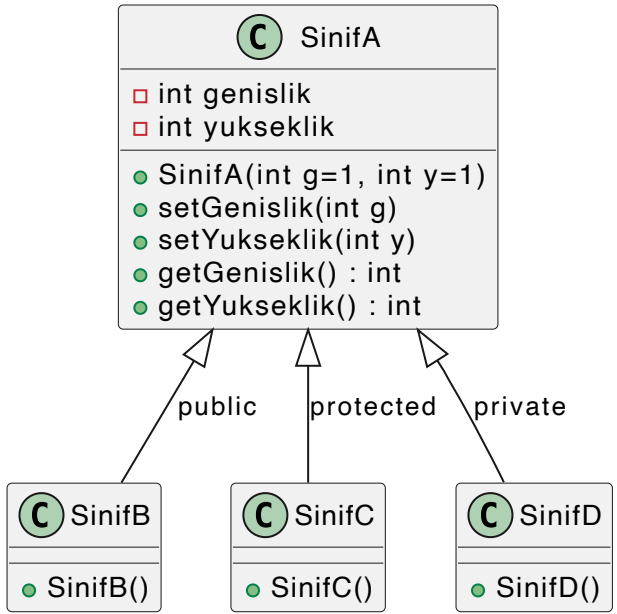
Örnek:

```
class Base {
public:
    Base() { cout << "Base Constructor\n"; }
    ~Base() { cout << "Base Destructor\n"; }
};
class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor\n"; }
    ~Derived() { cout << "Derived Destructor\n"; }
};
int main() {
    Derived obj;
    return 0;
}
```

Çıktı:

```
Base Constructor
Derived Constructor
```


Sekil
int genislik
int yukseklik
Sekil(int g, int y)
setGenislik(int g)
setYukseklik(int y)
getGenislik() : int
getYukseklik() : int
Deger
int birimDeger
setBirimDeger(int deger)
getBirimDeger() : int
hesaplaDeger(int alan) : int
Dikdortgen
getAlan() : int


SinifA
int genislik
int yukseklik
SinifA(int g=1, int y=1)
setGenislik(int g)
setYukseklik(int y)
getGenislik() : int
getYukseklik() : int
public
protected
private
SinifB
SinifB()
SinifC
SinifC()
SinifD
SinifD()

Kalıtım ile Polimorfizm Kullanımı

Kalıtım, sanal fonksiyonlar (‘virtual functions’) sayesinde polimorfizm sağlar.

Detaylar için sonraki ders: Polimorfizm

Nesne Yönelimli Programlama (OOP) – Polimorfizm (Polymorphism)

1. Polimorfizm Nedir?

Polimorfizm, Nesne Yönelimli Programlama'da (OOP) aynı isme sahip metotların farklı davranışlar sergilemesini sağlayan bir özelliktir. Kalıtım ile birleştirildiğinde, programlarda genişletilebilirliği ve esnekliği artırır.

Temel Kavramlar:

- **Compile-Time Polimorfizm (Statik Polimorfizm):** Fonksiyon aşırı yükleme (function overloading) ve operatör aşırı yükleme (operator overloading) kullanarak sağlanır.
- **Run-Time Polimorfizm (Dinamik Polimorfizm):** Sanal fonksiyonlar (virtual functions) ve geç bağlılık (late binding) kullanarak gerçekleştirilir.

