

PROGRAMOWANIE GENERYCZNE

Funkcja zamien (C#)

```
...  
void zamien(ref int a, ref int b)  
{  
    int tmp = a;  
    a=b;  
    b=tmp;  
}  
...
```

```
...  
void zamien(ref double a, ref double b)  
{  
    double tmp = a;  
    a=b;  
    b=tmp;  
}  
...
```

```
...  
void zamien(ref string a,ref string b)  
{  
    string tmp = a;  
    a=b;  
    b=tmp;  
}  
...
```

```
...  
void zamien(ref char a, ref char b)  
{  
    char tmp = a;  
    a=b;  
    b=tmp;  
}  
...
```

Klasa Punkt (C#)

```
class Para {  
    public int Pierwszy { get; }  
    public int Drugi { get; }  
  
    public Para(int pierwszy,  
                int drugi)  
    {  
        Pierwszy = pierwszy;  
        Drugi = drugi;  
    }  
}
```

```
class Para {  
    public char Pierwszy { get; }  
    public char Drugi { get; }  
  
    public Para(char pierwszy,  
                char drugi)  
    {  
        Pierwszy = pierwszy;  
        Drugi = drugi;  
    }  
}
```

Klasa Punkt (C#)

```
class Para {  
    public object Pierwszy { get; }  
    public object Drugi { get; }  
  
    public Para(object pierwszy,  
                object drugi)  
    {  
        Pierwszy = pierwszy;  
        Drugi = drugi;  
    }  
}
```

Programowanie generyczne

Programowanie generyczne polega na tworzeniu algorytmów i struktur danych operujących na danych dowolnego typu (lub prawie dowolnego typu).

Paradygmat programowania generycznego jest najczęściej rozumiany jako uogólnienie paradygmatu programowania obiektowego.

Programowanie generyczne w językach programowania

Generyki



Szablony



Metody generyczne w C#

```
class Klasa
{
    public static void zamien<T> (ref T a, ref T b)
    {
        T tmp = a;
        a=b;
        b=tmp;
    }
}
```

Klasy generyczne w C#

```
class Para<T> {  
    public T Pierwszy { get; }  
    public T Drugi { get; }  
  
    public Para(T pierwszy,  
                T drugi)  
    {  
        Pierwszy = pierwszy;  
        Drugi = drugi;  
    }  
}
```


Klasy generyczne w C#

```
class Para<T> {  
    public T Pierwszy { get; }  
    public T Drugi { get; }  
  
    public Para()  
    {  
        Pierwszy = default(T);  
        Drugi = default(T);  
    }  
}
```

Metody generyczne

```
class Para<T> {  
    public T Pierwszy { get; private set;}  
    public T Drugi { get; }  
  
    public Para()  
    {  
        Pierwszy = default(T);  
        Drugi = default(T);  
    }  
  
    void Zmien(T pierwszy) { Pierwszy = pierwszy; }  
}
```

Wiele zmiennych typu

```
class Para<TPierwszy, TDrugi> {  
    public TPierwszy Pierwszy { get; private set;}  
    public TDrugi Drugi { get; }  
  
    public Para(TPierwszy pierwszy, TDrugi drugi)  
    {  
        Pierwszy = pierwszy;  
        Drugi = drugi;  
    }  
}
```

Ograniczenia generyczne (1)

```
class Para<T>
{
    public T Pierwszy { get; }
    public T Drugi { get; }

    public Para(T pierwszy, T drugi)
    {
        Pierwszy = pierwszy;
        Drugi = drugi;
    }

    public T max()
    {
        return Pierwszy < Drugi ? Drugi : Pierwszy;
    }
}
```

Ograniczenia generyczne (1)

```
class Para<T> where T:IComparable<T>
{
    public T Pierwszy { get; }
    public T Drugi { get; }

    public Para(T pierwszy, T drugi)
    {
        Pierwszy = pierwszy;
        Drugi = drugi;
    }

    public T max()
    {
        return Pierwszy.CompareTo(Drug) < 0
            ? Drugi : Pierwszy;
    }
}
```

Ograniczenia generyczne (2)

```
class Para<T> where T: IComparable<T>, ICloneable
{
    public new T Pierwszy { get; }
    public new T Drugi { get; }

    public Para(T pierwszy, T drugi)
    {
        Pierwszy = pierwszy;
        Drugi = drugi;
    }

    public T max()
    {
        return Pierwszy.CompareTo(Drug) < 0
            ? Drugi : Pierwszy;
    }
}
```

Ograniczenia generyczne (3)

```
class ListaUporzadkowana<K, V>  
    where K : IComparable<K>  
{  
  
}
```

```
class ListaUporzadkowana<K, V>  
    where K : IComparable<K>  
    where V : ICloneable  
{  
  
}
```

