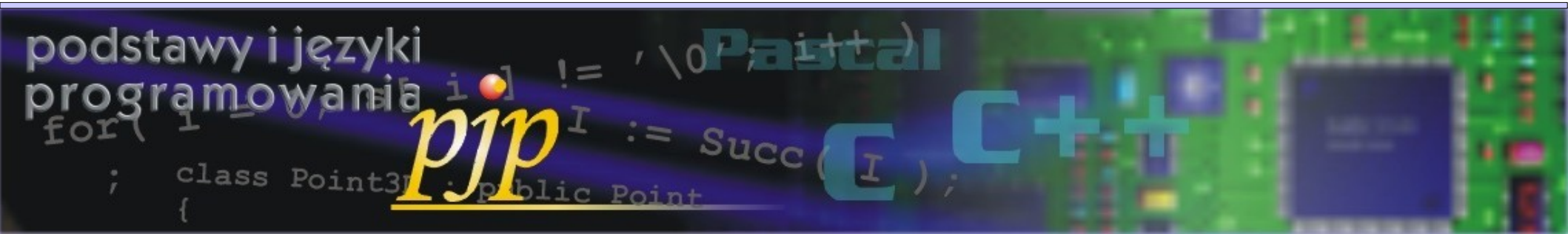
Programowanie obiektowe w języku C++

Roman Simiński
roman.siminski@us.edu.pl
www.siminskionline.pl

Koncepcja programowania obiektowego



Programowanie obiektowe — trochę historii

- Programowanie obiektowe nie jest nowością ostatnich lat — obiektowość w programowaniu pojawia się mniej lub bardziej jawnie od wielu lat.
- Pierwszym znanym językiem *wykorzystującym obiekty* był język *Simula67*, opracowany w latach 60-tych w Norweskim Ośrodku Obliczeniowym w Oslo przez O-J. Dahla, B. Myhrhauga i K. Nygaarda.
- Termin *object oriented programming* pojawił się w latach 70-tych wraz z językiem *Smalltalk*, wywodzącym się z Xerox Palo Alto Research Center.
- Język *Smalltalk,* inspirowany językiem *Simula 67*, posiada szereg interesujących właściwości. Oprócz cech obiektowych posiada rozbudowane biblioteki, środowisko programistyczne, działa z wykorzystaniem maszyny wirtualnej.
- Mimo swojego nowatorstwa jak i zaawansowania technologicznego *Smalltalk* nie zdobywa popularności, staje się jednak *źródłem inspiracji* dla wielu innych języków obiektowych.

Programowanie obiektowe — trochę historii, cd...

- Koncepcja obiektowości w programowaniu znajduje wielu zwolenników. Powstają obiektowe rozszerzenia istniejących języków programowania (np. *Object Pascal*, *C++*, *Objective-C*).
- Powstają nowe języki, rdzennie posiadające wbudowane mechanizmy obiektowe. Są to zarówno języki ortodoksyjnie obiektowe (np. *Oberon*, *Ruby*), jak i języki wywodzące się z nurtu hybrydowego (np. *Java*, *JavaScript*, *C#*, *PHP*).
- Obiektowość w programowaniu jest jednolita na poziomie koncepcji, jednak w zakresie rozwiązań szczegółowych i implementacyjnych występują różnice.
- Popularne aktualnie języki programowania, takie jak *Java*, *C#*, *PHP*, *JavaScript*, *ActionScript*, wykorzystują (w różnym co prawda stopniu) koncepcję programowania obiektowego wywodzącą się z języka *C++*.

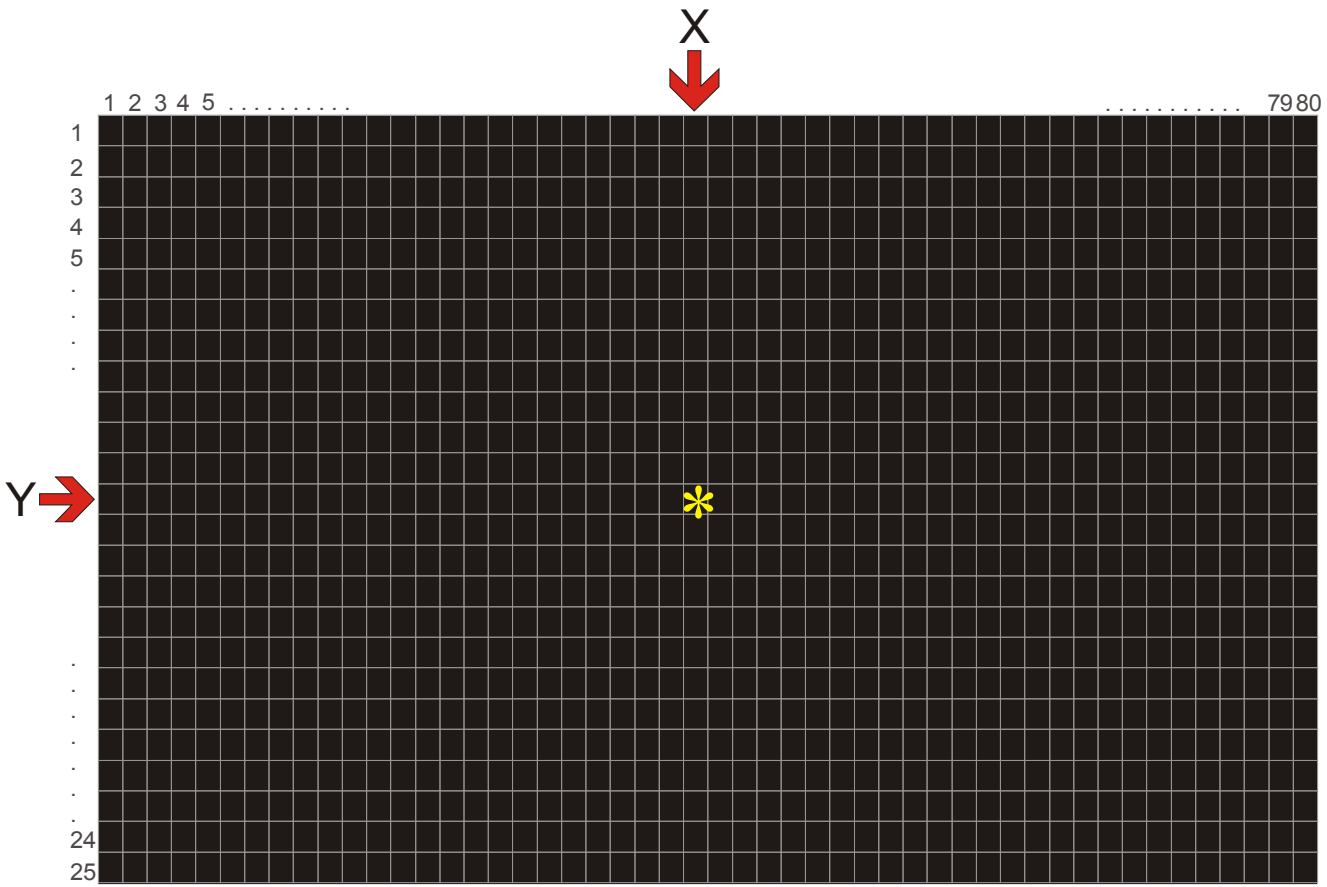
Programowanie obiektowe — ogólna koncepcja

- Co to jest obiekt?
- Jak jest zbudowany?
- Jak działa?

Koncepcja programowania obiektowego jest prosta i naturalna. Aby ją sugestywnie przedstawić, proponuję rozważenie pewnego przykładu, zrealizowanego najpierw strukturalnie a potem obiektowo.

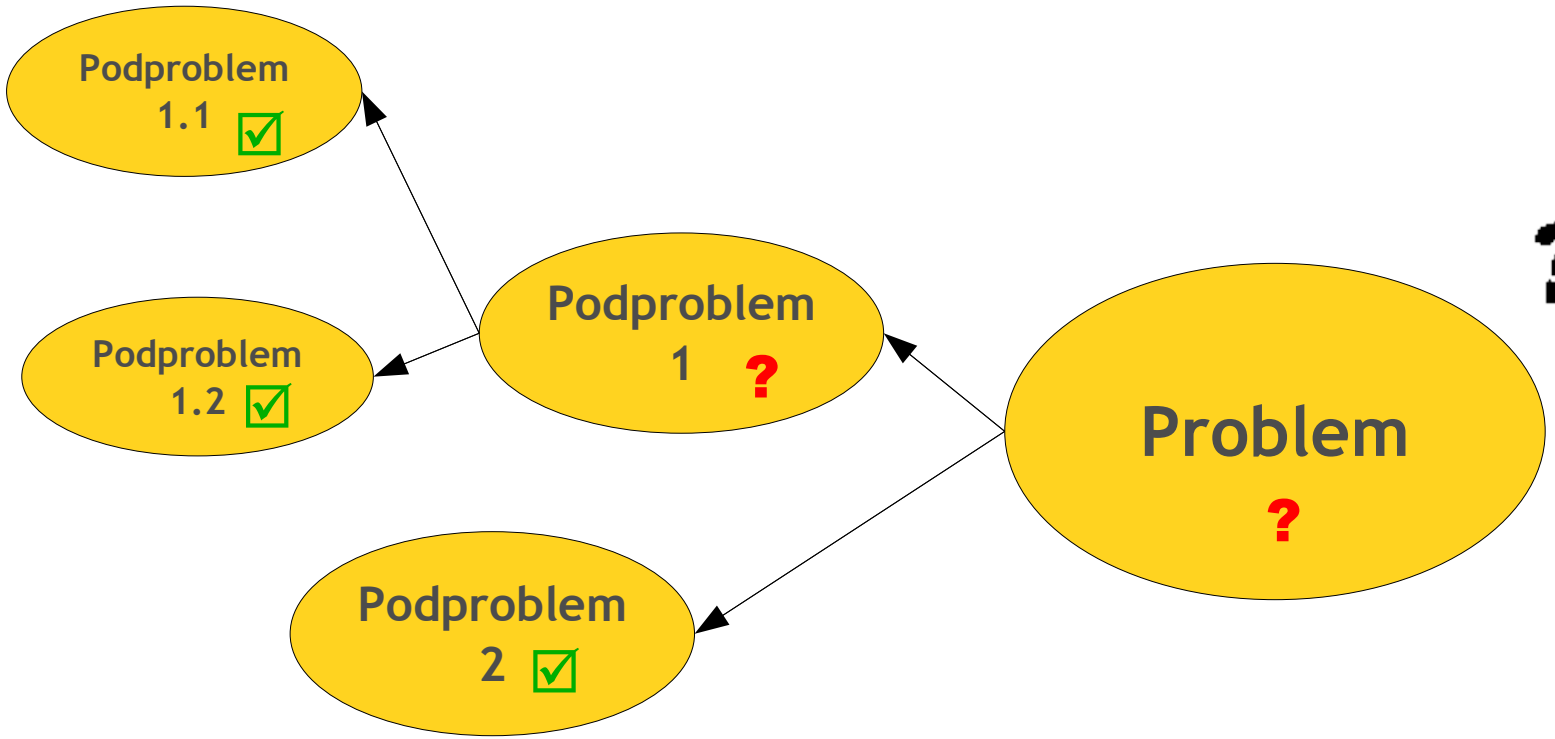
Przykład

Napisać program wyświetlający na pustym ekranie tekstowym zadany znak (np. '*'), pozwalający na przesuwanie tego znaku w górę, dół, lewo i prawo, przy użyciu klawiszy sterowania kursorem.