2. Compile-Time Polimorfizm

a) Fonksiyon Aşırı Yükleme (Function Overloading)

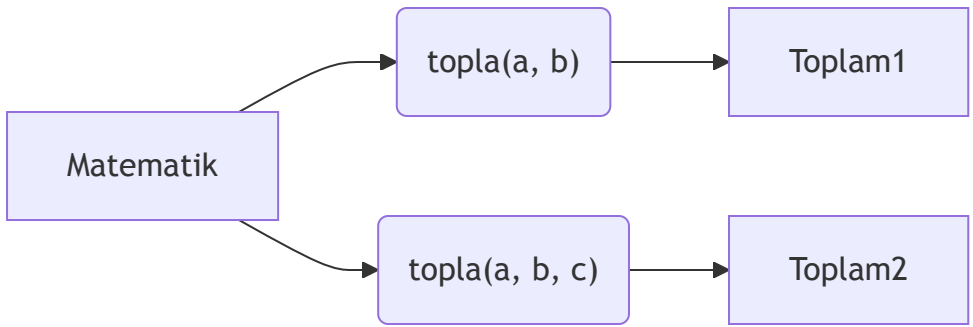
Aynı isimde fakat farklı parametre listesine sahip fonksiyonların tanımlanmasıdır.

```cpp
#include<iostream>
using namespace std;

class Matematik {
public:
    int topla(int a, int b) {
        return a + b;
    }
    int topla(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Matematik m;
    cout << "2 sayının toplamı: " << m.topla(5, 10) << endl;
    cout << "3 sayının toplamı: " << m.topla(5, 10, 15) << endl;
    return 0;
}
```

Çıktı:

```
2 sayının toplamı: 15
3 sayının toplamı: 30
```

Çizim: Fonksiyon Aşırı Yükleme



b) Operatör Aşırı Yükleme (Operator Overloading)

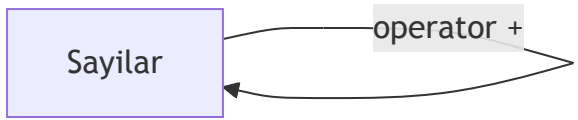
Operatörlerin özel anlamlarla yeniden tanımlanmasıdır.

```cpp
#include<iostream>
using namespace std;

class Sayilar {
    int deger;
public:
    Sayilar(int d) : deger(d) {}
    Sayilar operator + (Sayilar obj) {
        return Sayilar(deger + obj.deger);
    }
    void yazdir() {
        cout << "Deger: " << deger << endl;
    }
};

int main() {
    Sayilar s1(10), s2(20);
    Sayilar s3 = s1 + s2; // + operatorü aşırı yüklendi
    s3.yazdir();
    return 0;
}
```

Çıktı:

```
Deger: 30
```

Çizim: Operatör Aşırı Yükleme



```cpp
#include <iostream>
using namespace std;

class Kuvvet {
private:
    double x, y; // Kuvvetin X ve Y bileşenleri

public:
    // Yapıcı Fonksiyon
    Kuvvet(double x = 0, double y = 0) : x(x), y(y) {}

    // + Operatörünün Aşırı Yüklenmesi (İki kuvveti toplar)
    Kuvvet operator+(const Kuvvet& other) const {
        return Kuvvet(x + other.x, y + other.y);
    }

    // Kuvveti ekrana yazdırma fonksiyonu
    void yazdir() const {
        cout << "Kuvvet: (" << x << ", " << y << ")" << endl;
    }
};
```

```cpp
int main() {
    // İki farklı kuvvet
    Kuvvet f1(10, 5);  // Örneğin, X ekseninde 10 N, Y ekseninde 5 N
    Kuvvet f2(3, 7);   // X ekseninde 3 N, Y ekseninde 7 N

    // Kuvvetlerin toplamı
    Kuvvet toplam = f1 + f2;

    // Sonuçları yazdır
    cout << "Birinci Kuvvet: ";
    f1.yazdir();

    cout << "İkinci Kuvvet: ";
    f2.yazdir();

    cout << "Toplam Kuvvet: ";
    toplam.yazdir();

    return 0;
}
```