Rodzaje ograniczeń generycznych

- ograniczenia dziedziczenia (parametr typu musi dziedziczyć po określonym typie)

```
class KlasaBazowa {}  
class MojaKlasa<T> where T : KlasaBazowa {}
```

```
class MojaKlasa<T,U> where T : U { }
```

- ograniczenia konstruktora (parametr typu musi posiadać konstruktor domyślny)

```
class MojaKlasa<T> where T : new() {}
```

- ograniczenia typu

```
class MojaKlasa<T> where T : struct {}  
class MojaKlasa<T> where T : class {}
```


Kowariancja, kontrawariancja i inwariancja

Kowariancja pozwala na wykorzystanie bardziej szczegółowego typu niż pierwotnie określono

Kontrawariancja pozwala na wykorzystanie mniej szczegółowego typu niż pierwotnie określono

Inwariancja pozwala na wykorzystanie tylko takiego typu, który był oryginalnie określony.

Kowariancja, kontrawariancja i inwariancja

Kowariancja

Typ może być wykorzystany tylko jako typ zwracany

```
interface
przyklad<out T> {
    T metoda();
}
```

Kontrawariancja

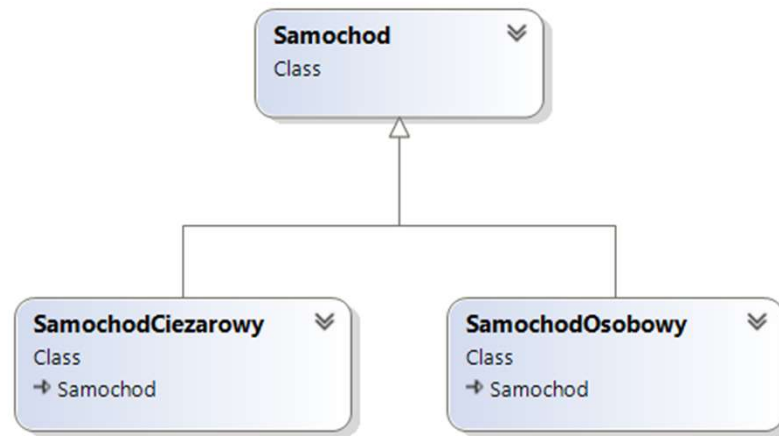
Typ może być wykorzystany tylko jako typ parametru

```
interface
przyklad<in T> {
    void metoda(T p);
}
```

Inwariancja

Typ może być wykorzystany jako typ zwracany lub typ parametru

```
interface
przyklad<T> {
    T metoda();
    void metoda(T p);
}
```



```
class Samochod {}
```

```
class SamochodOsobowy : Samochod {}
```

```
class SamochodCiezarowy : Samochod {}
```

Dziedziczenie, a przypisanie





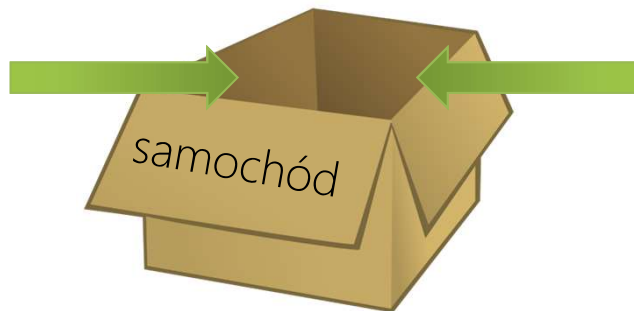
```
interface IFabryka<T>
{
    T Stworz();
}
```

```
class FabrykaFerrari : IFabryka<SamochodOsobowy> {  
    public SamochodOsobowy Stworz() {  
        throw new NotImplementedException();  
    }  
}
```

```
class FabrykaScania : IFabryka<SamochodCiezarowy> {  
    public SamochodCiezarowy Stworz() {  
        throw new NotImplementedException();  
    }  
}
```

```
class FabrykaVolvo : IFabryka<Samochod> {  
    public Samochod Stworz() {  
        throw new NotImplementedException();  
    }  
}
```