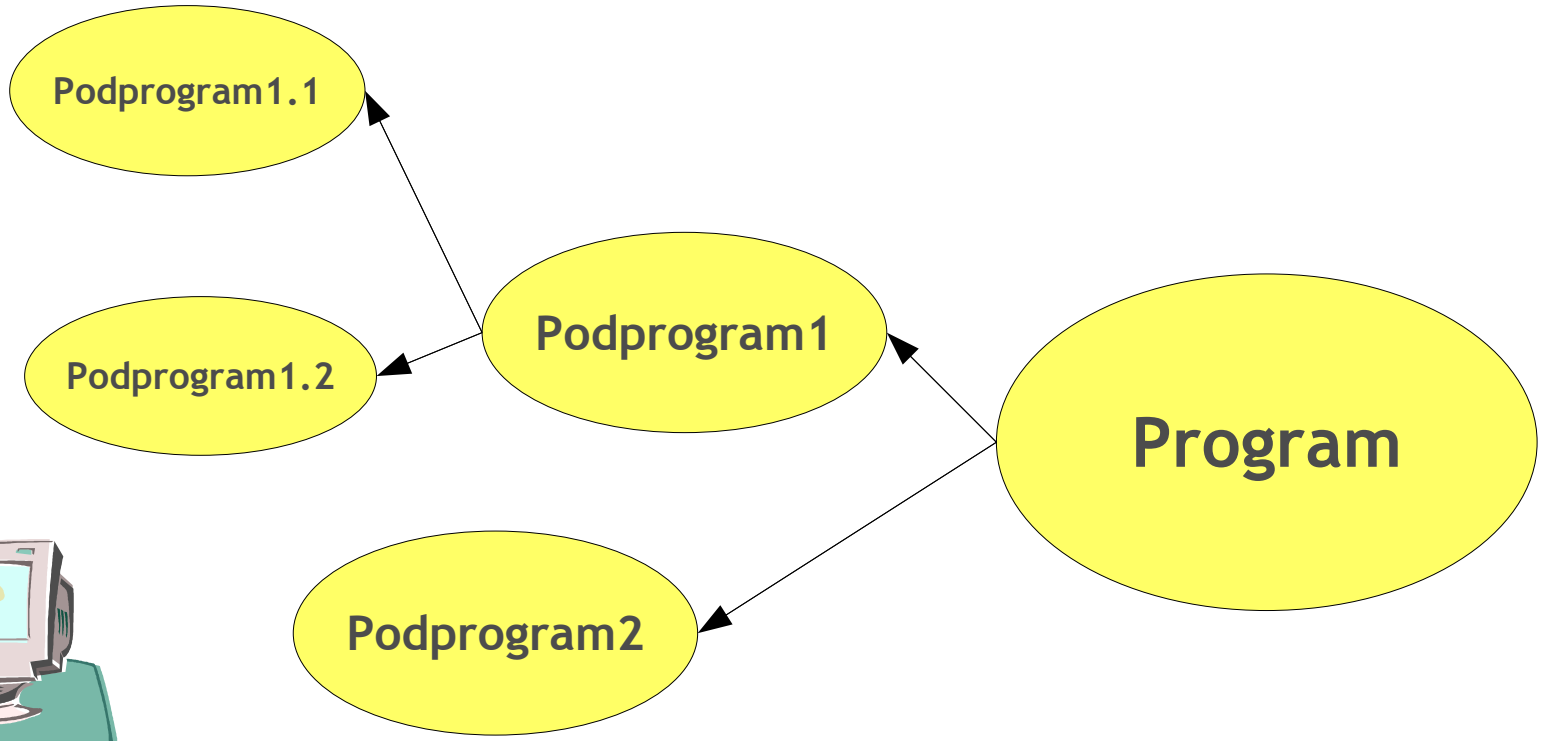
Dekompozycja problemu – z mniejszym być może łatwiej walczyć

- Rozwiązanie zadanego problemu w programowaniu polega na wykonaniu sekwencji pewnych *czynności*, zgodnie z algorytmem rozwiązania tego problemu.
- *Dekompozycyjna strategia rozwiązywania problemów* polega na dzieleniu potencjalnie *dużego i złożonego problemu na podproblemy* tak długo, aż wyodrębnione podproblemy staną się proste do rozwiązania.



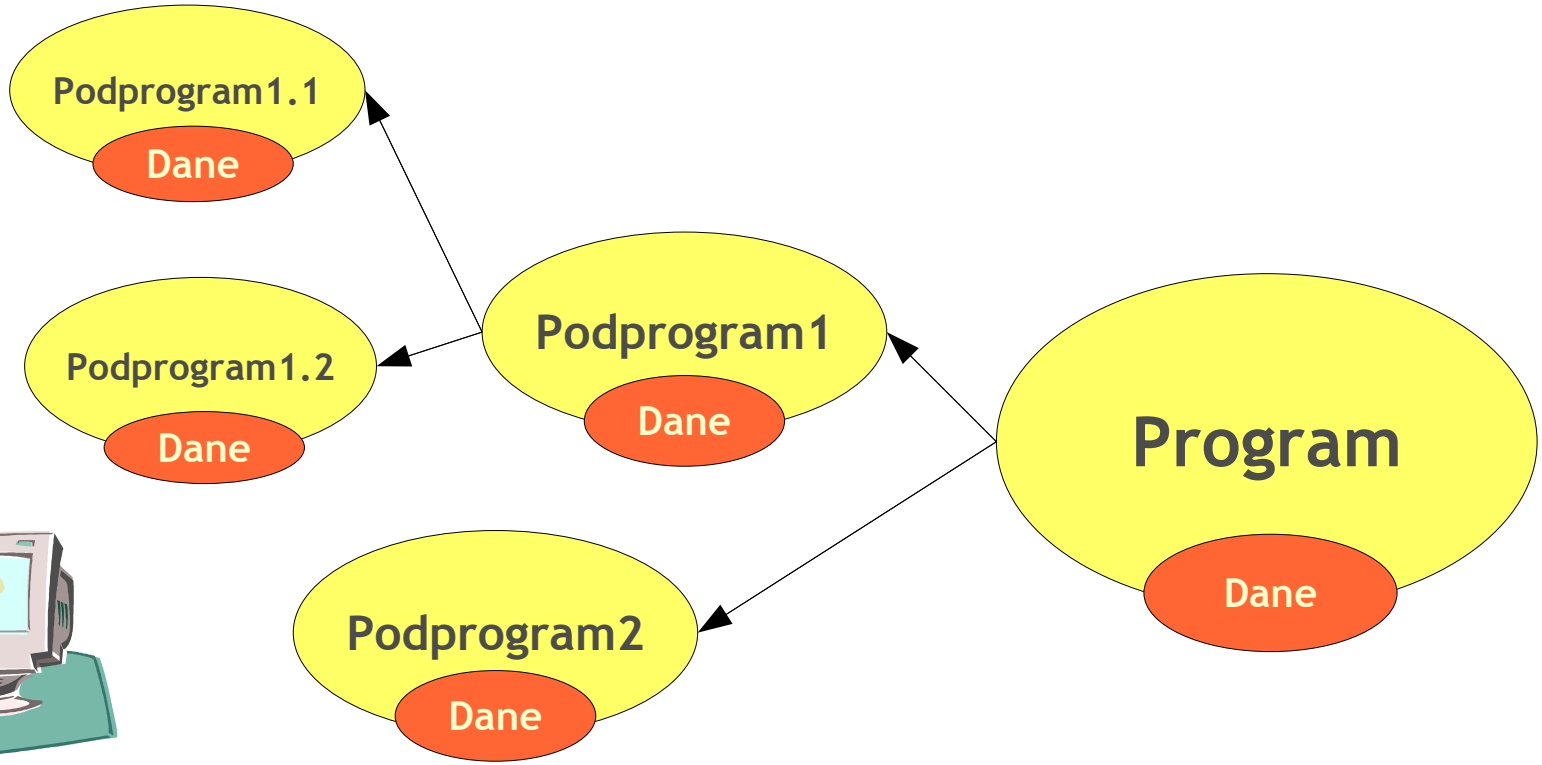
Podejście proceduralne

- *Proceduralne podejście* do programowania zakłada, że złożone czynności niezbędne do rozwiązania problemu będą opisane *procedurami (podprogramami),* rozbijanymi na kolejne *podprocedury* tak długo, aż uzyskane zostaną czynności proste, łatwe do zaprogramowania.



Podejście strukturalne

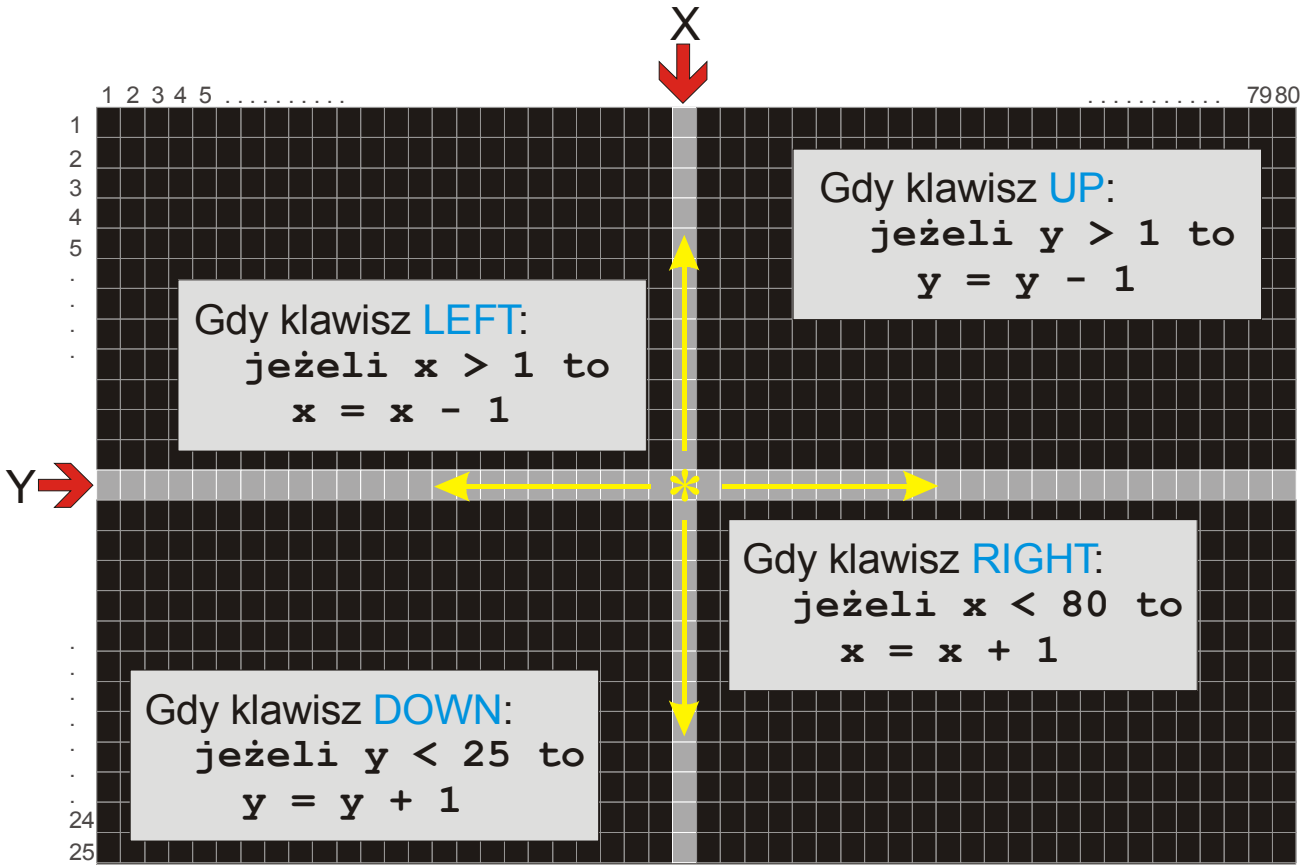
- *Podejście strukturalne* rozwija ideę programowania proceduralnego, zakładając tworzeniu programów w taki sposób, że blok główny programu stanowi szkielet, określający *kolejność realizacji czynności*, zapisanych w postaci odpowiednich procedur i/lub funkcji.
- Podprogramy wykorzystują *globalne* i *lokalne* struktury danych.



