

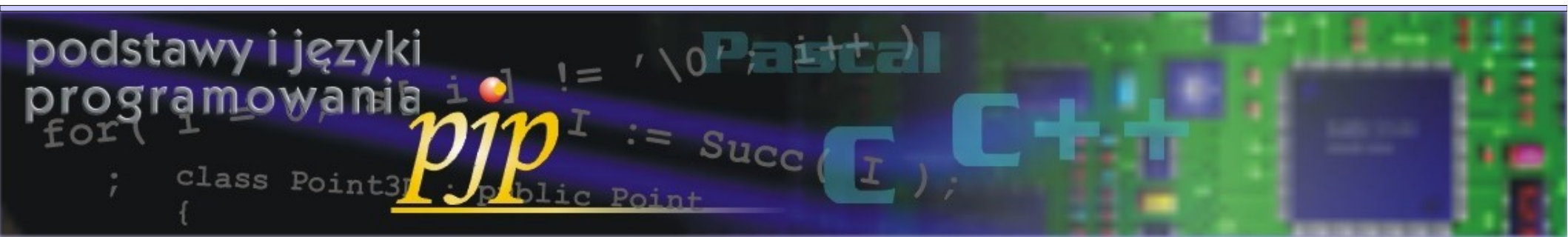
Projektowanie obiektowe

Roman Simiński

roman.siminski@us.edu.pl

www.siminskionline.pl

Klasy abstrakcyjne Interfejsy



Klasa abstrakcyjna – po co?

W programowaniu obiektowym wykorzystywane są:

- ▶ **Abstrakcja** – programowanie na wysokim poziomie ogólności, ukierunkowane na *identyfikowanie głównych obiektów* w systemie i ich *najważniejszych zachowań* oraz na definiowanie *głównych zależności* pomiędzy *obiektami* oraz ogólnych *scenariuszy* ich zachowań.
- ▶ **Rozszerzalność** – programowanie pozwalające na definiowanie *szczególnych zachowań* obiektów, realizowanych w ramach *ogólnych scenariuszy*, co zwykle osiąga się z wykorzystaniem *dziedziczenia* i *polimorfizmu*.

Środkiem do osiągnięcia rozszerzalności jest:

- ▶ **Antycypacja** – przewidywanie oraz odwzorowanie w kodzie przyszłych, choć jeszcze nie do końca znanych *zmian* i *modyfikacji* wymagań dla systemu.

Wskaźniki do obiektów a hierarchia klas

Konwersja wskaźników C++

Upcasting, czyli dziecko staje się rodzicem

```
class Silnik
{
public:
    int pojemnosc;
    int lbCyldindrow;
    int moc;
};
```

Upcasting występuje wtedy, gdy do wskaźnika lub referencji klasy bazowej, przypisujemy odwołanie do obiektu klasy pochodnej.

Z obiektem klasy pochodnej możemy robić wszystko to, co z obiektem klasy bazowej (relacja *is-a*).

```
class SilnikTurbo : public Silnik
{
public:
    float cisnienieDoladowania;
};
```

```
...
Silnik * s;
s = new SilnikTurbo;
s->pojemnosc = 2400;
s->moc = 163;
s->lbCyldindrow = 5;
```

Upcasting nie wymaga żadnych dodatkowych operacji, jest legalny i bezpieczny.

```
cout << endl << s->pojemnosc;
cout << endl << s->moc;
cout << endl << s->lbCyldindrow;
```

Upcasting działa ze wskaźnikami i referencjami

```
Silnik * s;  
s = new SilnikTurbo;  
s->pojemnosc = 2400;  
s->moc = 163;  
s->lbCylindrow = 5;
```

Upcasting via wskaźnik i referencja, obiekt anonimowy tworzony dynamicznie

```
Silnik & sr = *s;  
sr.pojemnosc = 2400;  
sr.moc = 163;  
sr.lbCylindrow = 5;
```

```
SilnikTurbo silT;
```

```
Silnik * s = &silT;  
s->pojemnosc = 2400;  
s->moc = 163;  
s->lbCylindrow = 5;
```

Upcasting via wskaźnik i referencja do zwykłego obiektu

```
Silnik & sr = silT;  
sr.pojemnosc = 3200;  
sr.moc = 240;  
sr.lbCylindrow = 8;
```

Upcasting oznacza chwilowe ograniczenie dostępności pól „pochodnych”

```
class Silnik
{
public:
    int pojemnosc;
    int lbCylindrow;
    int moc;
};
```

Utworzony obiekt jest reprezentantem klasy pochodnej *SilnikTurbo*, jednak lokalizujący go wskaźnik związany jest z klasą bazową *Silnik*. Doszło do upcastingu — dla kompilatora wskazywany obiekt jest teraz obiektem klasy *Silnik*.

```
class SilnikTurbo : public Silnik
{
public:
    float cisnienieDoladowania;
};
```

Ponieważ wskaźnik jest związany z klasą *Silnik*, a w niej nie ma pola *cisnienieDoladowania*, odwołania to są nieprawidłowe — mimo że tak naprawdę wskazywany obiekt jest klasy *SilnikTurbo*.

```
..
Silnik * s;
s = new SilnikTurbo;
s->pojemnosc = 2400;
s->moc = 163;
s->lbCylindrow = 5;
s->cisnienieDoladowania = 0.9;

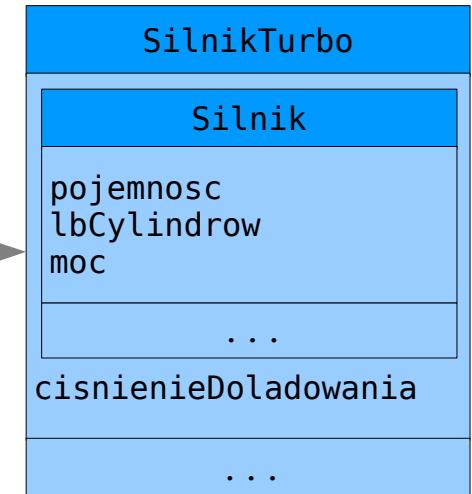
cout << endl << s->pojemnosc;
cout << endl << s->moc;
cout << endl << s->lbCylindrow;
cout << endl << s->cisnienieDoladowania;
```

Upcasting czasem oznacza ograniczenie dostępności pól

Dostęp do obiektu poprzez wskaźnik jego klasy

```
SilnikTurbo * s;  
s = new SilnikTurbo;  
s->pojemnosc = 2400;  
s->moc = 163;  
s->lbCylindrow = 5;  
s->cisnienieDoladowania = 0.9;
```

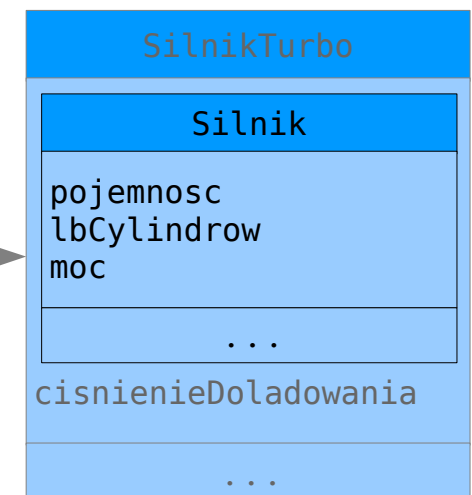
Pełny dostęp do
własnych pól oraz
pól odziedziczonych.



Dostęp do obiektu poprzez wskaźnik jego klasy bazowej — upcasting

```
Silnik * s;  
s = new SilnikTurbo;  
s->pojemnosc = 2400;  
s->moc = 163;  
s->lbCylindrow = 5;  
s->cisnienieDoladowania = 0.9;
```

Dostępne tylko
pola odziedziczone, czyli
jak w klasie bazowej.



Dlaczego upcasting jest potrzebny? Bo jest wygodny! I daje możliwości!

```
void zerujBazoweDaneSilnika( Silnik * s )
{
    s->pojemnosc = 0;
    s->moc = 0;
    s->lbCylindrow = 0;
}
void pokazBazoweDaneSilnika( Silnik * s )
{
    cout << endl << s->pojemnosc;
    cout << endl << s->moc;
    cout << endl << s->lbCylindrow;
}

. . .
Silnik sil;
SilnikTurbo silT;
SilnikHybryda silH;

zerujBazoweDaneSilnika( &sil );
zerujBazoweDaneSilnika( &silT );
zerujBazoweDaneSilnika( &silH );
silT.cisnienieDoladowania = 0;
silH.mocElektryczna = 0;

pokazBazoweDaneSilnika( &sil );
pokazBazoweDaneSilnika( &silT );
pokazBazoweDaneSilnika( &silH );
```

```
class Silnik
{
public:
    int pojemnosc;
    int lbCylindrow;
    int moc;
};

class SilnikTurbo
: public Silnik
{
public:
    float cisnienieDoladowania;
};

class SilnikHybryda
: public Silnik
{
public:
    int mocElektryczna;
};
```


Upcasting działa również z referencjami

```
void zerujBazoweDaneSilnika( Silnik & s )
{
    s.pojemnosc = 0;
    s.moc = 0;
    s.lbCylindrow = 0;
}
void pokazBazoweDaneSilnika( Silnik & s )
{
    cout << endl << s.pojemnosc;
    cout << endl << s.moc;
    cout << endl << s.lbCylindrow;
}

. . .
Silnik sil;
SilnikTurbo silT;
SilnikHybryda silH;

zerujBazoweDaneSilnika( sil );
zerujBazoweDaneSilnika( silT );
zerujBazoweDaneSilnika( silH );
silT.cisnienieDoladowania = 0;
silH.mocElektryczna = 0;

pokazBazoweDaneSilnika( sil );
pokazBazoweDaneSilnika( silT );
pokazBazoweDaneSilnika( silH );
```

```
class Silnik
{
public:
    int pojemnosc;
    int lbCylindrow;
    int moc;
};

class SilnikTurbo
: public Silnik
{
public:
    float cisnienieDoladowania;
};

class SilnikHybryda
: public Silnik
{
public:
    int mocElektryczna;
};
```

Czy jest downcasting? Tak, ale on może być przyczyną kłopotów...

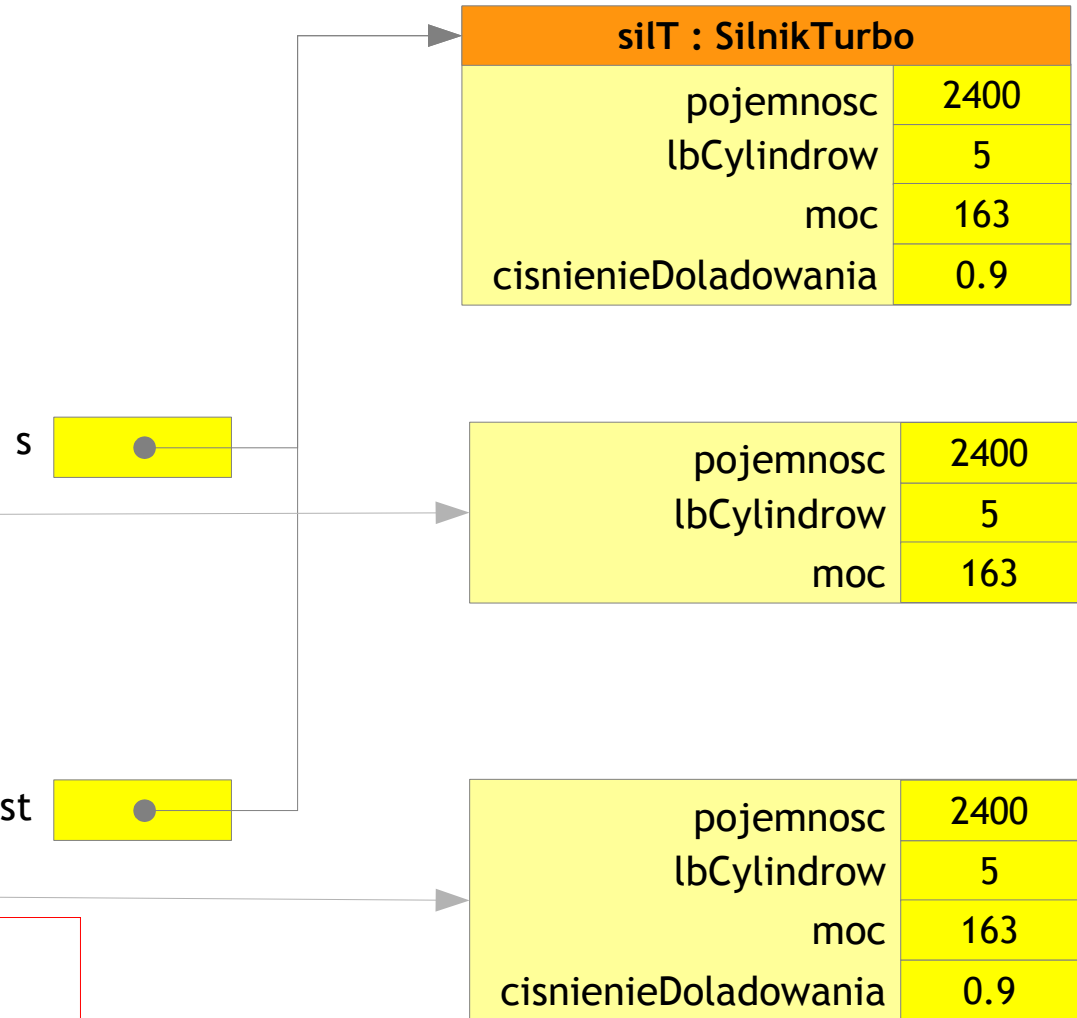
```
SilnikTurbo silT;  
s.pojemnosc = 2400;  
s.moc = 163;  
s.lbCylindrow = 5;  
s.cisnienieDoladowania = 0.9;
```