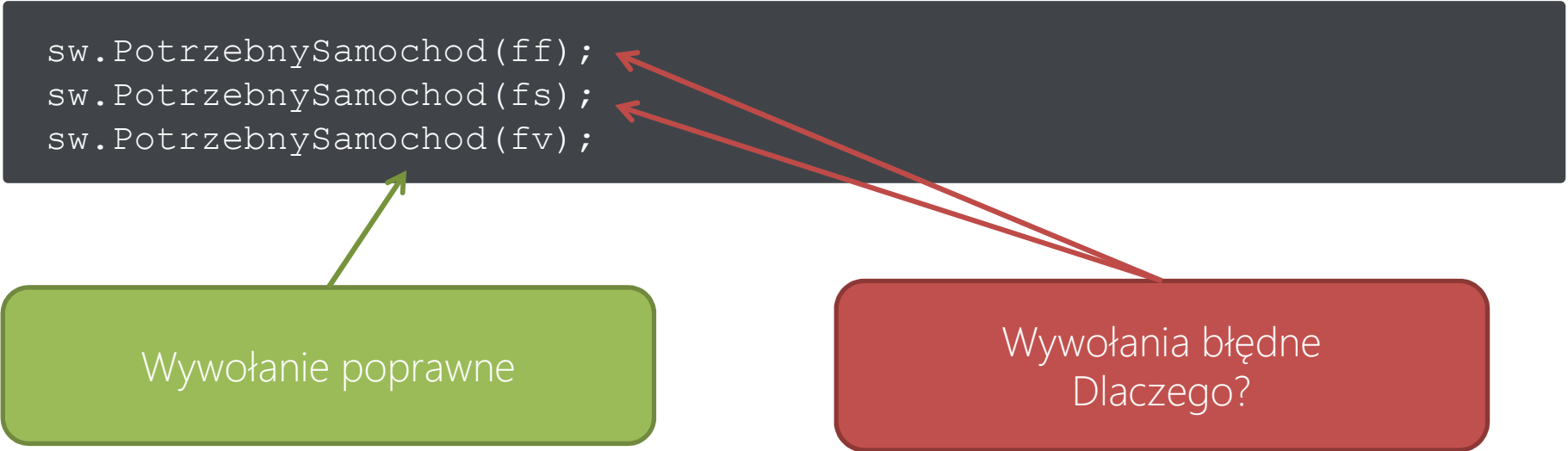
```
class SwiatWirutalny {  
  
    private Samochod samochod;  
  
    public void PotrzebnySamochod(IFabryka<Samochod> fabryka) {  
        samochod = fabryka.Stworz();  
    }  
}
```



```
class Program
{
    static void Main(string[] args)
    {
        SwiatWirutalny sw = new SwiatWirutalny();
        FabrykaFerrari ff = new FabrykaFerrari();
        FabrykaScania fs = new FabrykaScania();
        FabrykaVolvo fv = new FabrykaVolvo();
        ...
    }
}
```



```
sw.PotrzebnySamochod(ff);  
sw.PotrzebnySamochod(fs);  
sw.PotrzebnySamochod(fv);
```



Wywołanie poprawne

Wywołania błędne
Dlaczego?

Interfejs IFabryka<T> jest inwariantny

Inwariancja zwala na wykorzystanie tylko takiego typu, który był oryginalnie określony.

Aby nasz świat stał się idealny musimy interfejs przekształcić w kowariantny

```
interface IFabryka<out T>
{
    T Stworz();
}
```

Kowariancja pozwala na wykorzystanie bardziej szczegółowego typu niż pierwotnie określono

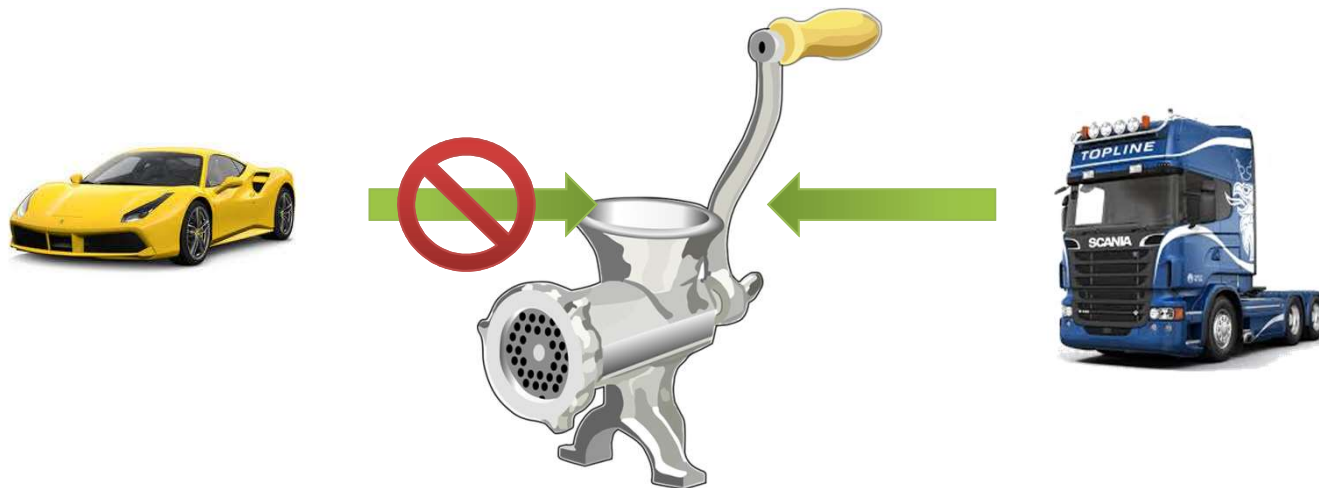


```
interface IService<T>
{
    void Napraw(T obiekt);
}
```

```
class ASOFerrari : ISerwis<SamochodOsobowy> {  
    public void Napraw(SamochodOsobowy obiekt) {  
        throw new NotImplementedException();  
    }  
}
```



```
class ASOScania : ISerwis<SamochodCiezarowy> {  
    public void Napraw(SamochodCiezarowy obiekt) {  
        throw new NotImplementedException();  
    }  
}
```