Çıktı:

```
Birinci Kuvvet: (10, 5)
İkinci Kuvvet: (3, 7)
Toplam Kuvvet: (13, 12)
```

```cpp
#include <iostream>
using namespace std;

class Vektor3D {
private:
    double x, y, z;

public:
    // Yapıcı Fonksiyon
    Vektor3D(double x = 0, double y = 0, double z = 0) : x(x), y(y), z(z) {}

    // + Operatörünün Aşırı Yüklenmesi (İki vektörün toplamını yapar)
    Vektor3D operator+(const Vektor3D& other) const {
        return Vektor3D(x + other.x, y + other.y, z + other.z);
    }

    // - Operatörünün Aşırı Yüklenmesi (İki vektörün farkını alır)
    Vektor3D operator-(const Vektor3D& other) const {
        return Vektor3D(x - other.x, y - other.y, z - other.z);
    }

    // Ekrana Yazdırma (Arkadaş Fonksiyon)
    friend ostream& operator<<(ostream& os, const Vektor3D& vektor) {
        os << "(" << vektor.x << ", " << vektor.y << ", " << vektor.z << ")";
        return os;
    }
};
```

```
int main() {
    // Robot kolunun başlangıç ve hedef pozisyonları
    Vektor3D baslangic(10.5, 20.0, 15.3);
    Vektor3D hedef(5.5, 10.0, 5.3);

    // Operatör Aşırı Yükleme Kullanımı
    Vektor3D hareket = hedef - baslangic; // Hedefe ulaşmak için gereken hareket
    Vektor3D yeniPozisyon = baslangic + hareket; // Yeni pozisyon

    cout << "Başlangıç Pozisyonu: " << baslangic << endl;
    cout << "Hedef Pozisyonu: " << hedef << endl;
    cout << "Hareket Vektörü: " << hareket << endl;
    cout << "Yeni Pozisyon: " << yeniPozisyon << endl;

    return 0;
}
```

Çıktı:

```
Başlangıç Pozisyonu: (10.5, 20, 15.3)
Hedef Pozisyonu: (5.5, 10, 5.3)
Hareket Vektörü: (-5, -10, -10)
Yeni Pozisyon: (5.5, 10, 5.3)
```

3. Run-Time Polimorfizm

Run-Time Polimorfizm, sanal fonksiyonlar (virtual functions) kullanarak kalıtım ile sağlanır. Anahtar sözcük `virtual` kullanılır.

Sanal Fonksiyon (Virtual Function) Örneği

```
#include<iostream>
using namespace std;
class Sekil {
public:
    virtual void ciz() {
        cout << "Sekil çiziliyor..." << endl; }
};

class Dikdortgen : public Sekil {
public:
    void ciz() override {
        cout << "Dikdörtgen çiziliyor..." << endl; }
};

class Cember : public Sekil {
public:
    void ciz() override {
        cout << "Çember çiziliyor..." << endl;  }
};
```

```
int main() {
    Sekil* sekil1 = new Dikdortgen();
    Sekil* sekil2 = new Cember();

    sekil1->ciz(); // Dikdörtgen
    sekil2->ciz(); // Çember

    delete sekil1;
    delete sekil2;
    return 0;
}
```

Çıktı:

```
Dikdörtgen çiziliyor...
Çember çiziliyor...
```

Çizim: Run-Time Polimorfizm

4. Polimorfizmin Avantajları

- **Kodun Esnekliği:** Aynı arayüzü kullanarak farklı nesnelerin davranışlarını yönetir.
- **Kod Tekrarını Azaltma:** Aynı isimle farklı işlevler sunulabilir.
- **Bakım Kolaylığı:** Yeni sınıflar eklemek mevcut kodu bozmadan mümkündür.

Overload ve Override Arasındaki Farklar

1. Overload (Aşırı Yükleme)

Tanım

Bir sınıfta aynı isimde, ancak farklı **parametre listelerine** sahip birden fazla fonksiyon tanımlanmasıdır. Bu yöntemle, aynı işleve sahip ancak farklı türlerde veya sayıda parametre kabul eden fonksiyonlar oluşturulabilir.

Özellikler

- Parametrelerin sayısı, türü veya sırası farklı olmalıdır.
- Aynı sınıf içinde gerçekleşir.
- Geri dönüş tipi overload için fark yaratmaz (yalnızca parametre listesi önemlidir).