Upcasting:

```
Silnik * s;  
s = &silT;  
s->cisnienieDoladowania = 0.5;  
. . .
```

Downcasting:

```
. . .  
SilnikTurbo * st;  
st = ( SilnikTurbo * )s;  
st->cisnienieDoladowania = 0.5;
```



*Downcasting: konwersja wskazania na obiekt klasy bazowej do wskaźnika na klasę pochodną.
Możliwe tylko poprzez konwersję typów wskaźników jawnie zapisaną przez programistę.*

Wskaźniki pozwalają „patrzeć” na obiekt w różny sposób

```
SilnikTurbo silT;  
s.pojemnosc = 2400;  
s.moc = 163;  
s.lbCylindrow = 5;  
s.cisnienieDoladowania = 0.9;
```

Upcasting:

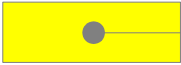
```
Silnik * s;  
s = &silT;  
s->cisnienieDoladowania = 0.5;  
. . .
```

Downcasting:

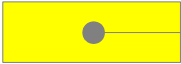
```
. . .  
SilnikTurbo * st;  
st = ( SilnikTurbo * )s;  
st->cisnienieDoladowania = 0.5;
```

silT : SilnikTurbo	
pojemnosc	2400
lbCylindrow	5
moc	163
cisnienieDoladowania	0.9

Wskaźnik s „widzi” obiekt silT tak:

s	
pojemnosc	2400
lbCylindrow	5
moc	163

Wskaźnik st „widzi” obiekt silT tak:

st	
pojemnosc	2400
lbCylindrow	5
moc	163
cisnienieDoladowania	0.9

W tym przypadku downcasting jest bezpieczny — obiekt wskazywany przez s jest rzeczywiście obiektem klasy *SilnikTurbo* i downcasting „odslania” przesłonięte pole *cisnienieDoladowania*.

Ale teraz będą kłopoty...

```
Silnik sil;  
s.pojemnosc = 2400;  
s.moc = 163;  
s.lbCylindrow = 5;
```

Zakotwiczenie wskaźnika:

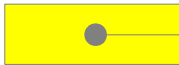
```
Silnik * s;  
s = &sil;  
s->cisnienieDoładowania = 0.5;  
. . .
```

Downcasting:

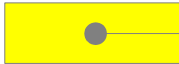
```
. . .  
SilnikTurbo * st;  
st = ( SilnikTurbo * )s;  
st->cisnienieDoładowania = 0.5;
```

sil : Silnik	
pojemnosc	2400
lbCylindrow	5
moc	163

Wskaźnik s „widzi” obiekt silT tak:

s								
		<table><tr><td>pojemnosc</td><td>2400</td></tr><tr><td>lbCylindrow</td><td>5</td></tr><tr><td>moc</td><td>163</td></tr></table>	pojemnosc	2400	lbCylindrow	5	moc	163
pojemnosc	2400							
lbCylindrow	5							
moc	163							

Wskaźnik st „widzi” obiekt silT tak:

st										
		<table><tr><td>pojemnosc</td><td>2400</td></tr><tr><td>lbCylindrow</td><td>5</td></tr><tr><td>moc</td><td>163</td></tr><tr><td>cisnienieDoładowania</td><td>0.9</td></tr></table>	pojemnosc	2400	lbCylindrow	5	moc	163	cisnienieDoładowania	0.9
pojemnosc	2400									
lbCylindrow	5									
moc	163									
cisnienieDoładowania	0.9									

Tego nie ma!

cisnienieDoładowania

W tym przypadku downcasting jest **błędny** — obiekt wskazywany przez *s* nie jest obiektem klasy *SilnikTurbo* i downcasting produkuje nieistniejące pole *cisnienieDoładowania*.

Downcasting wykonywany poprzez *dynamic_cast*

Nie do końca bezpieczne metody konwersji wskaźników:

```
Silnik * s;  
SilnikTurbo * st;  
.  
.  
.  
st = ( SilnikTurbo * )s;  
st = static_cast< SilnikTurbo * >( s );  
st = reinterpret_cast< SilnikTurbo * >( s );
```

Rzutowanie z kontrolą jego poprawności:

```
st = dynamic_cast< SilnikTurbo * >( s );  
if( st != 0 )  
    st->cisnienieDoladowania = 0.5;
```

Operator *dynamic_cast* pozwala na konwersję wskaźników z kontrolą, czy taka konwersja może być przeprowadzona. Jeżeli możliwa jest konwersja do postaci wskaźnika na klasę pochodną, operator taką konwersję realizuje. Jeżeli konwersja nie może zostać zrealizowana, rezultatem operatora jest wartość zerowa.

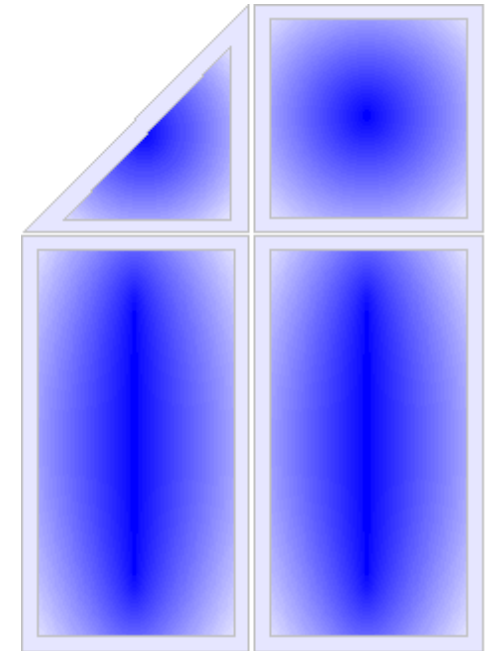
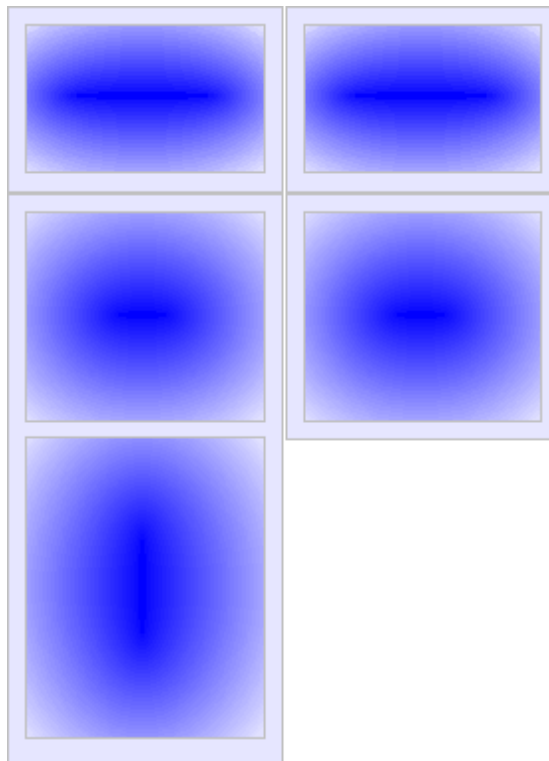
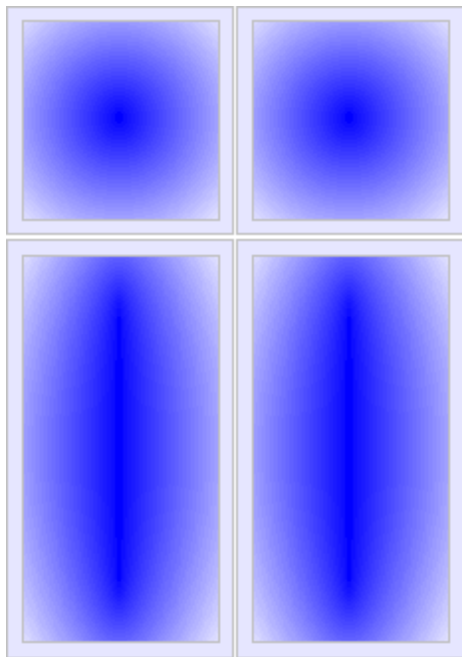
Mechanizm pozwalający na *dynamic_cast*, znany jest jako Run-Time Type Information — RTTI.

Polimorfizm w akcji

Polimorfizm w akcji

Należy napisać program wspomagający pracę technologa w firmie produkującej okna. Zadaniem programu jest:

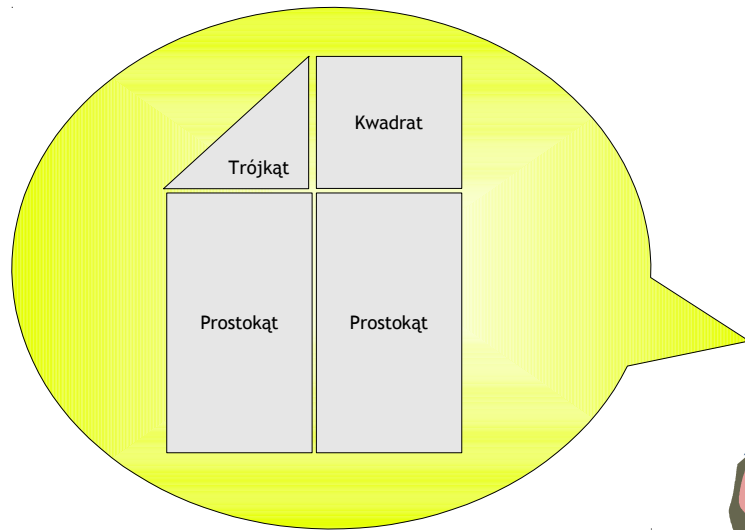
- ▶ obliczanie łącznego pola powierzchni wszystkich skrzydeł okna,
- ▶ przybliżonej, łącznej długości profili, wymaganych do produkcji każdego ze skrzydeł.



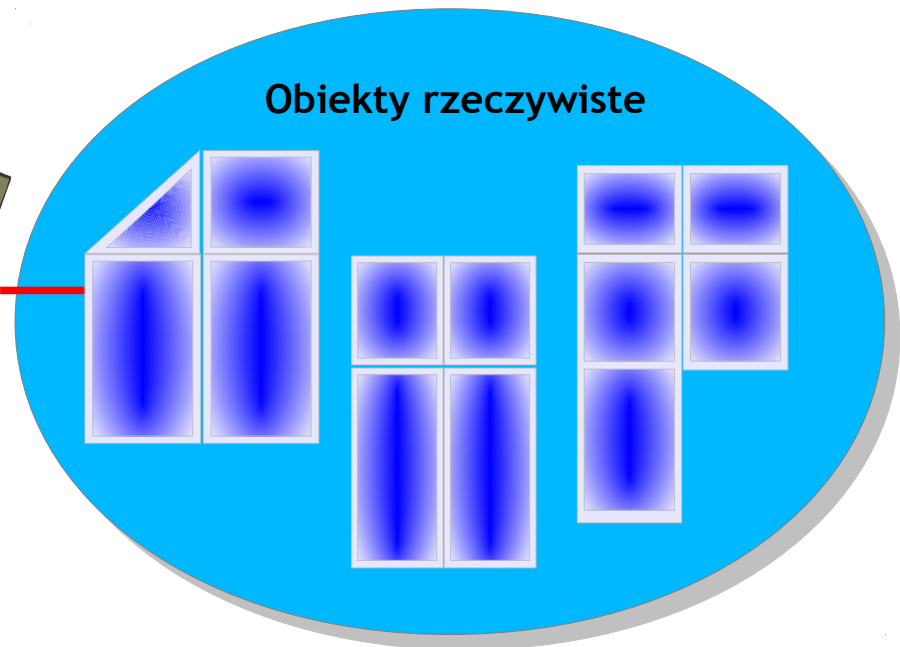
Analiza obiektowa

Stosując zasadę *abstrakcji* wyodrębniamy najistotniejsze cechy obiektów dla rozpatrywanego zagadnienia — szyby to figury geometryczne a okno to ich złożenie.