Analiza — dane i czynności związane z przesuwaniem punktu

- Pozycja przesuwanego znaku będzie opisana parą zmiennych **`x`** i **`y`**, kod tego znaku (wygląd) będzie opisany zmienną **`shape`**.

- Przesuwanie znaku polega na wymazywaniu znaku na pozycji poprzedniej i ponownym wyświetlaniu na pozycji nowej, o ile jest ona prawidłowa.

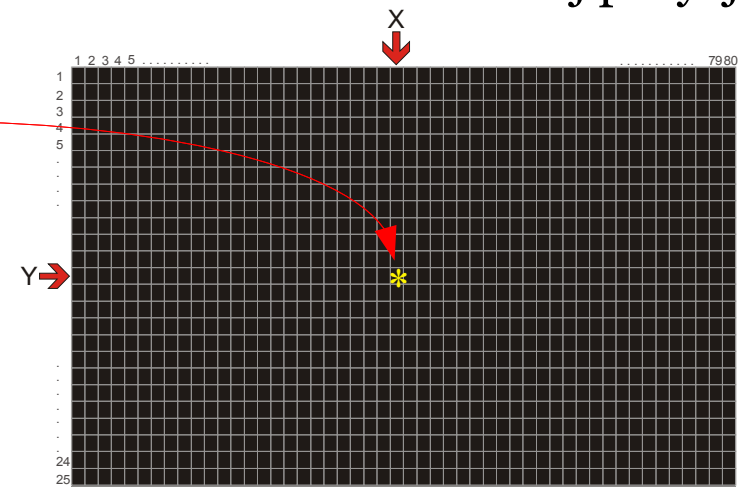


Analiza — wymagane funkcje obsługi konsoli

Zakładamy, że mamy do dyspozycji funkcje obsługi ekranu konsoli:

- Funkcję **`clearScreen`**, czyszczącą ekran tekstowy.
- Funkcję **`writeCharXY`**, wyświetlającą na ekranie znak na określonej pozycji.

```
writeCharXY( 40, 12, '*' );
```



Standard bibliotek języka C/C++ nie zawiera funkcji obsługi konsoli — operacje te są mocno zależne od środowiska systemowego, w niektórych zresztą nie występują wcale. Operacje te należy zaimplementować samemu lub wykorzystać rozszerzenia dodawane przez producentów kompilatorów.

Przykładowe implementacje powyższych funkcji zawiera suplement umieszczony na końcu tego dokumentu.

Analiza — wymagane funkcje obsługi konsoli

Zakładamy, że mamy do dyspozycji obsługę klawiatury konsoli:

- Funkcję **getKey**, której rezultatem jest kod klawisza, w tym tzw. *klawiszy sterujących kursorem* (czytanie znaku bez echa na ekranie i bez "potwierdzania" Enterem).
- Stałe określające kody klawiszy, wtym klawiszy sterujących kursorem (dla funkcji **getKey**): **KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT, KEY_ESC.**

```
int  key;
key = getKey(); /* Czekaj na naciśnięcie klawisza */
switch( key )
{
  case KEY_UP    : /* Obsługa strzałki w gorę  */
                   break;
  case KEY_DOWN  : /* Obsługa strzałki w dół   */
                   break;
  case KEY_LEFT  : /* Obsługa strzałki w lewo  */
                   break;
  case KEY_RIGHT : /* Obsługa strzałki w prawo */
                   break;
}
```

Przykładową implementację funkcji **getKey** zawiera suplement umieszczony na końcu tego dokumentu.

Realizacja strukturalna — najprostsza wersja

```
int  key, x = 40, y = 12;
char shape = '*';

clearScreen();

do
{
  writeCharXY( x, y, shape );   /* Wyświetl znak na zadanej pozycji */
  key = getKey();               /* Czekaj na naciśnięcie klawisza   */
  writeCharXY( x, y, ' ' );     /* Wymaż znak, potencjalny ruch     */

  switch( key )
  {
    case KEY_UP    : if( y > 1 ) --y;   /* Przeskocz wyżej   */
                     break;
    case KEY_DOWN  : if( y < 24 ) ++y;  /* Przeskocz niżej   */
                     break;
    case KEY_LEFT  : if( x > 1 ) --x;   /* Przeskocz w lewo  */
                     break;
    case KEY_RIGHT : if( x < 80 ) ++x;  /* Przeskocz w prawo */
                     break;
  }
}
while( key != KEY_ESC );
```

Realizacja strukturalna — wady

Wady techniczne

- Brak jawnego powiązania pomiędzy zmiennymi `x`, `y` i `shape`. A przecież te zmienne opisują jeden obiekt!
- Brak jawnego powiązania pomiędzy fragmentem wyświetlającym (ukrywającym) punkt oraz zmiennymi go opisującymi.
- Siermiężne testowanie poprawności współrzędnych.

Wady metodyczne

- Z kodu programu nie wynika bezpośrednio, jakie jest jego przeznaczenie, brak powiązań pomiędzy wykonywanymi akcjami a danymi.
- Sam algorytm jest prosty, lecz jego zapis nie wspomaga zrozumienia jaka jest istota rozwiązywanego problemu.

Od programowanie proceduralnego do obiektowego

Podejdźmy do problemu inaczej — spróbujmy zrozumieć jego istotę

Zapraszamy na:

Krótki wywiad z głównym bohaterem programu!

Czwartek
24 maja 2011

Redaktor Prowadzący
Roman Simiński

Fakty IT

gazeta codzienna

Niecodzienny dodatek *Gazety Codziennej*

Bohaterem dzisiejszego wywiadu jest ruchomy punkt ekranowy. Dzięki temu wyjątkowemu wywiadowi poznamy unikatowe fakty z życia przedstawicieli tego gatunku.

Tylko u nas, ekskluzywny wywiad z głównym bohaterem Programu

Jak się nazywasz?

— No, wiesz, na imię mam nieciekawie ... jestem *R*, po prostu *R*.

Kim jesteś?

— Jestem ruchomym punktem ekranowym. Ponieważ zasuwam po całym ekranie, można o mnie powiedzieć z angielska, że jestem jednym z Runner'ów.

Czyli może Was być więcej?

— Takich jak ja może być więcej. W tym programie potrzeba tylko jednego Runner'a, więc oto jestem.

Powiedz nam coś o sobie...

— Co tu dużo gadać. Jak trzeba to mnie widać, jak nie trzeba to znikam. Potrafię się przesunąć o jedno pole w górę, w dół, w lewo lub w prawo. Nie wylecę poza ekran — to pewne — dbam o to. Możesz mi podać pozycję ekranową oraz znak, jakim mam być.

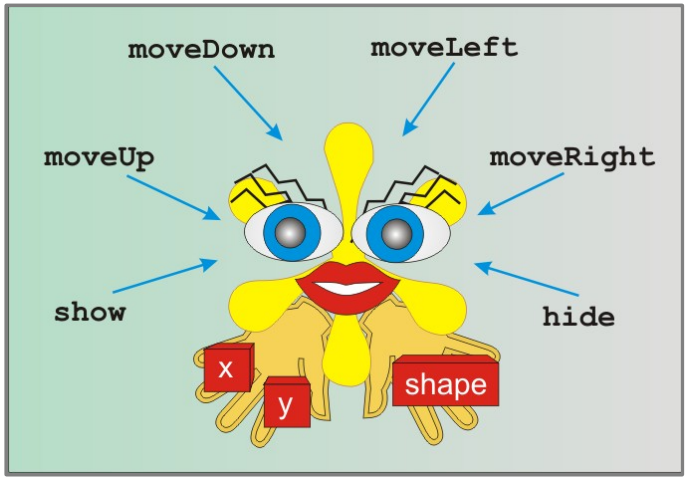
gazeta codzienna

Jak to „możesz mi podać”?

— Co jak to... zwyczajnie. Przecież wiesz jak mam na imię, to mnie zawołaj, ustal gdzie mam być na ekranie i jak mam wyglądać. W ogóle to możesz mną sterować, zawołaj mnie i prześlij mi *komunikat*, rozumiesz, takie polecenie co mam zrobić.

Jak niby mam Ci przesyłać komunikat?

— Jaki ty jesteś... no, niedoinformowany. Przecież już Ci mówiłem, co potrafię. Dokładnie komunikaty nazywają się: *show*, *hide*, *moveUp*, *moveDown*, *moveLeft*, *moveRigt*. Dodatkowo mam swoją wewnętrzną pamięć i trzymam tam wszystko, co powinienem wiedzieć. Pozycję przechowuję w komórkach o nazwie *x* i *y*, a znak jakim mam być, znaczy, wyglądać, w szufladce *shape*. Chcesz, to mi coś tam wstaw. Co łaska, byleby to było legalne. Wiesz, my gwiazdy programów musimy uważać ... .



A skąd Ty to wszystko wiesz i potrafisz?

— Nauczył mnie tego programista. Wymyślił, że będą Runner’y, określił, co każdy Runner powinien wiedzieć, co powinien robić. A potem nauczył nas, Runner’ów jak to wszystko mamy robić. Ja jestem Runner'em, to umiem to wszystko. Każdy Runner umie to wszystko. No, przydałoby się, żeby nauczyła nas jeszcze kilku rzeczy..., no wiesz... ale nie nauczył :(.

gazeta codzienna

A skoro już tu jesteś... mogę trochę Ciebie poużywać?

— Taaak, Wy to, jak zwykle, zawsze o jednym. A dobra masz mieć, nie jestem mydełko, nie wymydlę się. Ale ostrożnie, najpierw Ci wszystko wyjaśnię. Jeżeli chcesz, żebym wiedział, że mam się pojawić np. w 40-stej kolumnie 12-go wiersza, to piszesz w swoim programie:

```
R.x = 40;
R.y = 12;
```

— Zauważ, że piszesz najpierw moje imię, potem po kropce nazwę miejsca, gdzie przechowuję dane. To miejsce nazywam *polem*. Możesz do tego pola przypisać wartość np.: 40, 12. To tak, jakbyś mi wkładał tą wartość do kieszeni.

Jeżeli chcesz, żebym pokazał się na ekranie piszesz:

```
R.show();
```

— I znowu, piszesz moje imię, potem po kropce nazwę polecenia, jakie mam wykonać. To tak jakbyś do mnie wysyłał *komunikat*. To wygląda trochę jak wywołanie funkcji w języku C..., i rzeczywiście, na Twój *komunikat* odpowiadam wywołaniem funkcji, którą mam w siebie wbudowaną. Taka funkcja nazywa się *funkcją składową*.

gazeta codzienna

— Jeżeli mam się przesunąć o jedno pole do góry, potem o jedno w lewo i jeszcze raz o jedno do góry, to piszesz:

```
R.moveUp();
R.moveLeft();
R.moveUp();
```

— Jeżeli chcesz, żebym przeszedł po kwadracie: pięć kroków w prawo, pięć w dół, pięć w lewo i pięć w górę, to napiszesz np.:

```
int steps;

R.x = 1;  // Powiedz mi, skąd mam wystartować
R.y = 1;

R.show(); // Powiedz żebym się pokazał

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w prawo
  R.moveRight(); // Krok w prawo

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w dół
  R.moveDown();  // Krok w dół

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w lewo
  R.moveLeft();  // Krok w lewo

for( steps = 5; steps > 0 ; steps-- ) // Pięć kroków w górę
  R.moveUp();    // Krok w górę
```

Czy możesz naszym czytelnikom wyjaśnić jak są zbudowane funkcje składowe, np. *show*, *hide*?