Örnek

```cpp
#include <iostream>
using namespace std;

class Hesaplayici {
public:
    int topla(int a, int b) {
        return a + b;
    }

    double topla(double a, double b) {
        return a + b;
    }

    int topla(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Hesaplayici h;
    cout << "2 + 3 = " << h.topla(2, 3) << endl;
    cout << "2.5 + 3.5 = " << h.topla(2.5, 3.5) << endl;
    cout << "1 + 2 + 3 = " << h.topla(1, 2, 3) << endl;
    return 0;
}
```

2. Override (Geçersiz Kılma)

Tanım

Bir **türetilmiş sınıfta** (alt sınıfta), temel sınıfta tanımlanmış bir **sanal fonksiyonun (virtual)** yeniden tanımlanmasıdır. Override, genellikle **kalıtım** (inheritance) ile ilgilidir.

Özellikler

- Fonksiyon adı, parametre listesi ve geri dönüş tipi temel sınıftakiyle **tamamen aynı** olmalıdır.
- Geçersiz kılma işlemi için temel sınıfta sanal (virtual) olarak işaretlenmiş bir fonksiyon bulunmalıdır.
- `override` anahtar kelimesi (isteğe bağlı) kodun okunabilirliğini artırır ve derleyici denetimi sağlar.

Örnek

```cpp
#include <iostream>
using namespace std;
class Sekil {
public:
    virtual void ciz() {
        cout << "Bir şekil çiziliyor." << endl; }
};

class Daire : public Sekil {
public:
    void ciz() override {
        cout << "Bir daire çiziliyor." << endl;  }
};

int main() {
    Sekil* sekil = new Daire();
    sekil->ciz(); // Daire'nin ciz() fonksiyonu çağrılır.
    delete sekil;
    return 0;
}
```

Çıktı:

```
Bir daire çiziliyor.
```

Overload ve Override Arasındaki Farklar

Özellik	**Overload**	**Override**
Tanım	Aynı isimli fonksiyonları farklı parametrelerle tanımlamak.	Türetilmiş sınıfta temel sınıf fonksiyonunu yeniden tanımlamak.
Kapsam	Aynı sınıf içinde gerçekleşir.	Türetilmiş sınıf ve temel sınıf arasında gerçekleşir.
Parametre Listesi	Farklı olmalıdır.	Aynı olmalıdır.
Virtual Gereksinimi	Yoktur.	Temel sınıfta `virtual` olmalıdır.
Kullanım Alanı	Fonksiyonlara esneklik kazandırmak.	Polimorfizm ve dinamik bağlama sağlamak.

Overload ve Override Karşılaştırmalı Kod Örneği

Overload Örneği

```cpp
#include <iostream>
using namespace std;

class Hesaplayici {
public:
    int carp(int a, int b) {
        return a * b;
    }

    double carp(double a, double b) {
        return a * b;
    }
};

int main() {
    Hesaplayici h;
    cout << "2 * 3 = " << h.carp(2, 3) << endl;
    cout << "2.5 * 3.5 = " << h.carp(2.5, 3.5) << endl;
    return 0;
}
```

Override Örneği

```cpp
#include <iostream>
using namespace std;

class Arac {
public:
    virtual void hareket() {
        cout << "Araç hareket ediyor." << endl;
    }
};

class Araba : public Arac {
public:
    void hareket() override {
        cout << "Araba hareket ediyor." << endl;
    }
};

int main() {
    Arac* arac = new Araba();
    arac->hareket(); // Araba'nın hareket() fonksiyonu çağrılır.
    delete arac;
    return 0;
}
```

Çıktı:

```
2 * 3 = 6
2.5 * 3.5 = 8.75
Araç hareket ediyor.
Araba hareket ediyor.
```

Bir sonraki konu: Soyut Sınıflar ve Arayüzler (Abstract Classes & Interfaces)