- ▶ Łączna powierzchnia okna to, w przybliżeniu, suma pól figur opisujących szyby,
- ▶ Łączna długość profili to, w przybliżeniu, suma obwodów figur opisujących szyby.



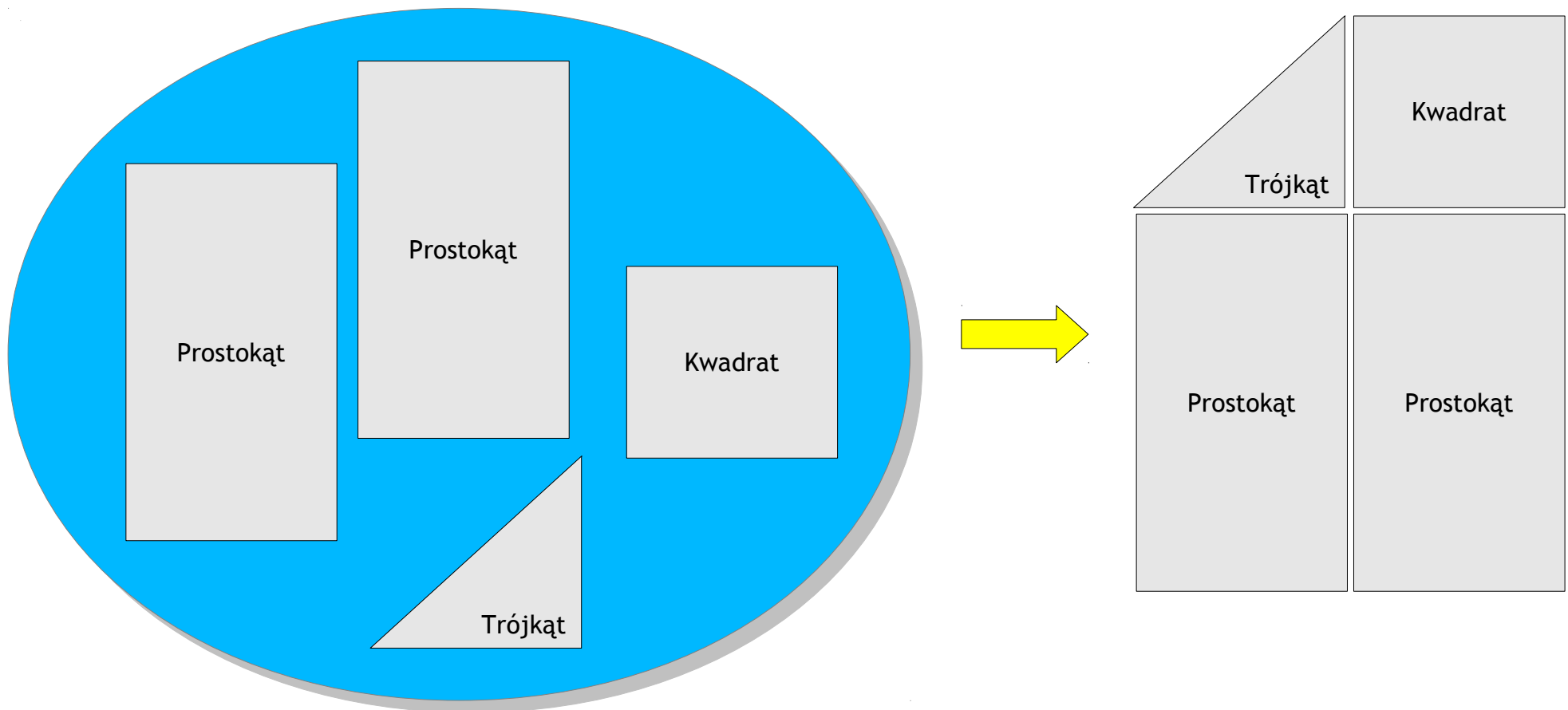
**Abstrakcyjny
model analityczny**



Analiza obiektowa, cd...

Realizacja programu sprowadza się do obliczeń pól powierzchni i obwodów obiektów, będących złożeniem elementarnych figur płaskich.

- ▶ Utworzyliśmy już klasy reprezentujące figury płaskie,
- ▶ Nie wiemy jak reprezentować ich złożenia.



Figury płaskie raz jeszcze — budujemy hierarchię klas

Rozpoczynamy od utworzenia nieco dziwnej klasy — reprezentującej *jakąś abstrakcyjną figurę geometryczną*. Wyposażamy ją w funkcje obliczania pola i obwodu.

```
class Figure
{
    public :
        Figure() {}
        double area() const { return 0; }
        double circumference() const { return 0; }
        const char * getName() const { return "Figura"; }
};
```

- ▶ Klasa **Figure** służyć będzie jak klasa bazowa dla specjalizowanych klas pochodnych, reprezentujących *konkretne* figury geometryczne.
- ▶ Jej istotą jest stwierdzenie, że każda figura geometryczna powinna umieć *wyznaczać swoje pole i obwód*, w charakterystyczny dla siebie sposób.
- ▶ Zatem każda z klas pochodnych, *powinna redefiniować funkcje area i circumference*, w odpowiedni dla tych klas sposób.
- ▶ Dodatkowo klasa **Figure** i jej pochodne posiadać będą funkcję **getName()**, jej rezultatem będzie nazwa klasy.

Klasa *Square* jako pochodna klasy *Figure*

Klasa reprezentująca kwadrat będzie teraz klasą pochodną klasy *Figure*:

```
class Square: public Figure
{
    public :
        Square( double startSide = 0 );
        void    setSide( double newSide );
        double  getSide();

        double  area() const;
        double  circumference() const;
        const char * getName() const;

    private:
        double  side;
};
```

Redefinicja funkcji *area* i *circumference* i *getName* dla kwadratu:

```
double Square::area() const { return side * side; }
double Square::circumference() const { return 4 * side; }
const char * Square::getName() const { return "Kwadrat"; }
```

Klasa *Rectangle* jako pochodna klasy *Figure*

Klasa reprezentująca prostokąt będzie teraz klasą pochodną klasy *Figure*:

```
class Rectangle: public Figure
{
    public :
        Rectangle( double startWidth = 0, double startHeight = 0 );
        void setWidth( double newWidth );
        void setHeight( double newHeight );
        double getWidth() const;
        double getHeight() const;

        double area() const;
        double circumference() const;
        const char * getName() const;

    private:
        double width, height;
};
```

Redefinicja funkcji *area* i *circumference* i *getName* dla prostokąta:

```
double Rectangle::area() const { return width * height; }
double Rectangle::circumference() const { return 2 * width + 2 * height; }
const char * Rectangle::getName() const { return "Prostokat"; }
```

Klasa *Triangle* jako pochodna klasy *Figure*

Klasa reprezentująca trójkąt będzie teraz klasą pochodną klasy *Figure*:

```
class Triangle: public Figure
{
    public :
        Triangle( double startBase = 0, double startHeight = 0 );
        void setBase( double newBase );
        void setHeight( double newHeight );
        double getBase() const;
        double getHeight() const;

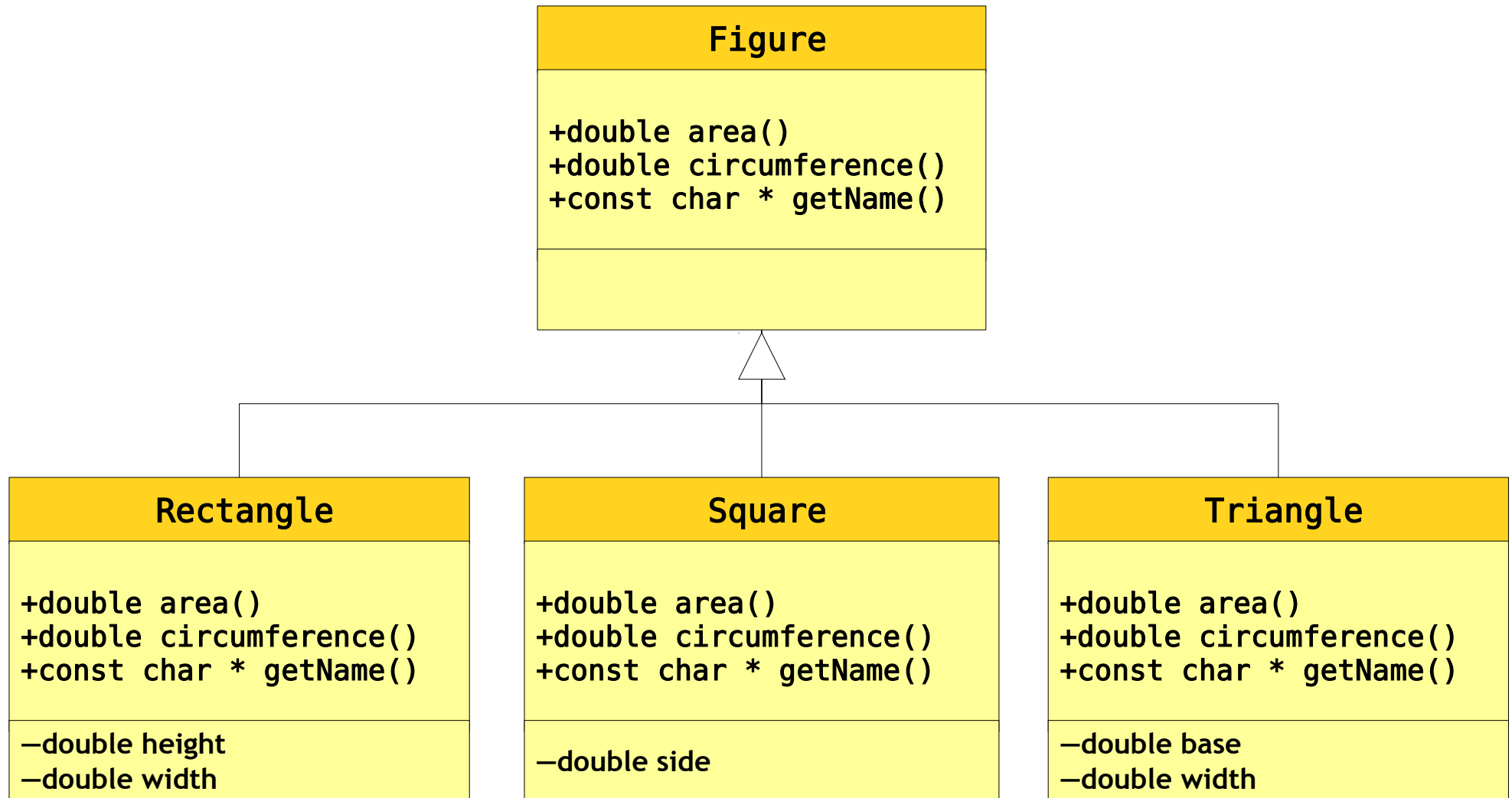
        double area() const;
        double circumference() const;
        const char * getName() const;
    private:
        double base, height;
};
```

Uwaga, zakładamy dla uproszczenia, że trójkątne
szyby będą trójkątami prostokątnymi.

Redefinicja funkcji *area* i *circumference* i *getName* dla trójkąta:

```
double Triangle::area() const { return 0.5 * base * height; }
double Triangle::circumference() const {
    return sqrt( base * base + height * height ) + base + height;
}
const char * Triangle::getName() const { return "Trojkat"; }
```

Hierarchia klas figur płaskich



Krok w stronę polimorfizmu – upcasting z wykorzystaniem wskaźnika

Założmy, że zdefiniowano wskaźnik do klasy bazowej **Figure**:

```
Figure * fp;
```

Założmy, że zdefiniowano obiekty klas pochodnych:

```
Square    s( 10 );  
Rectangle r( 10, 20 );  
Triangle  t( 10,10 );
```

Wiemy już, że można przypisać do wskaźnika **fp** wskazanie na obiekt klasy pochodnej:

```
fp = &s;    // OK  
fp = &r;    // OK  
fp = &t;    // OK
```

Polimorfizm w akcji...?

```
Figure * fp;
```

```
Square    s( 10 );
```

```
Rectangle r( 10, 20 );
```

```
Triangle  t( 10,10 );
```

```
fp = &s;
```

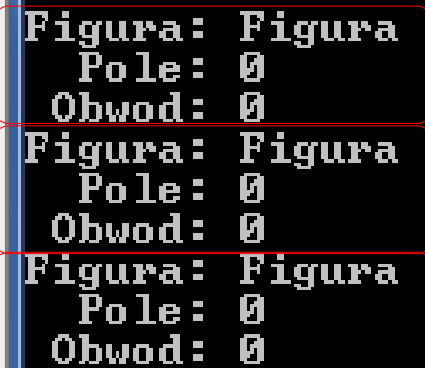
```
cout << endl << "Figura: " << fp->getName();  
cout << endl << "  Pole: " << fp->area();  
cout << endl << "  Obwod: " << fp->circumference();
```

```
fp = &r;
```

```
cout << endl << "Figura: " << fp->getName();  
cout << endl << "  Pole: " << fp->area();  
cout << endl << "  Obwod: " << fp->circumference();
```

```
fp = &t;
```

```
cout << endl << "Figura: " << fp->getName();  
cout << endl << "  Pole: " << fp->area();  
cout << endl << "  Obwod: " << fp->circumference();
```



```
Figura: Figura  
Pole: 0  
Obwod: 0  
Figura: Figura  
Pole: 0  
Obwod: 0  
Figura: Figura  
Pole: 0  
Obwod: 0
```

- ▶ Ale to nie działa! Obiekty zachowują się tak, jakby były obiektami klasy Figure!
- ▶ Zapomnieliśmy o funkcjach wirtualnych...!