— Mogę..., ale mi się nie chce. Zapytaj o to programistę. Ja jestem od zasuwania na ekranie a nie od programowania.

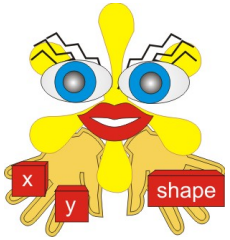
Szkoda..., ale i tak serdecznie dziękujemy za rozmowę.

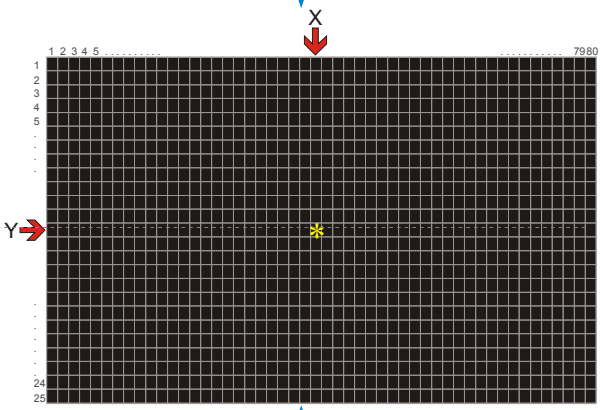
Realizacja obiektowa – najprostsza wersja

```
Runner R; // Utworzenie obiektu o nazwie R i typie Runner

R.x = 40; // Ustalenie pozycji i wyglądu Runnera'a R
R.y = 12;
R.shape = '*';

clearScreen();
R.show();  // Niech R się pokaże
do
{
  key = getKey();
  switch( key )
  {
    case KEY_UP    : R.moveUp();    // Niech R się przesunie w górę
                     break;
    case KEY_DOWN  : R.moveDown();  // Niech R się przesunie w dół
                     break;
    case KEY_LEFT  : R.moveLeft();  // Niech R się przesunie w lewo
                     break;
    case KEY_RIGHT : R.moveRight(); // Niech R się przesunie w prawo
                     break;
  }
}
while( key != KEY_ESC );
```

Obiektowo vs. strukturalnie

```
R.x = 40;
R.y = 12;
R.shape = '*';

R.show();
. . .
R.moveUp();
. . .
R.moveDown();
. . .
R.hide();
. . .
```





Obiektowo

Strukturalnie

```
int key, x = 40, y = 12;
int shape = '*';

clearScreen();

writeCharXY( x, y, shape );
key = getKey();
writeCharXY( x, y, ' ' );
. . .
if( y > 1 ) --y;
. . .
if( y < 24 ) ++y;
. . .
```

Obiektowo vs. strukturalnie

Obiektowo

- Wyodrębnij najważniesze *obiekty* w rozważanym problemie, ustal co powinny *wiedzieć* oraz co powinny *umieć*. *Naucz* je tego wszystkiego.
- Jeżeli problem tego wymaga, połącz obiekty w *grupy*, ustal *szefa grupy* — jemu będziesz przekazywała *polecenia*, on będzie *zarządzał* pracą grupy.
- Utwórz potrzebne *obiekty*, przekaż im konieczne *informacje* i każ im *pracować* wysyłając odpowiednie *komunikaty*, tak zby rozwiązać rozważany problem.

Strukturalnie

- Ustal jakie *dane* są istotne dla rozważanego problemu, dobierz odpowiednie *struktury* do ich przechowywania.
- Ustal jakie *czynności* powinny być wykonane na *danych*, jaka powinna być ich *kolejność*. Jeżeli ustalone czynności są *złożone*, dziel je na *podczynności* tak długo, aż będą łatwe do zaprogramowania. Zbuduj odpowiednie podprogramy.
- Uaktywnij zaplanowane *czynności* w ustalonej *kolejności*, tak aby rozwiązać rozważany problem.

Programming in the small → programming in the large.

- Programowanie *proceduralne* jest zorientowane na *czynności* wykonywane na *strukturach danych*.
- Czynności są realizowane przez *podprogramy* (*procedury* i *funkcje*) grupowane zwykle w *moduły*.
- Programowanie *proceduralne* wraz ze *wzrostem rozmiaru programu* zaczyna być coraz *trudniejsze*.
- Coraz więcej *podprogramów*, coraz więcej *modułów*, coraz więcej nieintuicyjnych i niejasnych *powiązań* pomiędzy podprogramami a strukturami danych.

Potrzebujemy ***porządku*** i ***hierarchii*** — trzeba jasno określić *kto* i *za co* jest odpowiedzialny.

W programie muszą się pojawić *kompetentne podmioty*, zarówno te *zarządzające* (kadra „managerska”) jak i te *zarządzane*.

W programowaniu *podmiot*, czyli *ktoś* to *obiekt*.

Obiektowość — analiza, projektowanie, programowanie

- Obiektowość jest strategią pozwalającą zapanować na złożonym projektem.
- Może być stosowana w każdej z faz realizacji systemu, zapewniając spójny i jednolity opis systemu w każdej z tych faz.
- W wymiarze czysto programistycznym obiektowość jest strategią organizacji kodu potencjalnie *złożonego* oprogramowania.
- Programowanie obiektowe *nie stoi w sprzeczności* z ideą programowania strukturalnego, stanowi raczej jego *nadbudowę*.
- Zazwyczaj realizacja *metod obiektów* w systemie przebiega zgodnie z koncepcją programowania *strukturalnego*.

Programowanie obiektowe — *strategiczne* narzędzie realizacji złożonych systemów.

Programowanie proceduralne — *taktyczne* narzędzie realizacji mniejszych systemów, bądź *wybranych elementów* systemów złożonych.

Obiektowość — przepis na program obiektowy raz jeszcze

- Wyodrębnij najważniesze *obiekty* w rozważanym problemie, ustal co powinny *wiedzieć* oraz co powinny *umieć*.
- Jeżeli problem tego wymaga, połącz obiekty w grupy, ustal szefów grup — im będziesz przekazywała *polecenia,* oni będą *zarządzali* pracą grupy.
- Sprawdź czy istnieją odpowiadające Ci *gotowe szablony* dla potrzebnych obiektów. Jeżeli tak to *użyj ich*, jeżeli nie to zbuduj *własne szablony*.
- Utwórz potrzebne *obiekty*, przekaż im konieczne *informacje* i każ im *pracować* wysyłając odpowiednie *komunikaty*, tak zby rozwiązać rozważany problem.

Wyodrębniliśmy już najważniejszy obiekt w naszym problemie, wiadomo co powinien *wiedzieć* i co powinien *umieć*. Jak go tego nauczyć?

Klasa Runner — szablon do tworzenia obiektów

- **`Runner`** to nazwa *klasy obiektów*. Wszystkie obiekty klasy **`Runner`** będą posiadały te same cechy i umiejętności.
- *Klasa* swoją deklarację przypomina deklarację *rekordu* (*struktury*).
- Klasa będzie posiadała *składowe*, którymi będą *pola* i *funkcje*.
- Deklaracja pustej klasy:

```
class Runner
{
  public:

      . . .
};
```

Wszystkie elementy zadeklarowane w sekcji **`public`** są dostępne dla tych, którzy chcą używać obiektu klasy Runner:

```
Runner R;
R.x = 40;
R.y = 12;
R.shape = '*';
```

Anatomia klasy Runner — pola, czyli to co obiekt wie

- To, co obiekt *wie o sobie i swoim otoczeniu* pamiętane jest w jego *polach*.
- Pola stanowią pamięć obiektu – każdy obiekt ma własny zestaw niestatyczych pól.
- Polami w klasie **`Runner`** są **`x`** i **`y`** — opisujące ekranową pozycję obiektu, oraz **`shape`** — opisujące wygląd obiektu.

```
class Runner
{
  public :
    int  x, y;
    char shape;

    . . .
};
```

Pola w obrębie klasy deklaruje się podobnie jak pola w obrębie typu rekordowego.

Anatomia klasy Runner – funkcje składowe, czyli to co obiekt potrafi

- Każdy obiekt klasy **Runner** posiada funkcje składowe, które mogą być przez niego wywoływane.
- Funkcje składowe pozwalają obiektowy *działać*, wykonywać *akcje*. W trakcie *wykonywania* funkcji składowych obiekt korzysta zwykle z informacji zapisanej w jego polach.

```
class Runner
{
  public :
    int  x, y;
    char shape;

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};
```

Prototypy funkcji składowych, ich ciała mogą być definiowane są poza klasą.

Anatomia klasy Runner — wywołanie funkcji a przesłanie komunikatu

- Podprogramy wbudowane w obiekt nazywa się w programowaniu obiektowym *metodami*, w języku C++ metody nazywane są *funkcjami składowymi* (ang. *member function*).
- Wywołanie funkcji składowej nazywane jest często *przesłaniem komunikatu* do obiektu.
- Obiekt "odpowiada" na komunikat wykonując kod odpowiedniej funkcji składowej.

```
switch( key )
{
  case KEY_UP    : R.moveUp();
                   break;
  case KEY_DOWN  : R.moveDown();
                   break;
  case KEY_LEFT  : R.moveLeft();
                   break;
  case KEY_RIGHT : R.moveRight();
                   break;
}
```

Wywołanie funkcji składowych dla obiektu *R*, inaczej — *przesłanie komunikatu* do obiektu.

Anatomia klasy Runner — implementacje funkcji składowych

- Właściwe *definicje* (*implementacje* zawierające konkretny kod) funkcji składowych umieszczone są zwykle poza definicją klasy.
- Nazwa każdej funcji składowej zdefiniowanej poza klasą jest kwalifikowana *nazwą klasy* wraz z *operatorem zakresu* "::".

```
class Runner
{
  . . .
  void show();
  void hide();
};

void Runner::show()
{
  . . .
}

void Runner::hide()
{
  . . .
}
```

Prototypy (deklaracje) funkcji składowych.

Definicje (implementacje) funkcji składowych.

Anatomia klasy Runner — implementacje funkcji show i hide

- Wyprowadź znak zapisany w polu **code** na pozycji zapisanej w polach **x** i **y**.

```
void Runner::show()
{
  writeCharXY( x, y, shape );
}
```

- Wyprowadź spację na pozycji zapisanej w polach **x** i **y**, wymazującą zapisany tam znak.

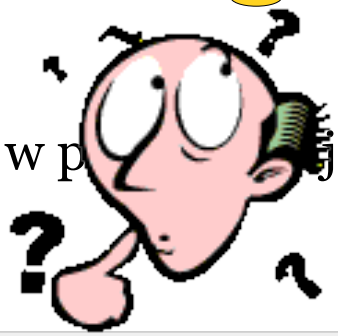
```
void Runner::hide()
{
  writeCharXY( x, y, ' ' );
}
```

Anatomia klasy Runner — implementacje funkcji show i hide

- Wyprowadź znak zapisany w polu **code** na pozycji zapisanej w polach **x** i **y**.

```
void Runner::show()
{
  writeCharXY( x, y, shape );
}
```

Do jakich **x**, **y** i **shape** odwołują się te funkcje?