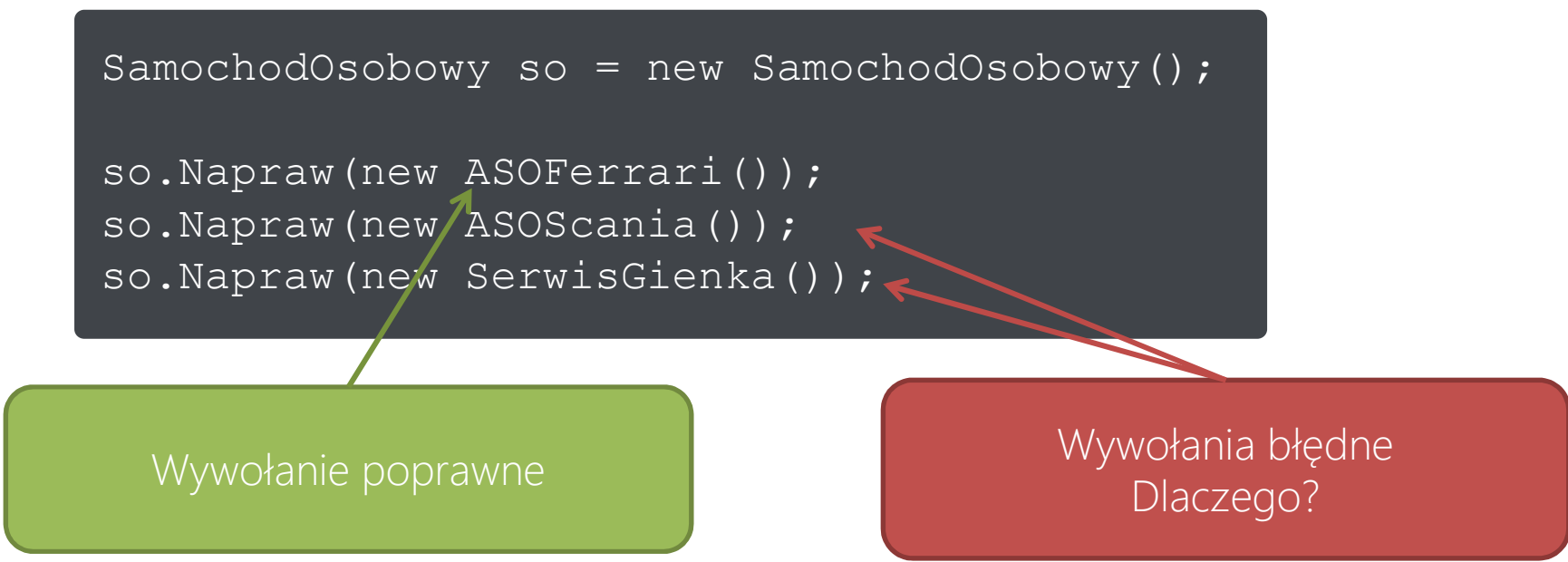
```
class SerwisGienka : ISerwis<Samochod> {  
    public void Napraw(Samochod obiekt) {  
        throw new NotImplementedException();  
    }  
}
```



```
class SamochodOsobowy : Samochod {  
    public void Napraw(ISerwis<SamochodOsobowy> serwis)  
    {  
        serwis.Napraw(this);  
    }  
}
```

```
class SamochodCiezarowy : Samochod {  
    public void Napraw(ISerwis<SamochodCiezarowy> serwis)  
    {  
        serwis.Napraw(this);  
    }  
}
```

```
SamochodOsobowy so = new SamochodOsobowy();  
  
so.Napraw(new ASOFerrari());  
so.Napraw(new ASOScania());  
so.Napraw(new SerwisGienka());
```



Wywołanie poprawne

Wywołania błędne
Dlaczego?

Interfejs `ISerwis<T>` jest inwariantny

Aby nasz świat stał się idealny musimy interfejs przekształcić w kontrawariantny

```
interface ISerwis<in T>
{
    void Napraw(T obiekt);
}
```