Zapomnieliśmy o funkcjach wirtualnych!

Jest:

```
class Figure
{
    public :
        Figure() {}
        double area() const { return 0; }
        double circumference() const { return 0; }
        const char * getName() const { return "Figura"; }
};
```

Powinno być:

```
class Figure
{
    public :
        Figure() {}
        virtual double area() const { return 0; }
        virtual double circumference() const { return 0; }
        virtual const char * getName() const { return "Figura"; }
};
```

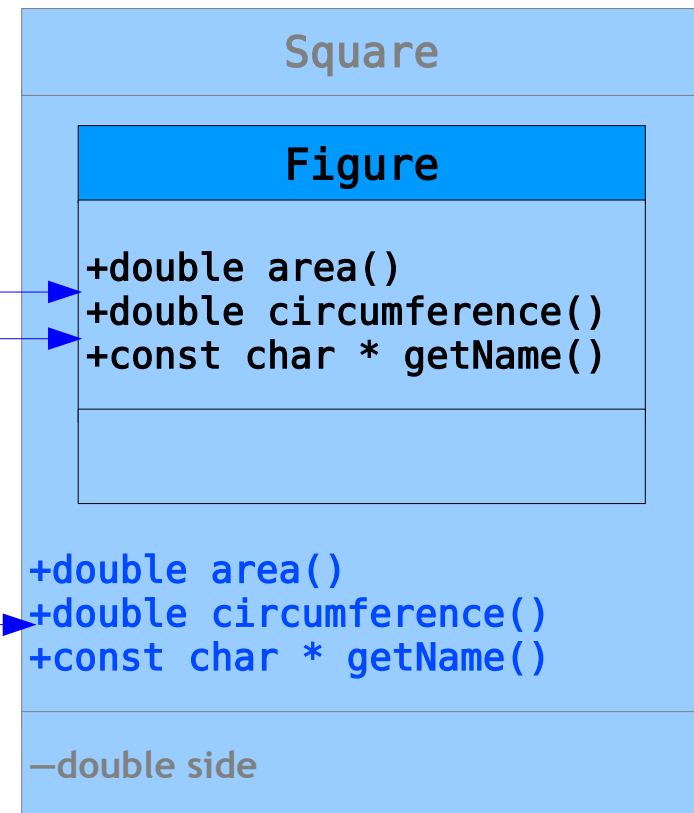
- ▶ Nie ma polimorfizmu bez metod wirtualnych w klasach tworzących hierarchię dziedziczenia.

Wiązanie statyczne i funkcje niewirtualne – raz jeszcze

- ▶ O tym która funkcja jest wywoływana decyduje kompilator na etapie kompilacji.
- ▶ Wywołanie ma taką samą postać jak wywołanie klasycznych funkcji w języku C.
- ▶ O tym która funkcja zostanie wywołana decyduje *typ występujący w deklaracji zmiennej wskaźnikowej*.

```
Figure * fp;  
Square s( 10 );  
  
fp = &s;  
cout << fp->area() << endl;  
cout << fp->circumference() << endl;
```

Mimo iż klasa **Square** przeddefiniowała obie funkcje i posiada ich własne wersje, nie zostaną one wywołane.



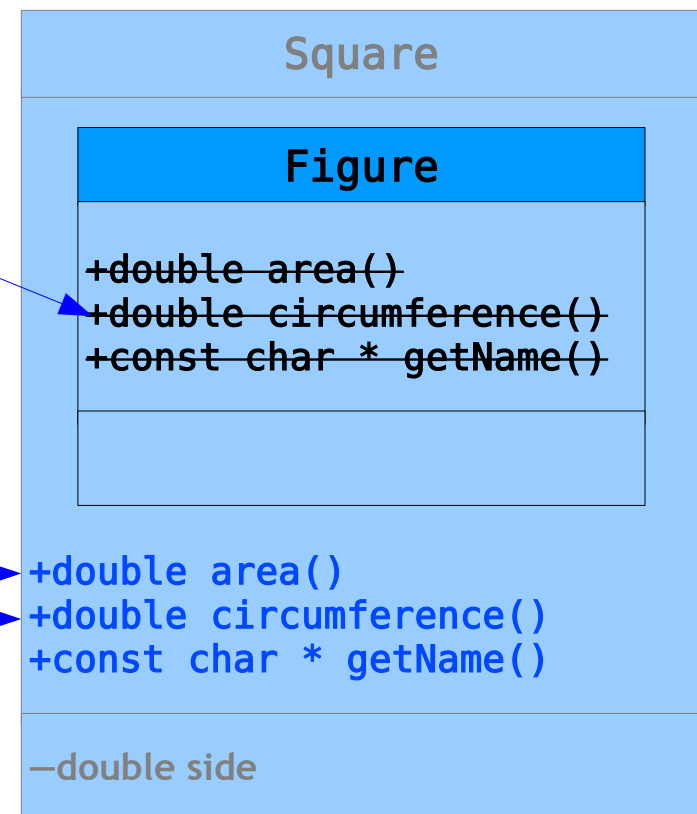
Wiązanie dynamiczne i funkcje wirtualne – raz jeszcze

- ▶ O tym która funkcja jest wywoływana decyduje **typ obiektu wskazywanego**.
- ▶ Funkcja wiązana dynamicznie musi być zadeklarowana jako *wirtualna*.
- ▶ Wystarczy, że słowo kluczowe **virtual** wystąpi w deklaracji klasy bazowej.

Te funkcje zostały „nadpisane” w klasie pochodnej.

```
Figure * fp;  
Square s( 10 );  
  
fp = &s;  
cout << fp->area() << endl;  
cout << fp->circumference() << endl;
```

Klasa **Square** przeddefiniowała obie funkcje wirtualne.
To one właśnie zostaną wywołane a nie funkcje z klasy bazowej.



Jeszcze raz z funkcjami wirtualnymi...

```
Figure * fp;
```

```
Square    s( 10 );
```

```
Rectangle r( 10, 20 );
```

```
Triangle  t( 10,10 );
```

```
fp = &s;
```

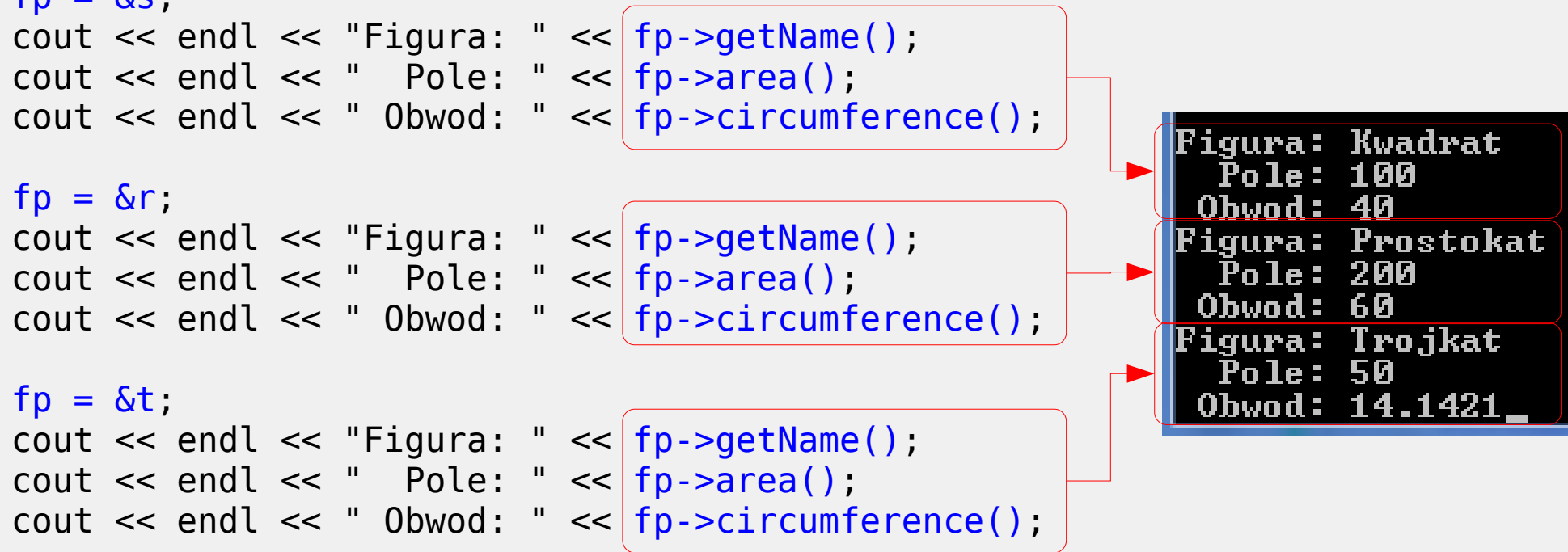
```
cout << endl << "Figura: " << fp->getName();  
cout << endl << "  Pole: " << fp->area();  
cout << endl << "  Obwod: " << fp->circumference();
```

```
fp = &r;
```

```
cout << endl << "Figura: " << fp->getName();  
cout << endl << "  Pole: " << fp->area();  
cout << endl << "  Obwod: " << fp->circumference();
```

```
fp = &t;
```

```
cout << endl << "Figura: " << fp->getName();  
cout << endl << "  Pole: " << fp->area();  
cout << endl << "  Obwod: " << fp->circumference();
```

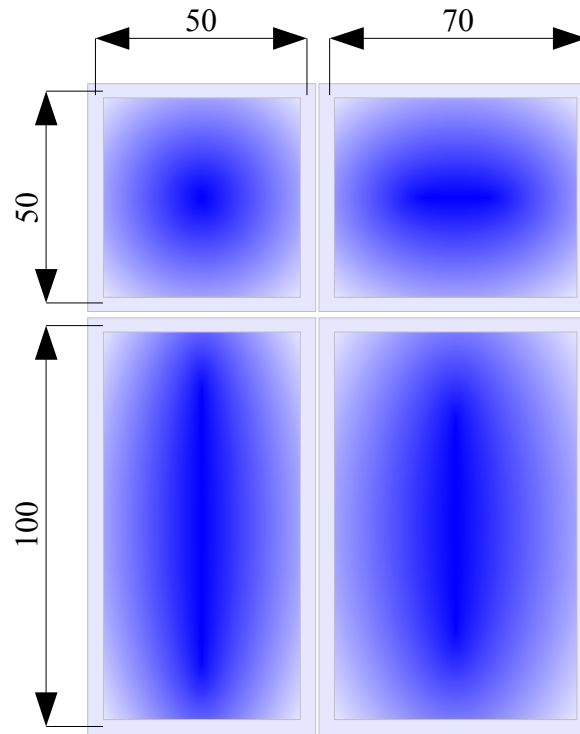


```
Figura: Kwadrat  
Pole: 100  
Obwod: 40  
Figura: Prostokat  
Pole: 200  
Obwod: 60  
Figura: Trojkat  
Pole: 50  
Obwod: 14.1421
```

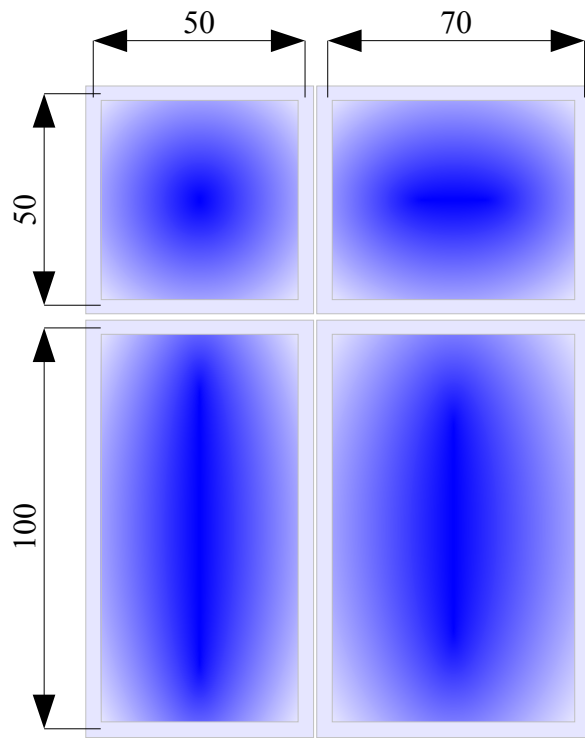
- **Polimorfizm** = dziedziczenie, redefinicja metod wirtualnych, wskaźniki (referencje) i upcasting.