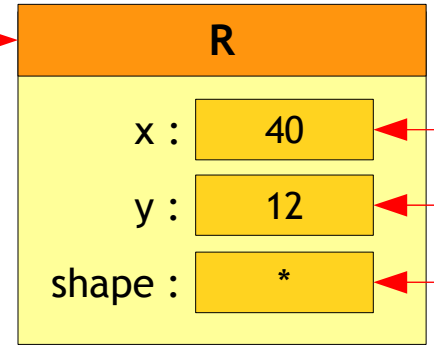


- Wyprowadź spację na pozycji zapisanej w p[illegible]jącą zapisany tam znak.

```
void Runner::hide()
{
  writeCharXY( x, y, ' ' );
}
```

Funkcje składowe wywoływane są dla konkretnego obiektu

```
Runner R;

R.x = 40;
R.y = 12;
R.shape = '*';


R.show();
```



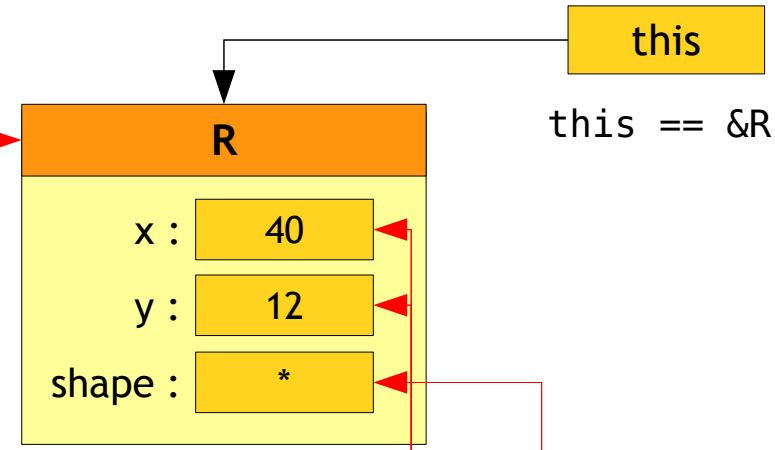

```
void Runner::show()
{
  writeCharXY( x , y , shape );
}
```

- Każda niestatyczna funkcja składowa musi zostać wywołana dla jakiegoś obiektu klasy **`Runner`**.
- W trakcie swojego działania *funkcja składowa* zwykle odwołuje się do pól obiektu, dla *którego została wywołana*.

Wskaźnik this

- W obrębie niestatycznych funkcji składowych istnieje predefiniowany wskaźnik **this**, wskazujący na obiekt, dla którego działa funkcja składowa.

```
Runner R;

R.x = 40;
R.y = 12;
R.shape = '*';


R.show();
```




```
void Runner::show()
{
  writeCharXY( this->x , this->y , this->shape );
}
```

- Wykorzystanie wskaźnika **this** jest konieczne tylko w niewielkiej liczbie przypadków i nie powinno być *nadużywane* — w tym przypadku użycie wskaźnika **this** jest niepotrzebne.

Anatomia klasy Runner — implementacje funkcji move...

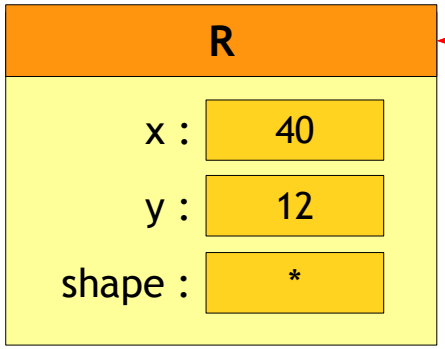
- Wymazanie znaku na bieżacej pozycji obiektu klasy **Runner**.
- Ustalenie nowej pozycji, o ile jest ona prawidłowa.
- Pokazanie znaku na nowej pozycji.

```
void Runner::moveUp()
{
  hide();
  if( y > 1 )
    --y;
  show();
}

void Runner::moveDown()
{
  hide();
  if( y < 24 )
    ++y;
  show();
}
```

```
void Runner::moveLeft()
{
  hide();
  if( x > 1 )
    --x;
  show();
}

void Runner::moveRight()
{
  hide();
  if( x < 80 )
    ++x;
  show();
}
```

Anatomia klasy Runner — implementacje funkcji move...

- W obrębie funkcji składowej (np. **moveUp**) można wywoływać inne funkcje składowe (np. **show**, **hide**), działają one wtedy dla tego obiektu, dla ktorego została wywołana funkcja nadrzędna.




```
Runner R;

R.x = 40;
R.y = 12;
R.code = '*';

R.moveUp();
```

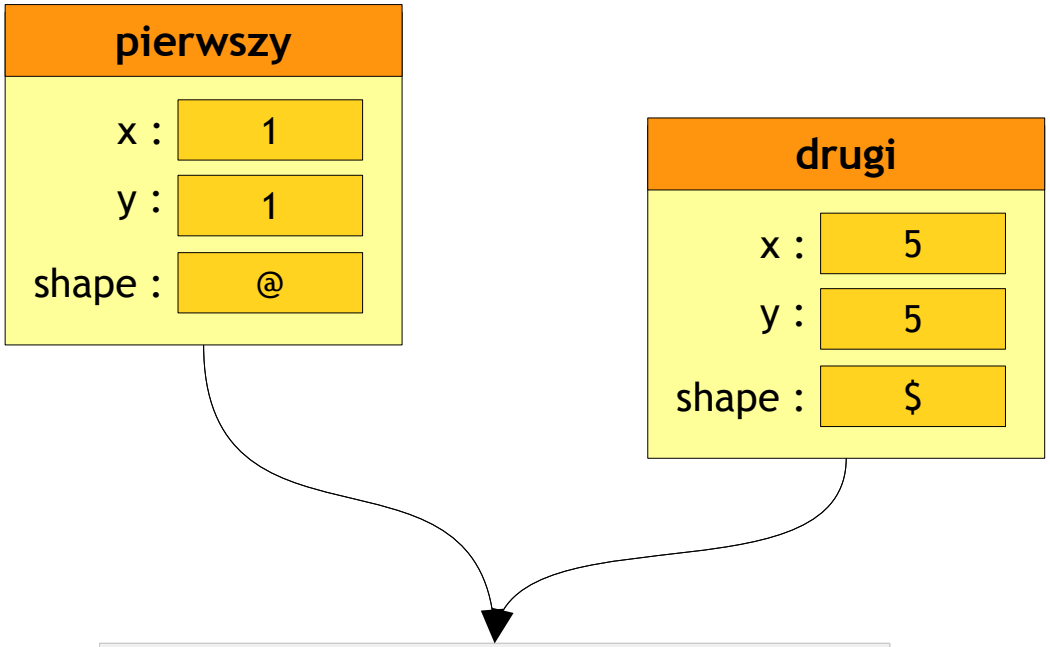
```
void Runner::moveUp()
{
  hide();
  if( y > 1 )
    --y;
  show();
}
```

```
void Runner::moveUp()
{
  this->hide();
  if( y > 1 )
    --y;
  this->show();
}
```

- Niestatycznych funkcje składowe "pracują" dla obiektu wskazywanego przez **this**.

Każdy egzemplarz klasy ma własne dane i wspólny kod funkcji

```
Runner pierwszy, drugi;

pierwszy.x = 1;
pierwszy.y = 1;
pierwszy.shape = '@';

drugi.x = 5;
drugi.y = 5;
drugi.shape = '$';


pierwszy.show();
drugi.show();
```




```
void Runner::show()
{
  writeCharXY( x, y, shape );
}
```

- Każdy obiekt na *własny* zestaw pól niestatycznych, zgodnie z definicją klasy.
- Każdy obiekt tej samej klasy wykonuje te *same* akcje, wywołując współdzielone funkcje składowe.

Obiekt — krótka definicja

Istnieje wiele, różniących się od siebie definicji pojęcia *obiekt* i *klasa*. Tutaj przyjmiemy następującą definicje:

Obiekt jest programowym tworem, stanowiącym *model* pewnego konkretnego bytu ze świata rzeczywistego, reprezentujący *rzecz* (obiekt fizyczny), *pojęcie* (obiekt konceptualny) lub obiekt jest pewnym *bytem programistycznym* nie mający swojego odpowiednika w rzeczywistości.

- W sensie programistycznym *obiekt* jest *zmienną złożoną*, zawierającą dane różnego typu. Jest zatem podobny do *rekordu*.
- Od rekordu różni go *zdolność do działania*, polegająca na możliwości przesłania do obiektu *komunikatu*, na który reaguje on *uruchomieniem* wbudowanej weń *funkcji składowej* (*metody*).

Klasa — krótka definicja

Klasa to szablon określający jak obiekt jest *zbudowany* i *co potrafi*.

W szczególności klasa zawiera informacje o nazwach, typie i reprezentacji pól, deklaracje i definicje funkcji składowych oraz informacje dotyczące relacji i powiązań z innymi klasami.

- W sensie programistycznym *klasa* jest dla *obiektu* tym, czym *typ* dla *zmiennej*.
- W języku Object Pascal mówi się: *zmienna obiektowa* i *typ obiektowy*, w C++ i pokrewnych językach obiektowych mówi się po prostu *obiekt* i *klasa*.
- Klasa ma charakter pierwotny — aby *utworzyć obiekt*, najpierw należy *zdefiniować jego klasę*.

Trzy podstawowe cechy obiektu

- Tożsamość (ang. *identity*).
- Stan (ang. *state*).
- Działanie, zachowanie (ang. *behavior*).

Tożsamość obiektu

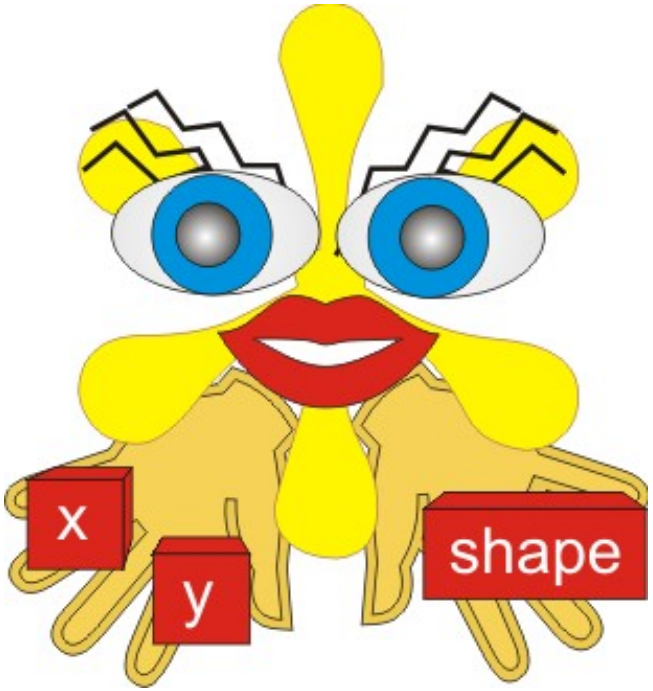
- *Tożsamość* (ang. *identity*), pozwala na odróżnienie danego obiektu od każdego innego.
- Tożsamość jest potrzebna, aby można było obiekt *zlokalizować* i *wykorzystać*.
- Programista przypisuje obiektowi *unikatową nazwę* lub włącza go do *identyfikowalnej kolekcji obiektów*, lub odwołuje się do obiektu posługując się *informacją o jego adresie* w pamięci operacyjnej.

```
Runner R;
. . .
R.show();
. . .

Runner duchy[ 3 ];
for( int i = 0; i < 3; i++ )
  duchy[ i ].show();
. . .

Runner * wsk = new Runner;
. . .
wsk->show();
. . .
```

Stan obiektu

- *Stan* (ang. *state*) obiektu w danym momencie opisany jest wartością jego pól.
- Pola obiektu nazywane są również *atrybutami* lub *składowymi danych*.
- Wartości pól oczywiście mogą się zmieniać w czasie wykonania programu, oznacza to, że obiekt zmienia swój stan.