Jak to wszystko wykorzystać w programie „okiennym”?

- ▶ Zadany jest układ okna oraz wymiary jego ościeżnic. Jedna ościeżnica jest kwadratowa (bok 50cm), pozostałe trzy prostokątne (70x50 cm, 50x100 cm i 70x100 cm).
- ▶ Należy wyznaczyć łączną powierzchnię szyb i długość profili.

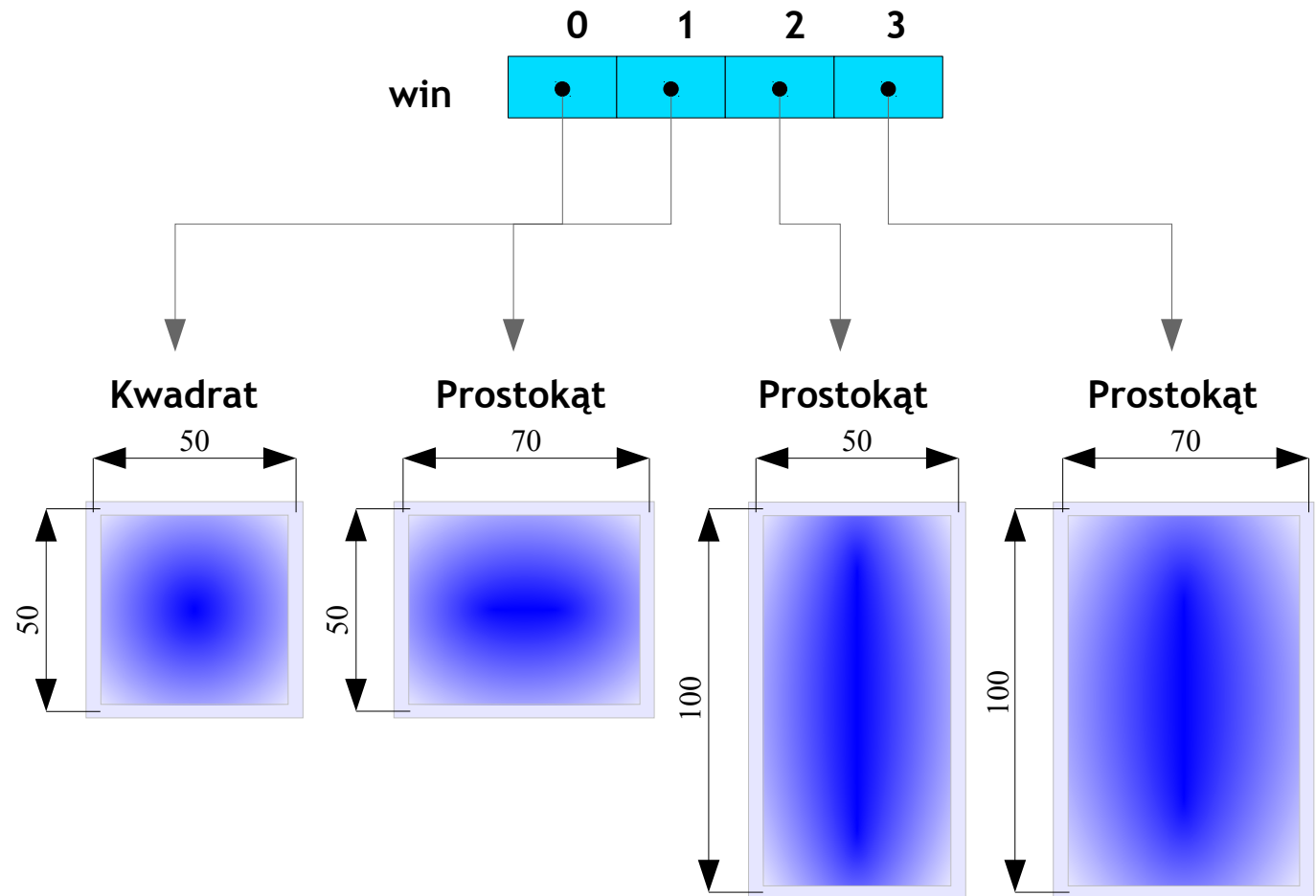


Jak reprezentować informacje o oknie?

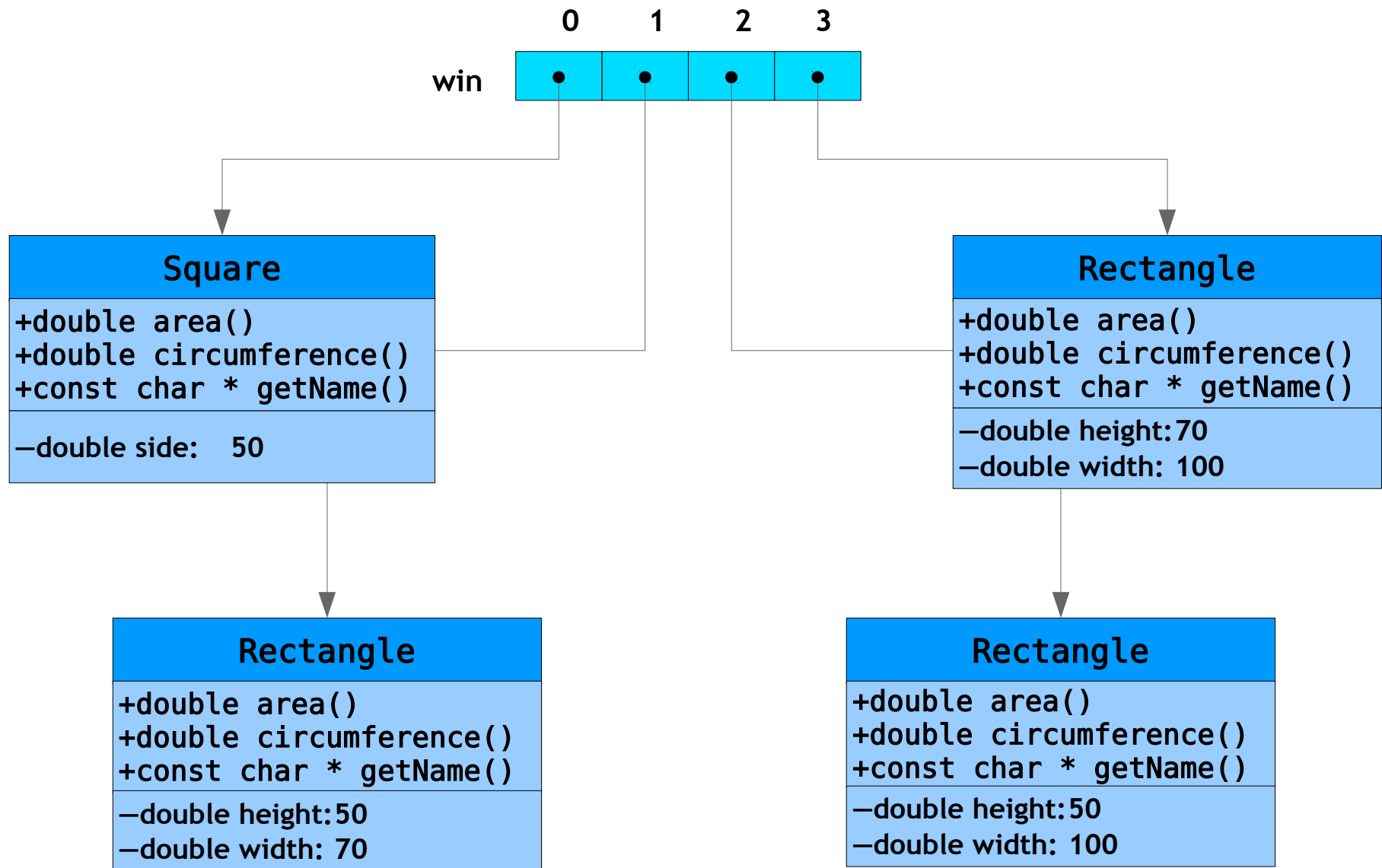


```
const int itemNo = 4;
```

```
Figure * win[ itemNo ];
```



Jak to wygląda w pamięci operacyjnej?



Definiowanie elementów okna, obliczenia, sprzątanie

```
// Określenie liczby elementów
const int itemNo = 4;

// Tablica wskaźników na elementy okna
Figure * win[ itemNo ];

// Przydział pamięci dla elementów okna
win[ 0 ] = new Square( 50 );
win[ 1 ] = new Rectangle( 50, 70 );
win[ 2 ] = new Rectangle( 50, 100 );
win[ 3 ] = new Rectangle( 70, 100 );

// Oblicz co trzeba i wyprowadz do strumienia wyjściowego
calcAndShowWinInfo( win, 4 );

// Zwolnij pamięć przydzieloną elementom okna
for( int i = 0; i < itemNo; delete win[ i++] )
    ;
```


Funkcja calcAndShowWinInfo

```
void calcAndShowWinInfo( Figure * window[], int numoSash )
{
    // Tutaj łączna powierzchnia szyb i długość profili
    double totalArea = 0, totalCircum = 0;

    // Zliczanie powierzchni i długości
    for( int i = 0; i < numoSash; i++ )
    {
        totalArea    += window[ i ]->area();
        totalCircum += window[ i ]->circumference();
    }

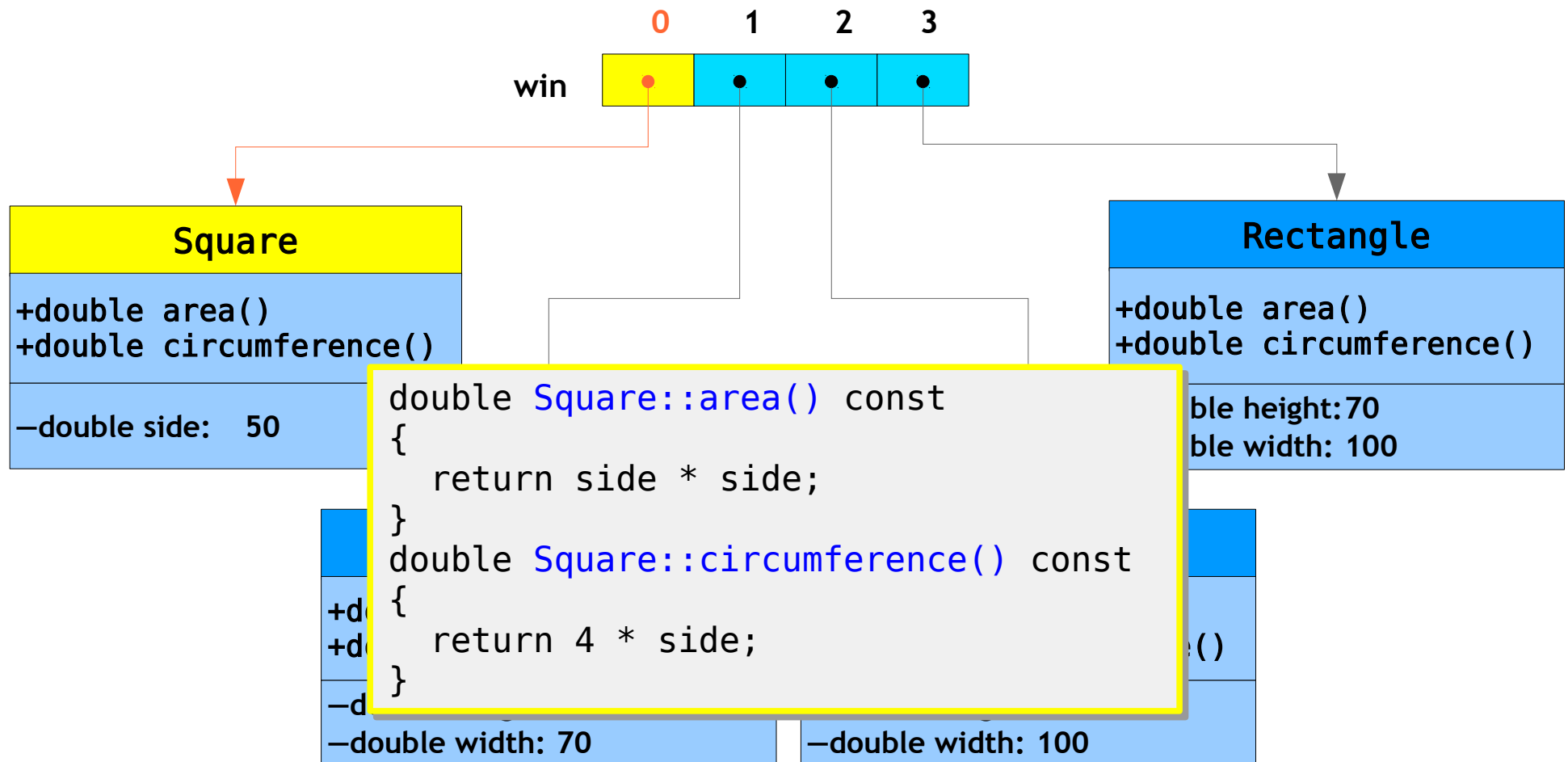
    // Wyprowadzanie wyników obliczeń
    cout << "Powierzchnia szyb : " << totalArea << endl;
    cout << "Długosc profili   : " << totalCircum << endl;
    cout << endl;
}
```

```
Powierzchnia szyb : 19400
Dlugosc profili   : 1120
```

Iteracja krok po kroku

```
for( int i = 0; i < numOfSash; i++ )  
{  
    totalArea    += window[ i ]->area();  
    totalCircum += window[ i ]->circumference();  
}
```

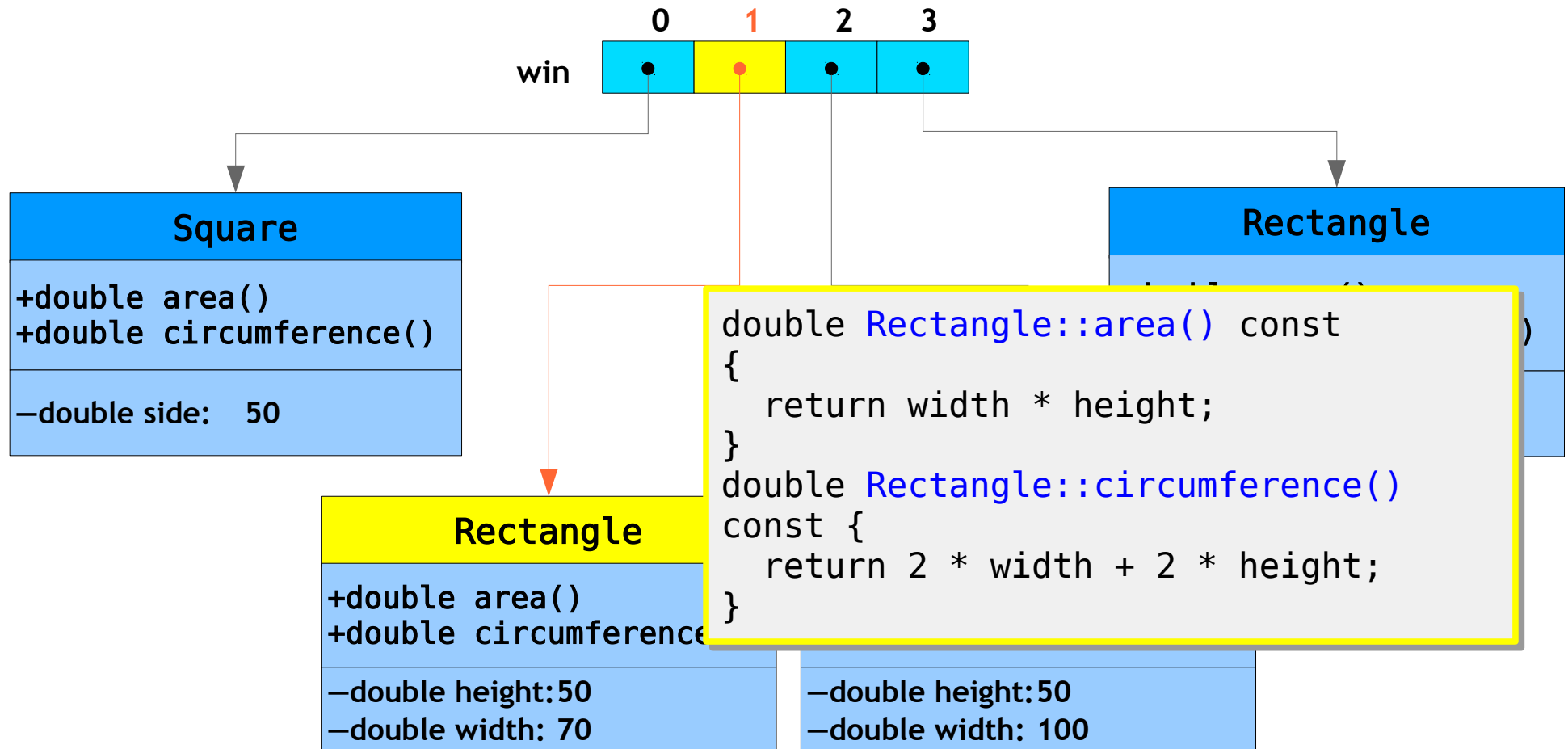
i 0



Iteracja krok po kroku

```
for( int i = 0; i < numOfSash; i++ )  
{  
    totalArea    += window[ i ]->area();  
    totalCircum  += window[ i ]->circumference();  
}
```

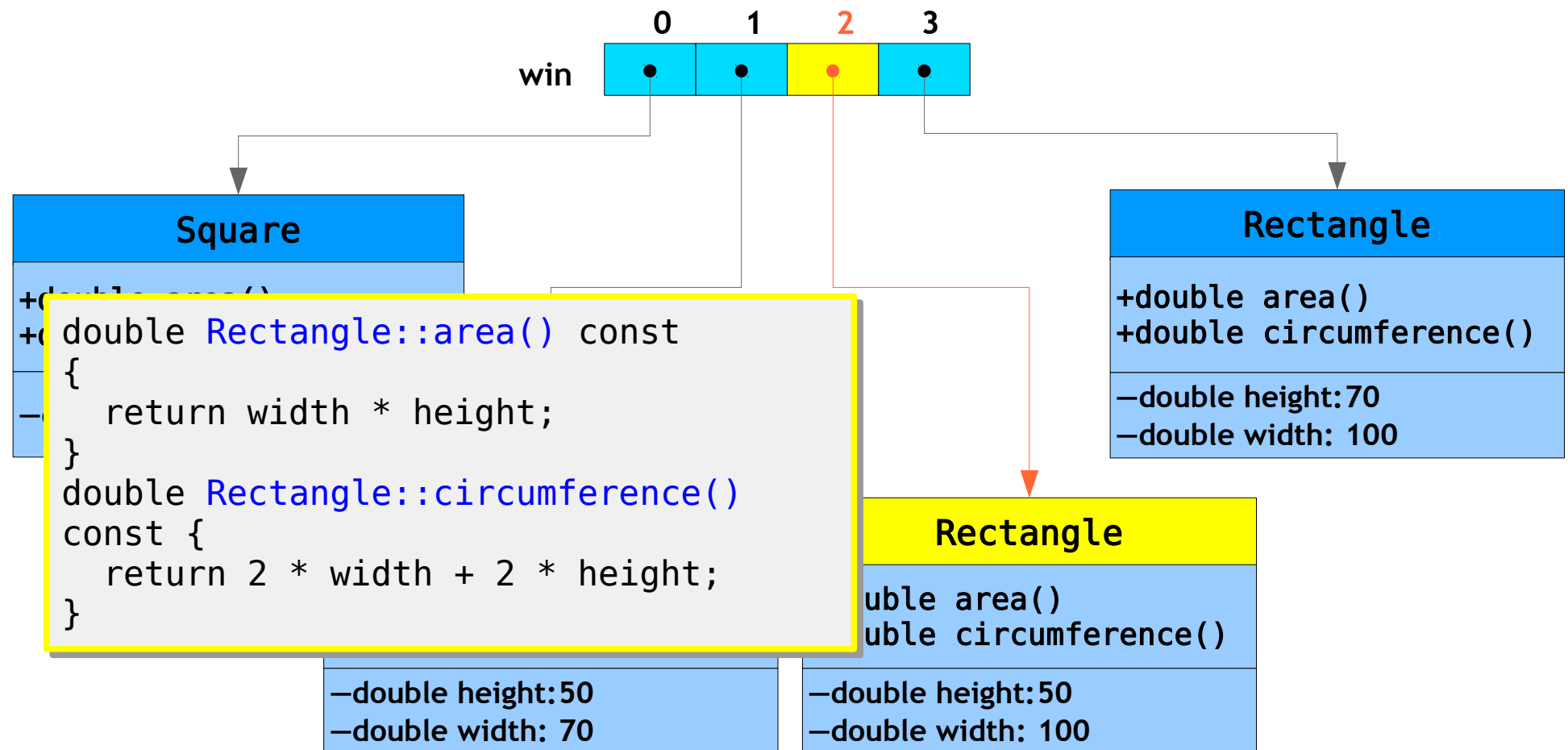
i 1



Iteracja krok po kroku

```
for( int i = 0; i < numOfSash; i++ )
{
    totalArea    += window[ i ]->area();
    totalCircum  += window[ i ]->circumference();
}
```

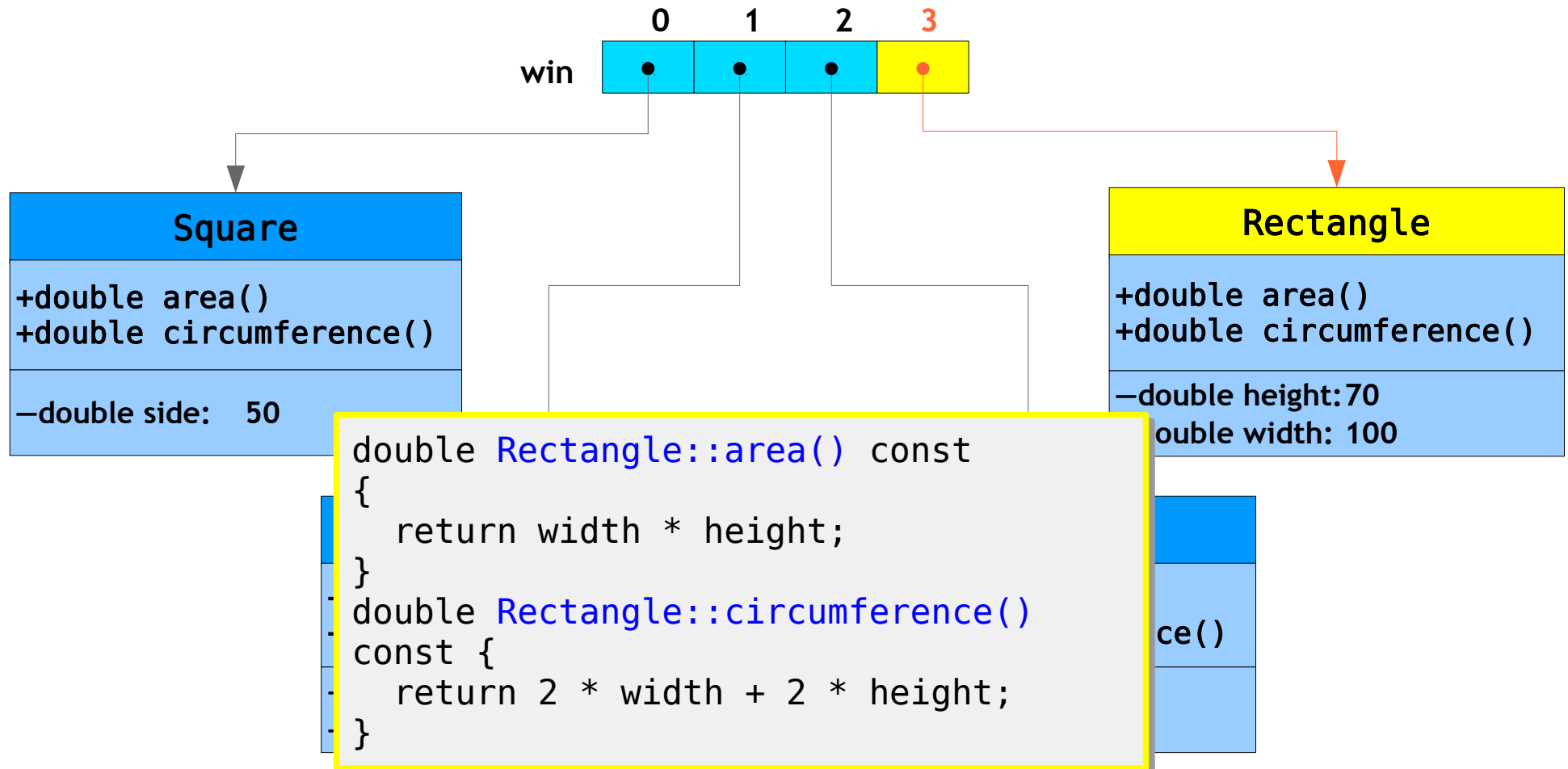
i 2



Iteracja krok po kroku

```
for( int i = 0; i < numOfSash; i++ )  
{  
    totalArea    += window[ i ]->area();  
    totalCircum  += window[ i ]->circumference();  
}
```

i 3



Klasy abstrakcyjne C++

Figury płaskie raz jeszcze – klasa abstrakcyjna

Klasa bazowa, określająca protokół rozmowy z obiektami klas pochodnych, powinna być klasą abstrakcyjną:

```
class Figure
{
    public :
        Figure() {}

        virtual double area() const = 0;
        virtual double circumference() const = 0;

        const char * getName() const { return "Figura"; }
};
```

Funkcje abstrakcyjne (ang. *pure virtual functions*).
Muszą zostać zdefiniowane w każdej nieabstrakcyjnej klasie pochodnej.