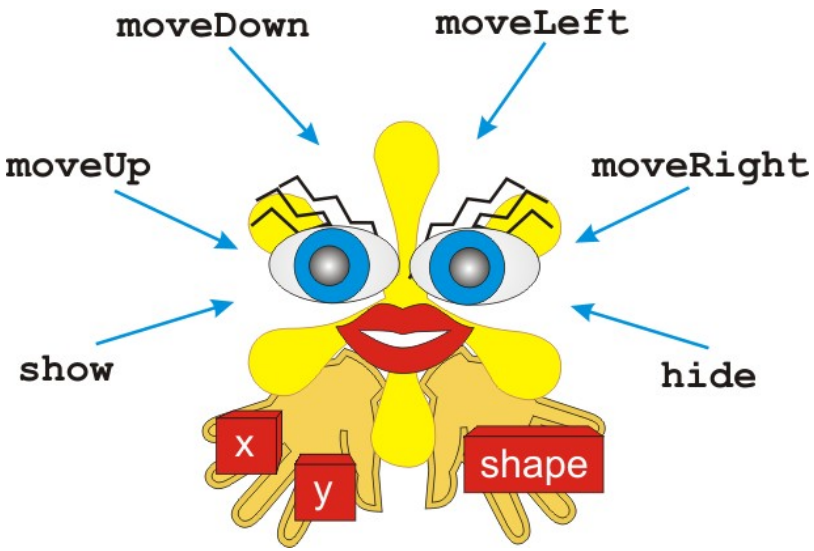
```
Runner R;

R.x = 1;
R.y = 1;
R.shape = '@';
R.show();
. . .

R.x = 5;
R.y = 5;
R.shape = '$';
R.show();
. . .
```



Zdolność obiektu do działania

- *Zdolność do działania* (ang. *behavior*) opisuje *operacje*, które obiekt może *wykonać*.
- Operacje są realizowane poprzez wykonanie *metod* zwanych w C++ *funkcjami składowymi*.
- Metody są częścią obiektu i mogą być aktywowane wtedy, gdy obiekt otrzyma *komunikat*, zgodny z nazwą metody, która ma być wykonana.
- Wykonanie metod zwykle *zmienia stan* obiektu poprzez zmianą *wartości jego pól*.

```
Runner R;
. . .
R.show();
R.moveLeft();
R.moveLeft();
R.moveUp();
. . .
R.hide();
```



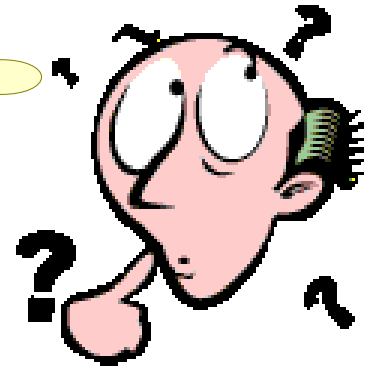
Najważniejsze cechy obiektowego podejścia do programowania

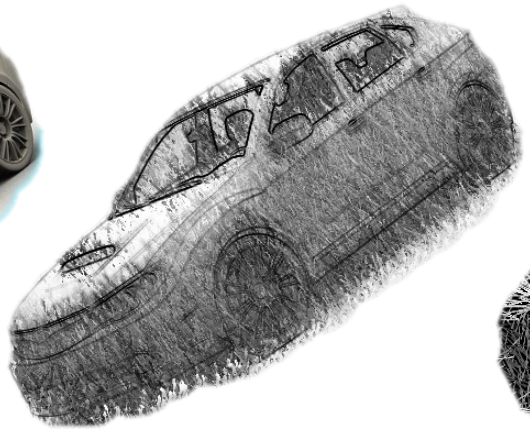
- Abstrakcja (ang. *abstraction*).
- Hermetyzacja (ang. *encapsulation*).
- Dziedziczenie (ang. *inheritance*).
- Polimorfizm (ang. *polymorphism*)

Abstrakcja – w sztuce to *abstrakcjonizm*..., po co w programowaniu?

Czy
muszę wiedzieć wszystko?

Czy potrzebne są wszystkie
szczegóły?

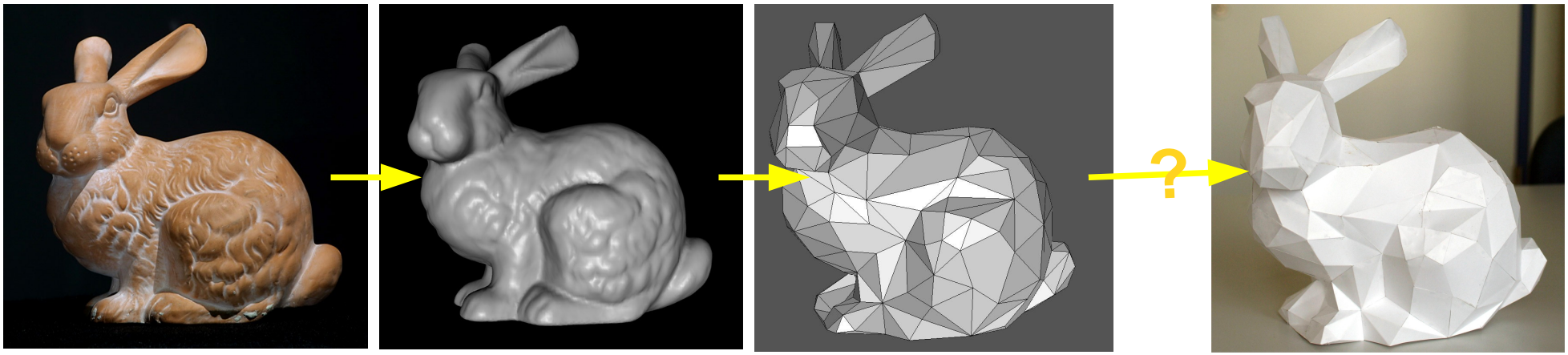




Rysunki: http://www.turbosquid.com

Abstrakcja — tworzenie komputerowych modeli

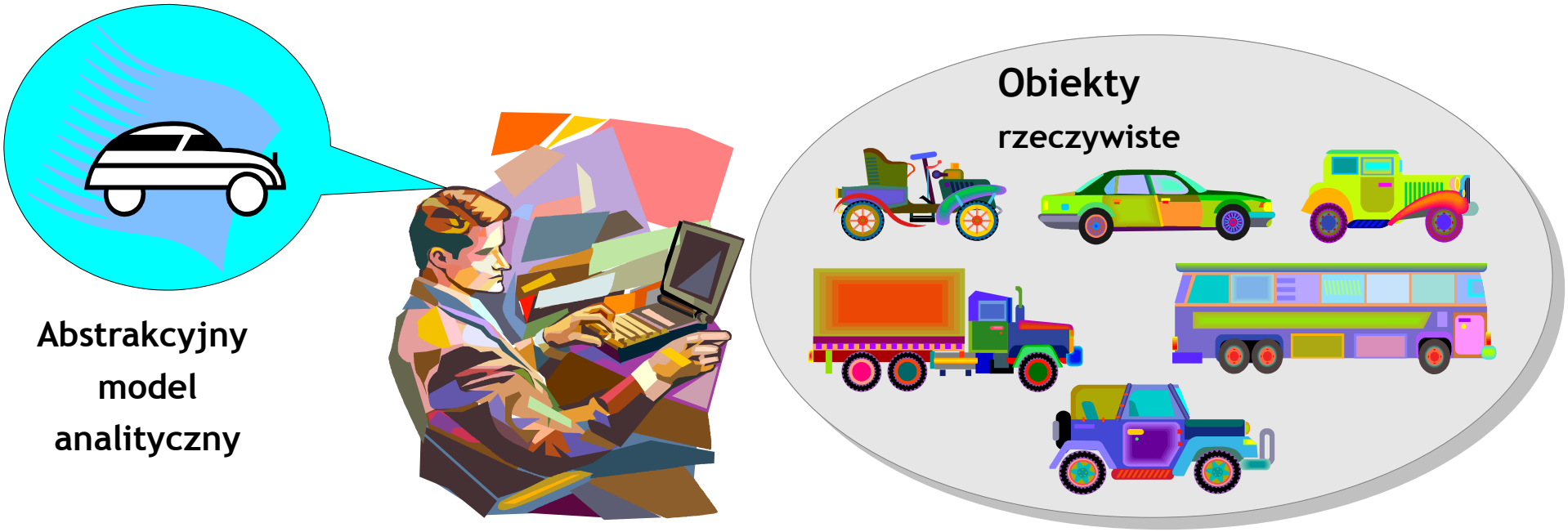
- Obiekt jest zwykle *programową abstrakcją* pewnego konkretnego bytu ze świata rzeczywistego, reprezentującego *rzecz* (obiekt fizyczny) lub *pojęcie* (obiekt konceptualny).
- W programowaniu obiekt stanowi zatem *komputerowy model* jakiegoś obiektu ze świta rzeczywistego.
- Obiekt-model zwykle jest *uproszczony* w stosunku do rzeczywistego pierwowzoru. Jest tak, ponieważ trudno *wiernie odwzorować rzeczywistość*, lub to wierne odwzorowanie *nie jest wcale potrzebne*.



The "Stanford Bunny"

Abstrakcja – tylko to co istotne

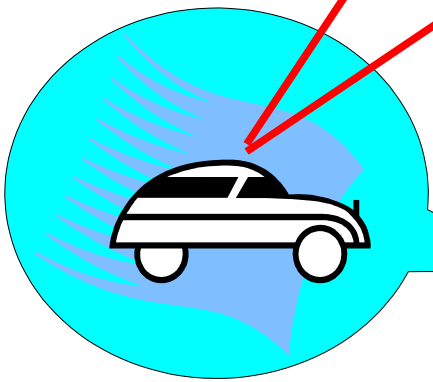
- *Abstrakcja* pozwala postrzegać modelowaną rzeczywistość *bez rozważania wszystkich szczegółów* tej rzeczywistości oraz *bez wnikania w jej wewnętrzną strukturę*.
- Stosowanie *abstrakcji* w programowaniu obiektowym prowadzi do tego, że obiekt powiniem posiadać tylko te elementy, które są *istotne dla tworzonego systemu*, elementy w danym momencie nieistotne *powinny zostać pominięte*.



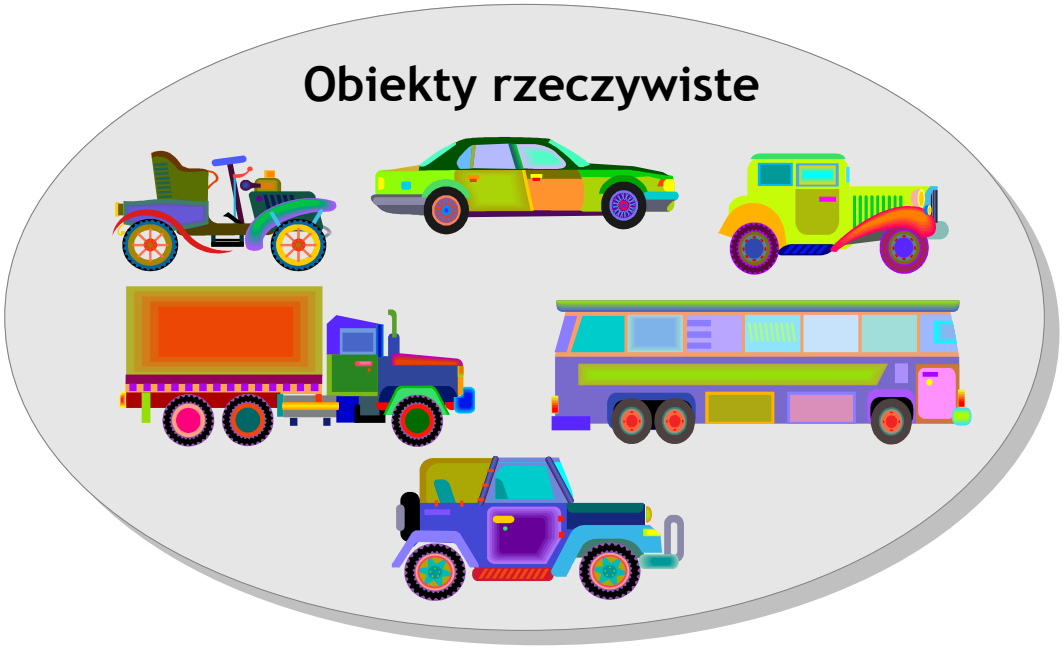
Abstrakcja — obiekt w rzeczywistości → model → obiekt w programie

Cechy	Marka Model Rok produkcji Nr rejestracyjny Kolor nadwozia Nr silnika . . .
Akcje	Podaj średnie spalanie Podaj przebieg . . .