- ▶ Służy ona jak wzorcowa klasa bazowa dla specjalizowanych klas pochodnych, reprezentujących *konkretne* figury geometryczne.
- ▶ Każda pochodna klasa nieabstrakcyjna musi zdefiniować własną wersję *czystej funkcji wirtualnej*.

Figury płaskie raz jeszcze — klasa abstrakcyjna, cd. ...

```
class Figure
{
    public :
        Figure() {}
        virtual double area() const = 0;
        virtual double circumference() const = 0;
        const char * getName() const { return "Figura"; }
};
```

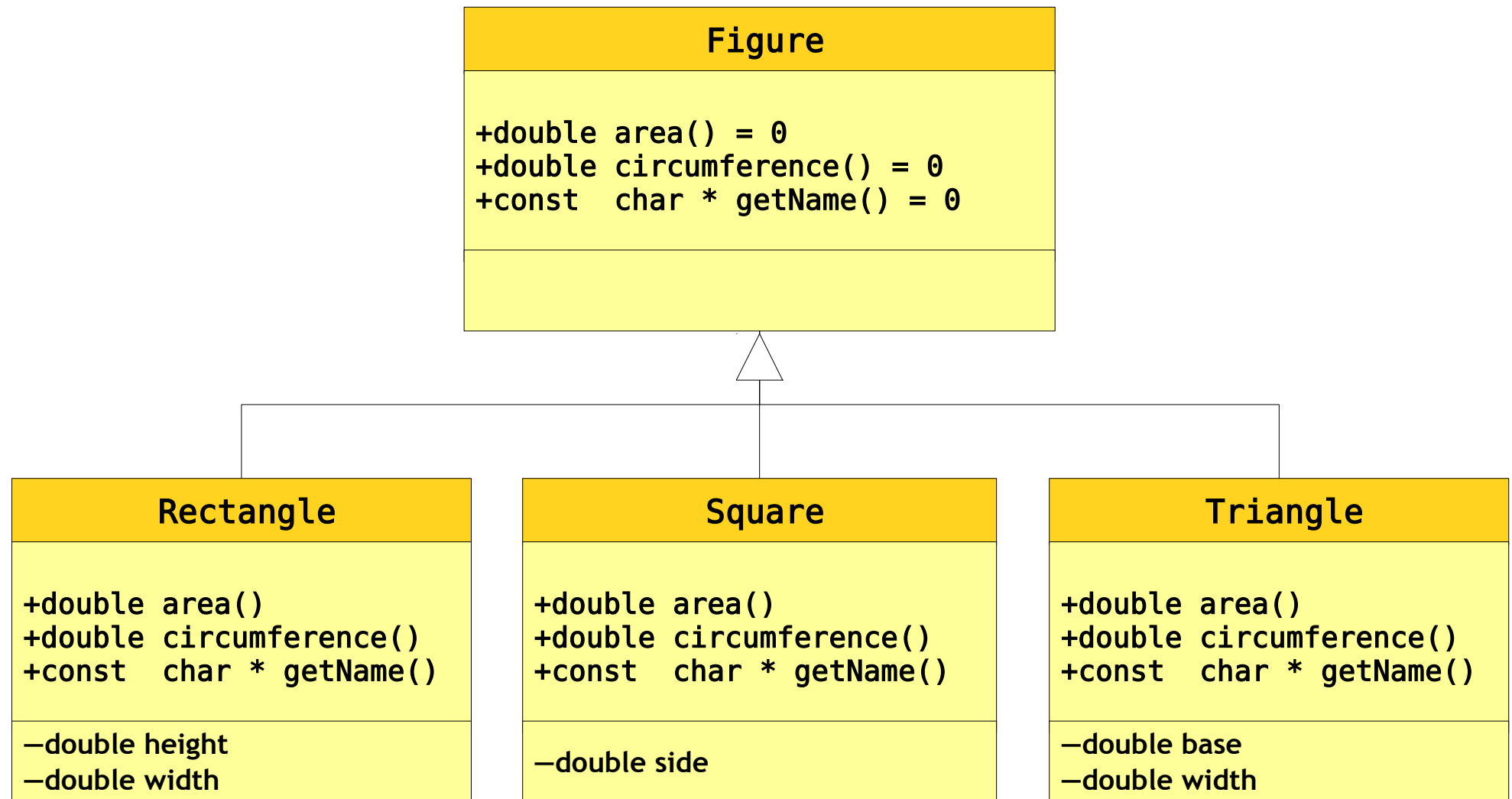
- ▶ Nie można zdefiniować obiektów abstrakcyjnej klasy **Figure**.

```
Figure f; // Nie definiujemy obiektów klas abstrakcyjnych
```

- ▶ Wolno jednak posługiwać się wskaźnikami i referencjami do klasy **Figure**.

```
Figure * f; // OK
. . .
void showFigureInfo( Figure & fp ) // OK
{
    . . .
}
```


Klasy pochodne muszą dostarczyć definicji funkcji abstrakcyjnych



Klasy pochodne muszą dostarczyć definicji funkcji abstrakcyjnych

```
class Figure
{
    public :
        . . .
        virtual double area() const = 0;
        virtual double circumference() const = 0;
        . . .
};
```

```
class Square: public Figure
{
    public :
        . . .
        double area() const;
        double circumference() const;
        . . .
};
```

Deklaracje funkcji

```
double Square::area() const
{
    return side * side;
}
double Square::circumference() const
{
    return 4 * side;
}
```

Deklaracje funkcji

Klasy abstrakcyjne Java

Figury płaskie raz jeszcze – klasa abstrakcyjna

```
abstract class Figure
{
    . . .
}
```

```
Figure = new Figure(); // Nie tworzymy obiektów klas abstrakcyjnych
```

Klasa abstrakcyjna:

- ▶ może posiadać implementację wybranych metod.
- ▶ może posiadać metody abstrakcyjne.
- ▶ może posiadać pola.

Nie tworzymy obiektów klas abstrakcyjnych.

Figury płaskie raz jeszcze – klasa abstrakcyjna

```
abstract class Figure
{
    public Figure() {}

    public abstract double area();
    public abstract double circumference();

    public String getName()
    {
        return "Figura";
    }

    static int lbFigur;
}
```

Metody abstrakcyjne. Muszą zostać zdefiniowane w każdej nieabstrakcyjnej klasie pochodnej.

Zwykła metoda, w klasach abstrakcyjnych mogą występować zarówno metody abstrakcyjne jak i zwykłe.

- ▶ Metody abstrakcyjne deklarujemy z wykorzystaniem słowa kluczowego **abstract**.
- ▶ Zapisujemy tylko *sygnaturę metody*, nie definiujemy jej ciała (nawet pustego).

Klasa abstrakcyjna pozwala na częściowe zdefiniowanie czynności realizowanych przez klasę, pozwalając na późniejsze uszczegółowienie czynności reprezentowanych przez metody abstrakcyjne.

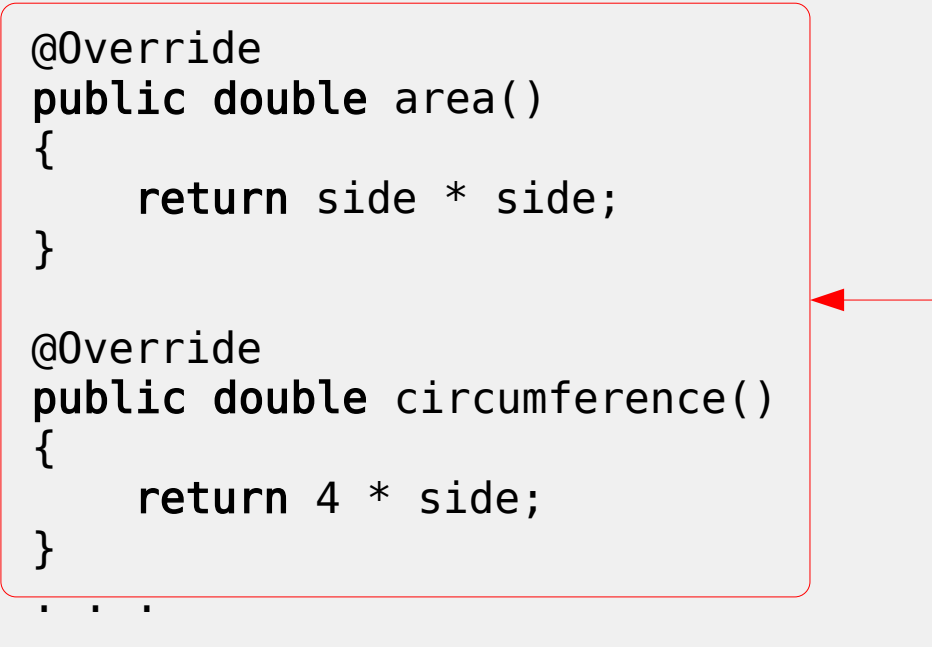
Implementacja metody abstrakcyjnej z klasy pochodnej

```
class Square extends Figure
{
    public Square()
    {
        super();
    }

    @Override
    public double area()
    {
        return side * side;
    }

    @Override
    public double circumference()
    {
        return 4 * side;
    }

    . . .
}
```



Aby można było tworzyć obiekty klasy pochodnej, każda metoda abstrakcyjna musi być w klasie pochodnej zaimplementowana. `@Override` – adnotacja sygnalizująca intencję programisty przeddefiniowania metody z klasy bazowej, pozwala na sprawdzenia w czasie kompilacji czy programista nie popełnił błędu,

Rola klas abstrakcyjnych – częściowa definicja działania

```
abstract class Party
{
    public enum State { NONE, BEFORE, IN_PROGRESS, AFTER };
    public Party() {
        state = State.NONE;
    }
    public void beforeParty() {
        state = State.BEFORE;
    }

    public abstract void makeParty();

    public void afterParty() {
        state = State.AFTER;
    }

    public void doParty() {
        beforeParty();
        makeParty();
        afterParty();
    }

    public State state;
}
```

Działanie tej metody zostanie uszczegółowione
w klasie pochodnej.

Ogólny przepis na realizację zachowania obiektu
– wymaga doprecyzowania *makeParty()*

Dziedziczenie z klas abstrakcyjnych – doprecyzowanie działania

```
class GardenParty extends Party
{
    public GardenParty()
    {
        super();
    }

    @Override
    public void makeParty()
    {
        state = State.IN_PROGRESS;

        prepareGrill();
        for( ; ; )
            openBeer();
    }

    public void prepareGrill() {}
    public void openBeer() {}
}
```

Doprecyzowanie metody abstrakcyjnej

Klasy abstrakcyjne C#

Definicja klasy abstrakcyjnej w C# jest zbliżona do znanej z języka Java

Metody abstrakcyjne. Muszą zostać zdefiniowane w każdej nieabstrakcyjnej klasie pochodnej.

```
abstract class Figure
{
    public Figure() {}

    public abstract double area();
    public abstract double circumference();

    public String getName()
    {
        return "Figura";
    }

    static int lbFigur;
}
```

Mogą występować pola statyczne i niestacyjne

Zwykła metoda, w klasach abstrakcyjnych mogą występować zarówno metody abstrakcyjne jak i zwykłe.

Definicja klasy pochodnej – różnice w stosunku do języka Java

```
class Square : Figure
{
    public Square() : base()
    {
    }

    public override double area()
    {
        return side * side;
    }

    public override double circumference()
    {
        return 4 * side;
    }

    . . .
}
```

Przypomnienie – dziedziczenie przypomina bardziej C++ niż Javę

Każda metoda abstrakcyjna musi być w klasie pochodnej zaimplementowana z wykorzystaniem słowa kluczowego *override*.

Rola klas abstrakcyjnych – częściowa definicja działania

```
abstract class Party
{
    public enum State { NONE, BEFORE, IN_PROGRESS, AFTER };
    public Party() {
        state = State.NONE;
    }
    public void beforeParty() {
        state = State.BEFORE;
    }

    public abstract void makeParty();

    public void afterParty() {
        state = State.AFTER;
    }

    public void doParty() {
        beforeParty();
        makeParty();
        afterParty();
    }

    public State state;
}
```

Działanie tej metody zostanie uszczegółowione
w klasie pochodnej.