Reprezentacja obiektowa



Abstrakcyjny model analityczny



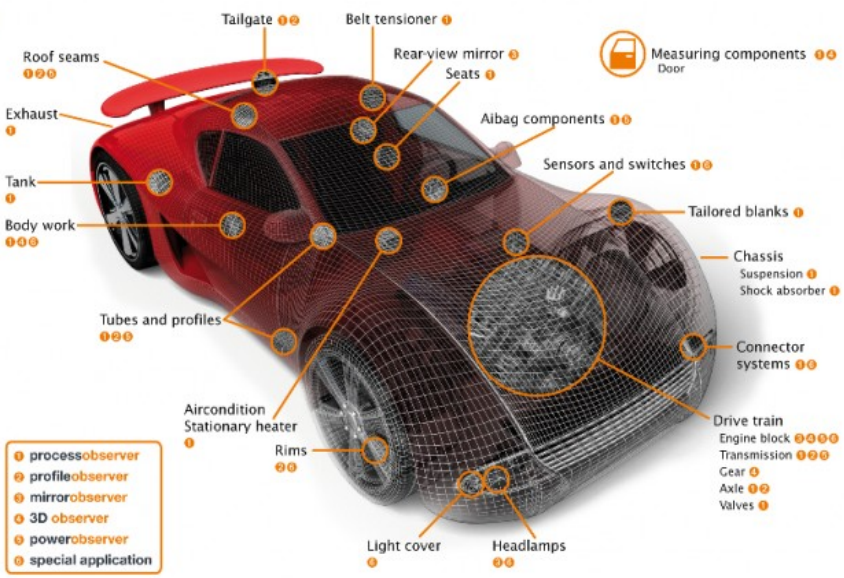
Abstrakcja — *prostota* ale nie *prostactwo*

- Abstrakcja ma dostarczyć maksymalnie *prostego* i *adekwatnego* dla problemu *opisu obiektu*.
- Obiekt ma być *prosty*, lecz nie *prostacki*.
- Stosowaniu abstrakcji w programowaniu ma towarzyszyć *antycypacja*, czyli *przewidywanie i wyprzedzenie* potencjalnych zmian *wymagań*.
- *Antycypacja* i *abstrakacja* mają dostarczać *prostego* opisu obiektu, jednocześnie *elastycznego* i *podatnego na rozwój*.

Hermetyzacja – czyli: *mam to w …*, po co w programowaniu?

Czy muszę
wiedzieć co jest w środku
aby tego używać?

Czy muszę wiedzieć jak to działa
aby tego używać?



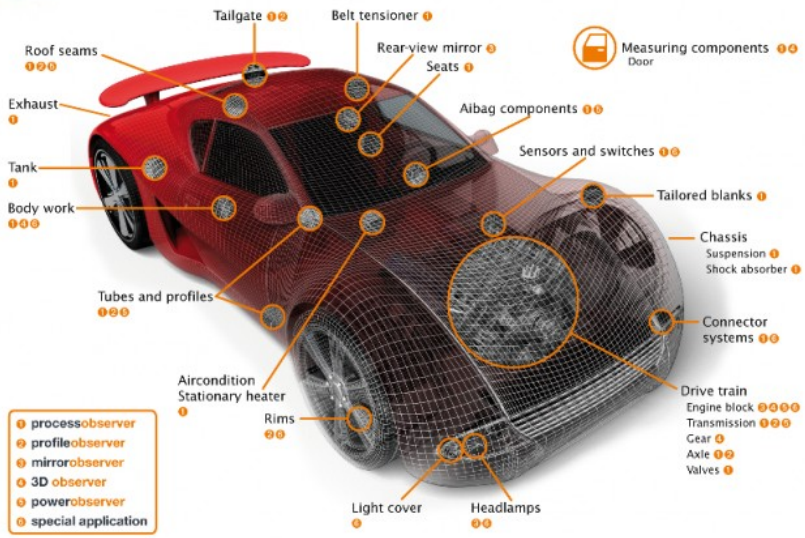
Hermetyzacja — totalny zez, czyli: *spójrz z dwóch stron…*

Nie wiem jak to działa,
i nie chcę wiedzieć.

Chcę tego używać!

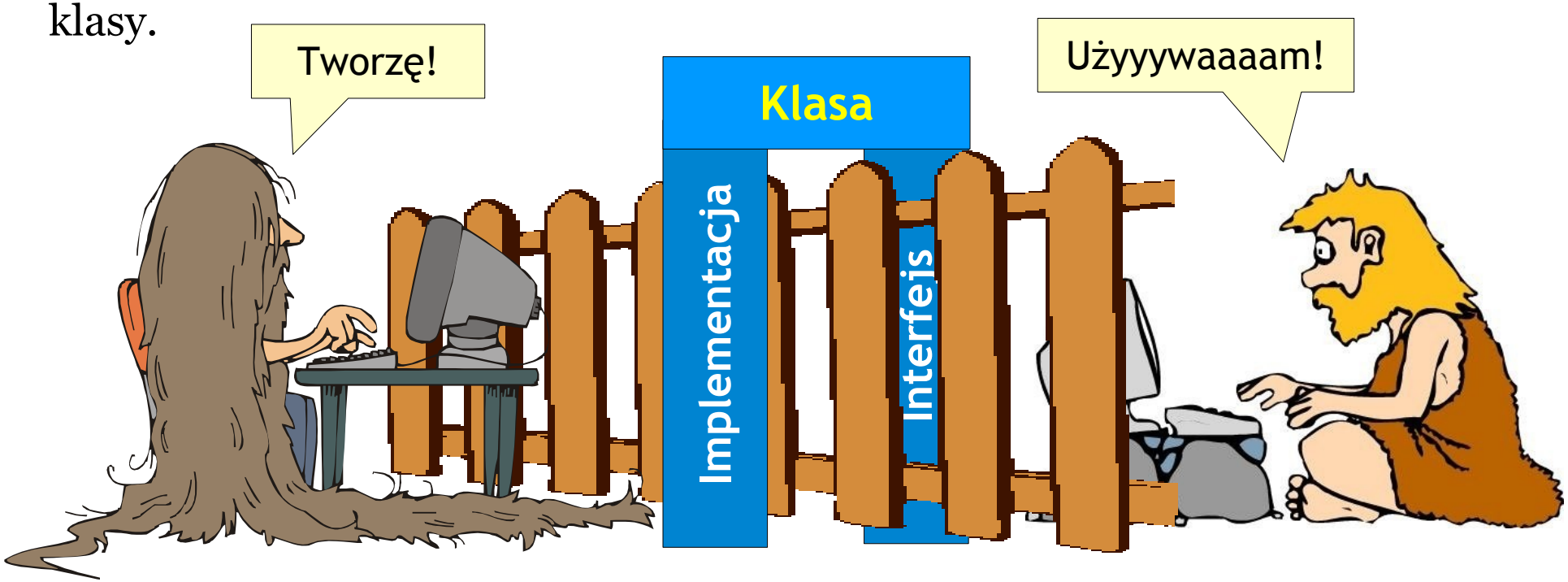
Wiem jak to działa,
ale Ty nie musisz tego wiedzieć
aby używać.

Tak, dbam o swoje interesy!



Hermetyzacja w programowaniu

- Bardzo często programista *wykorzystujący* obiekt pewnej klasy *nie pisze* jej samodzielnie, wykorzystując klasy przygotowane przez innych.
- Klasa powinna mieć jasno zdefiniowany *interfejs*, który wykorzystuje programista używający klasy.
- Szczegóły reprezentacji pól oraz implementacji metod powinny zostać przez twórcę klasy *oddzielone od interfejsu* i *niewidoczne* dla programisty używającego klasy.



Hermetyzacja

Klasa bez hermetyzacji

```
class Runner
{
  public:
    int  x, y;
    char shape;

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};
```

- Hermetyzację osiąga się to poprzez ukrycie (ang. *information hiding*) pól obiektu oraz niektórych funkcji składowych w sekcjach *prywatnych* lub *chronionych*.

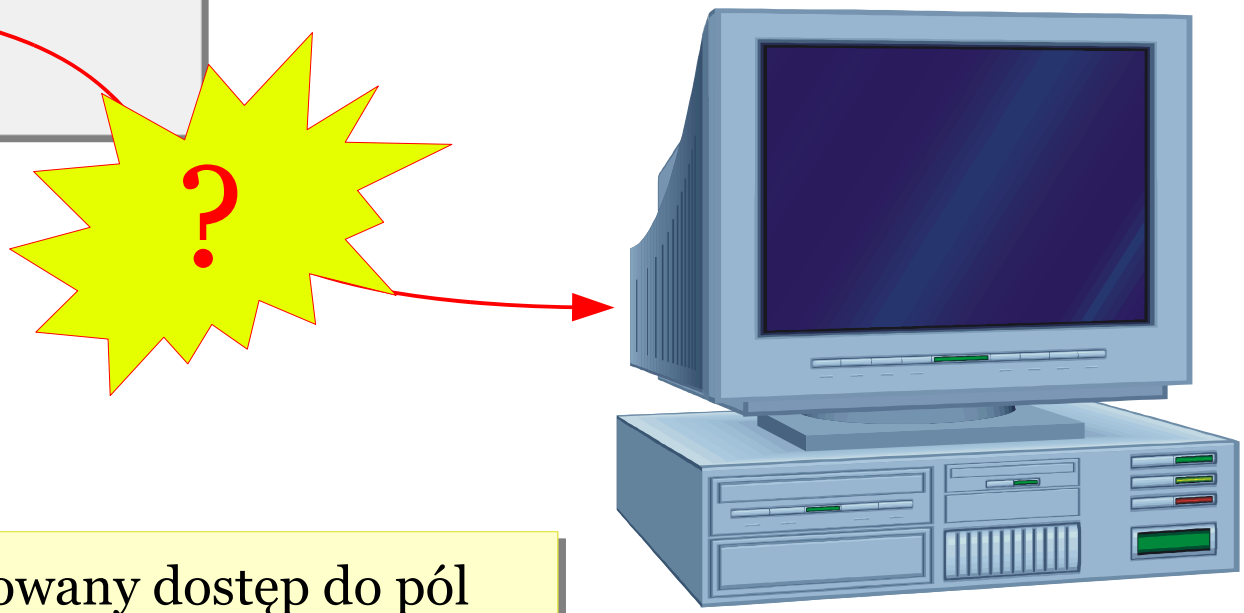
- Dzięki hermetyzacji można modyfikować wewnętrzną reprezentację klasy, sposób jej działania w sposób przeźroczysty dla programisty używającego klasy.

Niebezpieczeństwa braku hermetyzacji

```
Runner R;

R.x = -400;
R.y = 1200;

R.shape = ' ';
. . .

R.show();
```

Błędna pozycja ekranowa

Nieszczęśliwie dobrany znak reprezentujący obiekt



Bezpośredni i niekontrolowany dostęp do pól obiektu nie pozwala na sprawdzenie poprawności informacji przekazywanych obiektowi.

Ukrycie pól — dostęp za pomocą funkcji składowych

```
Runner R;

R.setX( -400 );
R.setY( 1200 );

R.setShape( ' ' );
. . .

R.show();
```

Ustalanie pozycji via funkcje składowe obiektu

Ustalanie znaku via funkcje składowe obiektu

Funkcje składowe zignorowały niepoprawne parametry i ustaliły wartości domyślne:
x = 1, y = 1, shape = '*'

Funkcje składowe mogą kontrolować poprawność danych przekazywanych obiektowi, odrzucając wartości niepoprawne, przyjmując ustalone arbitralnie wartości domyślne dla pól.

Publiczne i prywatne elementy klasy

```
class Runner
{
  public :
    void setX( int newX );
    void setY( int newY );
    void setShape( char newShape );

    int  getX();
    int  getY();
    char getShape();

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();

  private:
    int  x, y;
    char shape;

    bool isXOnScreen( int newX );
    bool isYOnScreen( int newY );
};
```

```
Runner R;

R.x = 10;
R.y = 20;
R.shape = '#';
R.isXOnScreen( 100 );
```

Interfejs klasy — sekcja publiczna

Tylko te elementy mogą być wykorzystywane przez programistę używającego klasę Runner.

```
R.setX( 10 );
R.setY( 20 );
R.setShape('#');
```

Wewnętrzne elementy klasy — sekcja prywatna

Te elementy mogą być wykorzystywane wyłącznie przez funkcje składowe klasy.

Podział publicznych funkcji składowych

```
class Runner
{
  public :
    void setX( int newX );
    void setY( int newY );
    void setShape( char newShape );

    int  getX();
    int  getY();
    char getShape();

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();

  private:
    int  x, y;
    char shape;

    bool isXOnScreen( int newX );
    bool isYOnScreen( int newY );
};
```

Modyfikatory — funkcje dokonujące modyfikacji wartości pól, zapewne w sposób kontrolowany.

Akcesory — funkcje umożliwiające kontrolowane pobieranie wartości pól.

Realizatory — funkcje realizujące właściwe dla klasy usługi.

Modyfikatory i *akcesory* mają charakter administracyjno-kontrolny, *realizatory* to esencja działania klasy.

Modyfikatory – wersja pierwsza

```
void Runner::setX( int newX )
{
  if( isXOnScreen( newX ) )
    x = newX;
  else
    x = 1; // Wartość domyślna
}
```

```
void Runner::setY( int newY )
{
  if( isYOnScreen( newY ) )
    y = newY;
  else
    y = 1; // Wartość domyślna
}
```

```
void Runner::setShape( char newShape )
{
  if( newShape > 32 && newShape < 255 )
    shape = newShape;
  else
    shape = '*'; // Wartość domyślna
}
```

Modyfikatory – wersja druga, bez ustawiania wartości domyślnych

```
void Runner::setX( int newX )
{
  if( isXOnScreen( newX ) )
    x = newX;
}
```

```
void Runner::setY( int newY )
{
  if( isYOnScreen( newY ) )
    y = newY;
}
```

```
void Runner::setShape( char newShape )
{
  if( newShape > 32 && newShape < 255 )
    shape = newShape;
}
```

Akcesory

```
int Runner::getX()
{
  return x;
}
```

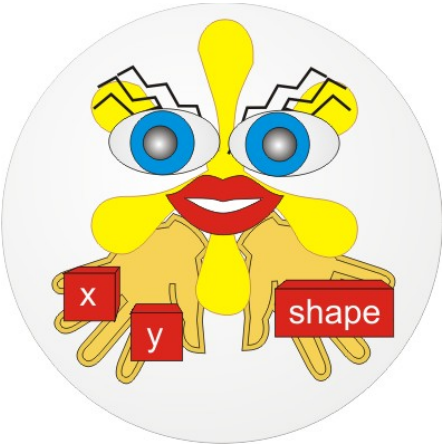
```
int Runner::getY()
{
  return y;
}
```

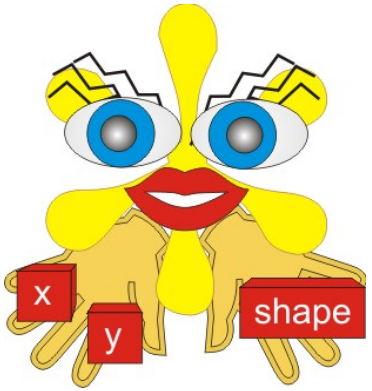
```
char Runner::getShape()
{
  return shape;
}
```

Z hermetyzacją i bez, czyli jak z polem siłowym i bez niego

```
class Runner
{
  public :
    void setX( int newX );
    void setY( int newY );
    void setShape( char newShape );

    int  getX();
    int  getY();
    char getShape();

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();

  private:
    int  x, y;
    char shape;

    bool isXOnScreen( int newX );
    bool isYOnScreen( int newY );
};
```

```
class Runner
{
  public:
    int x, y, shape;

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};
```





Jeszcze dwie cechy obiektowego podejścia do programowania...

- Dziedziczenie (ang. *inheritance*).

Osobny wykład

- Polimorfizm (ang. *polymorphism*)

Osobny wykład

Suplement – obsługa klawiatury, klawisze sterujące, Borland conio

Znaki sterujące na starszym bajcie (tzw. *scan code*), zwykłe na młodszym (notacja szesnastkowa):

```
enum KEY_CODES
{
  KEY_UP    = 0x4800,
  KEY_DOWN  = 0x5000,
  KEY_LEFT  = 0x4b00,
  KEY_RIGHT = 0x4d00,
  KEY_ESC   = 0x001b
};
```

Przykładowa implementacja funkcji **`getKey`** (biblioteka **`conio`** firmy Borland). Znak ASCII pozostawiany jest na młodszym bajcie, sterujący przesuwany jest na starszy bajt:

```
int getKey()
{
  int key = getch();
  return ( key == 0 ) ? getch() << 8 : key;
}
```

Suplement — obsługa ekranu, Borland conio

Implementacje funkcji czyszczenia ekranu oraz wyświetlania znaku na zadanej pozycji ekranu — wersje wykorzystujące bibliotekę **`conio`** (kompilatory firmy Borland):

```
void clearScreen()
{
  clrscr();
}
```

```
void writeCharXY( int x, int y, char c )
{
  gotoxy( x, y );
  putch( c );
}
```

Suplement — obsługa klawiatury, Borland + MINGW i GCC

```
enum KEY_CODES
{
  #ifdef __BORLANDC__
  KEY_BLANK = 0x0000,
  #endif

  #ifdef __MINGW32_VERSION
  KEY_BLANK = 0x00e0,
  #endif

  KEY_UP    = 0x4800,
  KEY_DOWN  = 0x5000,
  KEY_LEFT  = 0x4b00,
  KEY_RIGHT = 0x4d00,
  KEY_ESC   = 0x001b
};;
```

```
// Wymaga włączenia pliku conio.h
int getKey()
{
  int key = getch();
  return ( key == KEY_BLANK ) ? getch() << 8 : key;
}
```

Suplement – czyszczenie ekranu, Borland conio + MINGW i GCC

```
// Wymaga włączenia pliku conio.h (Borland) lub windows.h (MINGW i GCC)
void clearScreen()
{
  #ifdef __BORLANDC__
  clrscr();
  #endif

  #ifdef __MINGW32_VERSION
  COORD leftTop = { 0, 0 };
  CONSOLE_SCREEN_BUFFER_INFO consoleInfo;
  DWORD writtenItems;
  int numOfCells = 80 * 25;

  HANDLE consoleHandle = GetStdHandle( STD_OUTPUT_HANDLE );

  if( GetConsoleScreenBufferInfo( consoleHandle, &consoleInfo ) )
    numOfCells = consoleInfo.dwSize.X * consoleInfo.dwSize.Y;

  FillConsoleOutputAttribute( consoleHandle, 0xf, numOfCells,
                              leftTop, &writtenItems );

  FillConsoleOutputCharacter( consoleHandle, ' ', numOfCells,
                              leftTop, &writtenItems);

  #endif
}
```

Suplement — wypisywanie znaku, Borland conio + MINGW i GCC

```
// Wymaga włączenia pliku conio.h (Borland) lub windows.h (MINGW i GCC)
void writeCharXY( int x, int y, char c )
{
  #ifdef __BORLANDC__
  gotoxy( x, y );
  putch( c );
  #endif

  #ifdef __MINGW32_VERSION
  COORD cursorPos;
  DWORD written;

  cursorPos.X = x - 1;
  cursorPos.Y = y - 1;

  SetConsoleCursorPosition( GetStdHandle( STD_OUTPUT_HANDLE ), cursorPos );
  WriteConsole( GetStdHandle( STD_OUTPUT_HANDLE ), &c, 1,  &written, 0 );
  #endif
}
```

Suplement — wypisywanie napisu, Borland conio + MINGW i GCC

```
// Wymaga włączenia pliku conio.h (Borland) lub windows.h (MINGW i GCC)
void writeStrXY( int x, int y, char s[] )
{
  #ifdef __BORLANDC__
  gotoxy( x, y );
  cputs( s );
  #endif

  #ifdef __MINGW32_VERSION
  COORD cursorPos;
  DWORD written;
  HANDLE consoleHandle = GetStdHandle( STD_OUTPUT_HANDLE );

  cursorPos.X = x - 1;
  cursorPos.Y = y - 1;

  SetConsoleCursorPosition( consoleHandle, cursorPos );
  WriteConsole( consoleHandle, s, strlen( s ),  &written, 0 );
  #endif
}
```

Suplement — wypisywanie liczby *int* i *double*, Borland conio+MINGW i GCC

```
// Wymaga włączenia pliku nagłówkowego cstdio lub stdio.h
// Wykorzystuje wcześniej zdefiniowaną funkcję writeStrXY
void writeIntXY( int x, int y, int i )
{
  char s[ 80 ];
  sprintf( s, "%d", i );
  writeStrXY( x, y, s );
}

void writeDoubleXY( int x, int y, double d )
{
  char s[ 80 ];
  sprintf( s, "%g", d );
  writeStrXY( x, y, s );
}
```

Suplement – wyłączenie kursora tekstowego

```
// Wymaga włączenia pliku conio.h (Borland) lub windows.h (MINGW i GCC)
void cursorOff()
{
  #ifdef __BORLANDC__
  _setcursortype( _NOCURSOR );
  #endif

  #ifdef __MINGW32_VERSION
  CONSOLE_CURSOR_INFO cursorInfo;
  HANDLE consoleHandle = GetStdHandle( STD_OUTPUT_HANDLE );
  GetConsoleCursorInfo( consoleHandle, &cursorInfo );
  cursorInfo.bVisible = false;
  SetConsoleCursorInfo( consoleHandle, &cursorInfo );
  #endif
}
```

Suplement – włączenie kursora tekstowego

```
// Wymaga włączenia pliku conio.h (Borland) lub windows.h (MINGW i GCC)
void cursorOn()
{
  #ifdef __BORLANDC__
  _setcursortype( _NORMALCURSOR   );
  #endif

  #ifdef __MINGW32_VERSION
  CONSOLE_CURSOR_INFO cursorInfo;
  HANDLE consoleHandle = GetStdHandle( STD_OUTPUT_HANDLE );
  GetConsoleCursorInfo( consoleHandle, &cursorInfo );
  cursorInfo.bVisible = true;
  SetConsoleCursorInfo( consoleHandle, &cursorInfo );
  #endif
}
```