Ogólny przepis na realizację zachowania obiektu
– wymaga doprecyzowania *makeParty()*

Interfejsy

Po co?

Rola interfejsów w obiektowości

- ▶ Klasa reprezentuje szablon, według którego tworzony będzie obiekt.
- ▶ Klasa definiuje atrybuty i/lub metody w które taki obiekt będzie wyposażony.
- ▶ Nawet klasa abstrakcyjna docelowo służy do definiowania obiektów.

Klasy są narzędziem modelowania i definiowania *obiektów* w systemie. Czasem potrzebujemy modelować *potencjalne zachowania* obiektów, często w oderwaniu od nich samych. Do modelowania zachowań wykorzystujemy **interfejsy**.

- ▶ Interfejsy nie są po to, by definiować obiekty.
- ▶ Interfejsy są po to, by definiować zestaw zachowań.
- ▶ Obiekt pewnej klasy może implementować interfejs — realizować zachowania określone przez dany interfejs.
- ▶ Interfejsy zazwyczaj nie zawierają pól.

Rola interfejsów w obiektowości

- ▶ Gdy pewna klasa wykorzystuje interfejs, to oznacza, że gwarantuje obsługę metody zadeklarowanej w tym interfejsie.
- ▶ Metody interfejsu deklaruje się bez żadnej treści, konkretna definicja metody w interfejsie nie jest dozwolona.
- ▶ Interfejs przypomina nieco klasę abstrakcyjną, posiadającą wszystkie metody abstrakcyjne.
- ▶ Klasy abstrakcyjne bywają wykorzystywane – np. w C++ – do realizacji interfejsów nie występujących jawnie w języku.
- ▶ Ważna różnica – klasa pochodna abstrakcyjnej klasy bazowej może zmieniać widoczność metod odziedziczonych, jeśli jakaś klasa implementuje interfejs, to wówczas musi udostępniać wszystkie metody zdefiniowane w tym interfejsie.

- ▶ W języku C++ nie występuje osobna notacja dla interfejsów.
- ▶ Wykorzystanie koncepcji interfejsów wymaga zastosowania klas zawierających funkcje abstrakcyjne.
- ▶ Występujące w C++ dziedziczenie wielobazowe pozwala na swobodne budowanie klas posiadających wiele bezpośrednich klas bazowych.
- ▶ Interfejsy występują zazwyczaj w językach nie oferujących dziedziczenia wielobazowego.
- ▶ Jawnie wodrębnione interfejsy występują w językach Java, C#, PHP.

Niektóre kompilatory (np. VC++) wprowadzają własne rozszerzenia do C++ oferujące mechanizm podobny do interfejsów z języka Java i C#.

Interfejsy Java

Przykłady interfejsu

```
interface BasicCalculations
{
    double area();
    double circumference();
}
```

Domyślnie publicznie
i abstrakcyjne

- ▶ Wszystkie metody interfejsu są domyślnie *publiczne* i *abstrakcyjne*.
- ▶ Metody interfejsów nie mogą być statyczne (*static*) ani zakończone (*final*).

```
interface GearBoxActions
{
    int gearUp();
    int gearDown();

    int numberOfGears = 6;
}
```

Domyślnie publicznie, statyczne i zakończone.

- ▶ Interfejs może zawierać pola, są one wtedy domyślnie *publiczne*, *statyczne* i *finalne*.
- ▶ Wszystkie klasy, które kiedyś zaimplementują interfejs, będą miały zawsze bezpośredni dostęp do tych samych, stałych wartości pól.

Przykłady wykorzystania interfejsu

class Driver implements GearBoxActions

{

public Driver() {}

@Override

public int gearUp()

{

if(currentGear < numberOfGears)

++currentGear;

return currentGear;

}

@Override

public int gearDown()

{

if(currentGear > 0)

--currentGear;

return currentGear;

}

public int currentGear = 0;

}

Klasa *implementuje* dany
interfejs

Implementacje metod interfejsu
muszą być publiczne

Implementacja wielu interfejsów

- ▶ Interfejs definiujący akcje kontrolujące skrzynię biegów.

```
interface GearBoxActions
{
    int gearUp();
    int gearDown();
}
```

- ▶ Interfejs definiujący akcje kontrolujące prędkość (hamulec, przyspieszanie).

```
interface SpeedControl
{
    void pressBreak();
    void releaseBreak();

    void accelerate();
    void slowDown();
}
```

Implementacja wielu interfejsów

```
class Driver implements GearBoxActions, SpeedControl
```

```
{
```

```
    public Driver() {}
```

```
    @Override
```

```
    public int gearUp() { ... }
```

```
    @Override
```

```
    public int gearDown() { ... }
```

```
    @Override
```

```
    public void pressBreak() { ... }
```

```
    @Override
```

```
    public void releaseBreak() { ... }
```

```
    @Override
```

```
    public void accelerate() { ... }
```

```
    @Override
```

```
    public void slowDown() { ... }
```

```
}
```

Implementacja dwóch interfejsów

Implementacja interfejsów, zasady ogólne

Nieabstrakcyjna klasa implementująca interfejs:

- ▶ Musi posiadać implementację wszystkich metod interfejsu,
- ▶ Implementowane metody muszą zachować sygnaturę metod z interfejsu,
- ▶ Implementowane metody muszą być zdefiniowane jako publiczne.
- ▶ Klasa może implementować więcej niż jeden interfejs.
- ▶ Nie można definiować konstruktora i destruktora wewnątrz interfejsu.
- ▶ Interfejsy zwyczajowo definiuje rozpoczynając ich nazwę od litery „I”.

```
interface IGearBoxActions
{
    . . .
}

interface ISpeedControl
{
    . . .
}
```

Rozszerzanie interfejsów

```
interface SpeedControl
{
    void pressBreak();
    void releaseBreak();

    void accelerate();
    void slowDown();
}

interface BoostedSpeedControl extends SpeedControl
{
    void nitroBoosterOn();
    void nitroBoosterOff();
}
```

- ▶ Interfejsy mogą rozszerzać inne interfejsy.
- ▶ Interfejs nie może implementować innego interfejsu.

Rozszerzać można wiele interfejsów

```
interface GearBoxActions
{
    int gearUp();
    int gearDown();
}

interface SpeedControl
{
    void pressBreak();
    void releaseBreak();

    void accelerate();
    void slowDown();
}

interface BasicDriverActions extends GearBoxActions, SpeedControl
{
    void start();
    void stop();
}
```

Należy pamiętać, że ostatecznie jakaś klasa musi zaimplementować metody zadeklarowane w poszczególnych interfejsach.

Implementacja złożonych interfejsów

```
class BasicDriver implements BasicDriverActions {  
    @Override  
    public int gearUp() { ... }  
  
    @Override  
    public int gearDown() { ... }  
  
    @Override  
    public void pressBreak() { ... }  
  
    @Override  
    public void releaseBreak() { ... }  
  
    @Override  
    public void accelerate() { ... }  
  
    @Override  
    public void slowDown() { ... }  
  
    @Override  
    public void start() { ... }  
  
    @Override  
    public void stop() { ... }  
}
```

Elastycznym i wygodnym rozwiązaniem jest połączenie koncepcji interfejsów i klas abstrakcyjnych.

Klasy abstrakcyjne + interfejsy

```
abstract class BasicDriver implements SpeedControl {  
    public String nickName = "";  
    public boolean available = true;  
}
```

```
class Driver extends BasicDriver implements GearBoxActions {  
    @Override  
    public int gearUp() { ... }  
  
    @Override  
    public int gearDown() { ... }  
  
    @Override  
    public void pressBreak() { ... }  
  
    @Override  
    public void releaseBreak() { ... }  
  
    @Override  
    public void accelerate() { ... }  
  
    @Override  
    public void slowDown() { ... }  
}
```

Nie zawsze klasy abstrakcyjne są użyteczne

```
class BasicDriver implements SpeedControl {  
    @Override  
    public void pressBreak() { ... }  
  
    @Override  
    public void releaseBreak() { ... }  
  
    @Override  
    public void accelerate() { ... }  
  
    @Override  
    public void slowDown() { ... }  
  
    public String nickName = "";  
    public boolean available = true;  
}
```

```
class Driver extends BasicDriver implements GearBoxActions {  
    @Override  
    public int gearUp() { ... }  
  
    @Override  
    public int gearDown() { ... }  
}
```

Jeszcze raz przykład z figurami

```
abstract class Figure implements BasicCalculations
```

```
{  
    public Figure() {}  
  
    public String getName()  
    {  
        return "Figura";  
    }  
}
```

Klasa abstrakcyjna wykorzystuje interfejs, ale może go w rzeczywistości nie implementować — musi to zrobić pierwsza nieabstrakcyjna klasa pochodna

- ▶ Abstrakcyjna klasa słowem kluczowym *implements* sygnalizuje, że wykorzystuje określony interfejs, choć w istocie może go nie implementować w pełni.
- ▶ Obowiązek implementacji metod interfejsu spada na klasy pochodne, każda nieabstrakcyjna klasa pochodna musi posiadać pełną implementację interfejsu.

Jeszcze raz przykład z figurami

```
class Square extends Figure
```

```
{  
    public Square()  
    {  
        super();  
    }  
}
```

```
@Override  
public double area()  
{  
    return side * side;  
}
```

```
@Override  
public double circumference()  
{  
    return 4 * side;  
}
```

```
    public int side = 0;
```

```
}
```

Dziedziczenie po klasie abstrakcyjnej,
która nie zdefiniowała interfejsu,
wymagana implementacja wszystkich
metod interfejsu

Jeszcze raz przykład z figurami

```
class Circle extends Figure
{
    public Circle()
    {
        super();
    }

    @Override
    public double area()
    {
        return Math.PI * radius * radius;
    }

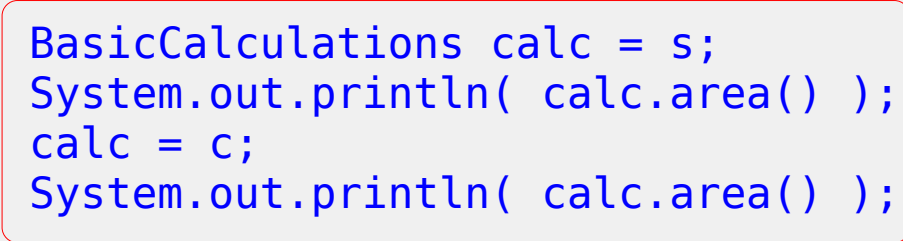
    @Override
    public double circumference()
    {
        return 2 * Math.PI * radius;
    }

    public int radius = 0;
}
```

Jeszcze raz przykład z figurami – interfejs jako obiekt

```
class FigDemoInterface
{
    public static void main(String[] args)
    {
        . . .
        Square s = new Square();
        Circle c = new Circle();

        BasicCalculations calc = s;
        System.out.println( calc.area() );
        calc = c;
        System.out.println( calc.area() );
        . . .
    }
}
```



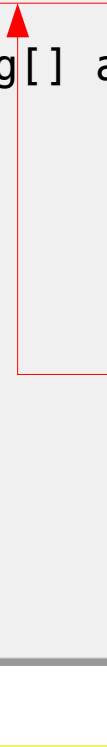
Uaktywnienie zachowania obiektu za pośrednictwem obiektu interfejsowego –

Jeszcze raz przykład z figurami — funkcja bazująca na zachowaniu

```
class FigDemoInterface
{
    public static void calcAndPrintAreas( BasicCalculations bc )
    {
        System.out.println( bc.area() );
    }

    public static void main(String[] args)
    {
        . . .
        Square s = new Square();
        Circle c = new Circle();

        calcAndPrintAreas( s );
        calcAndPrintAreas( c );
        . . .
    }
}
```



Interfejsy pozwalają na tworzenie kodu bazującego na *zachowaniu obiektów* a nie na ich typie — przynależności do zadanej hierarchii klas.

Interfejsy C#

Właściwe podobnie jak w języku Java...

Interfejsy w C# – podstawowe informacje

- ▶ Nie można definiować pól w interfejsie, nawet pól statycznych.
- ▶ Nie można definiować konstruktora i destruktora wewnątrz interfejsu.
- ▶ Wszystkie metody interfejsu są publiczne.
- ▶ Interfejs może zawierać właściwości.
- ▶ Nie można zagnieździć klasy, struktury, typów wyliczeniowych i innych interfejsów wewnątrz interfejsu.
- ▶ Interfejs nie może dziedziczyć po klasie.
- ▶ Interfejs może dziedziczyć zachowanie po innym interfejsie.

Interfejsy w C# – podstawowe informacje

```
class Driver : GearBoxActions
{
    public Driver() {}

    public int gearUp()
    {
        if( currentGear < numberOfGears )
            ++currentGear;
        return currentGear;
    }

    public int gearDown()
    {
        if( currentGear > 0 )
            --currentGear;
        return currentGear;
    }

    public int currentGear = 0;
}
```

Klasa *implementuje* dany
interfejs