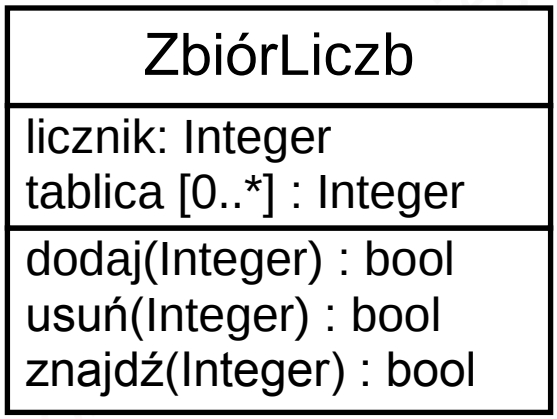
Klasy generyczne

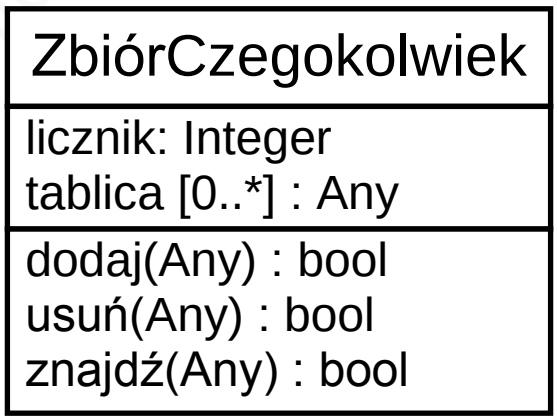
Konstrukcja uniwersalnych klas

Problemem do rozwiązania jest tworzenie uniwersalnego kodu, który w ten sam sposób - te same algorytmy – przetwarza różne typy danych.

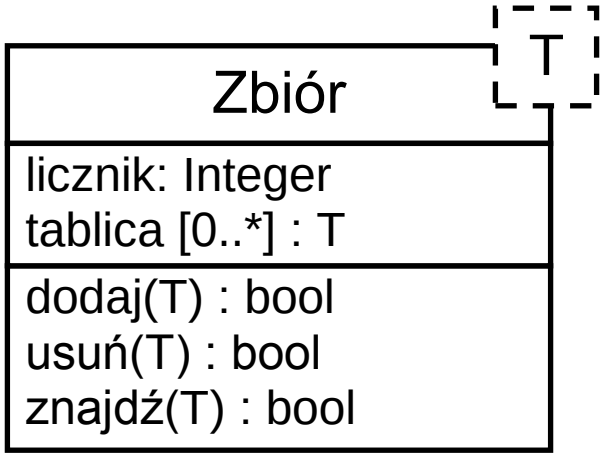
Specyfikacja **wąsko specjalizowanej klasy** *ZbiórLiczb*. Dany zbiór może zawierać jedynie liczby typu *Integer*. Klasa **weryfikuje poprawność** elementów zbioru.



Specyfikacja **uniwersalnej klasy** *Zbiór*. Dany zbiór może posiadać elementy dowolnego typu. Klasa **nie weryfikuje typów** elementów zbioru!!!



Specyfikacja **uniwersalnej** generycznej klasy *Zbiór*. Dany zbiór może zawierać jedynie elementy określonego typu, który **jest weryfikowany** przez klasę.



Uniwersalność, a poprawność przetwarzania programów

Wąsko specjalizowane klasy

- dzięki kontroli typów - duża niezawodność programów
- mała użyteczność modułów programowych
- redundancja kodu

```
ZbiórLiczb zl; //osobne klasy dla każdego typu elementów
ZbiórOsób zo;
ZbiórNnapisów zn;
...
zl.add(1250);
zo.add(kowalski);
zn.add("Ala ma kota");
zl.test(3200);
zo.test(127); // wykrycie błędu, niepoprawny typ danej
```

Klasy uniwersalne

- błędy w czasie wykonania, w wyniku braku kontroli typów
- większa użyteczność modułów programowych
- brak redundancji kodu, bardziej złożony kod

```
class Zbiór {
public:
    int add(void *element);
    int remove(void *element);
    int test(void *element);
};
Osoba kowalski("Kowalski");
Figura trójkąt(28, 15, 23);
...
Zbiór pracownicy;
Pracownicy.add(kowalski);
Pracownicy.add(trójkąt); //potencjalne błędy czasu wykonania
```

Statycznie i dynamicznie typowane języki programowania

Bezpieczeństwo: języki statycznie typowane

Elementami kolekcji są obiekty ściśle określonego typu

```
class SortedCollectionOfInteger {
protected:
    unsigned rozmiar;
    int tab[ ];
public:
    SortedCollectionOfInteger( );
    add(int el);
    ...
};
class SortedCollectionOfFloat {...};
class SortedCollectionOfString {...};

SortedCollectionOfFloat scf(10);
SortedCollectionOfString scs(10);
scf.add(12.495);   // kontrola typu
scs.add("Ala ma kota"); // kontrola typu
```

Elastyczność: języki dynamiczne typowane

Elementami kolekcji są obiekty dowolnego typu

```
napis ← String new: 'Boguś jest OK'.
liczba ← 3.1415.
tablica  ← #(1, 'dwa', $3).
kolekcja  ← SortedCollection new.
kolekcja add: napis. "brak kontroli typów"
kolekcja add: liczba.
kolekcja add: tablica
```

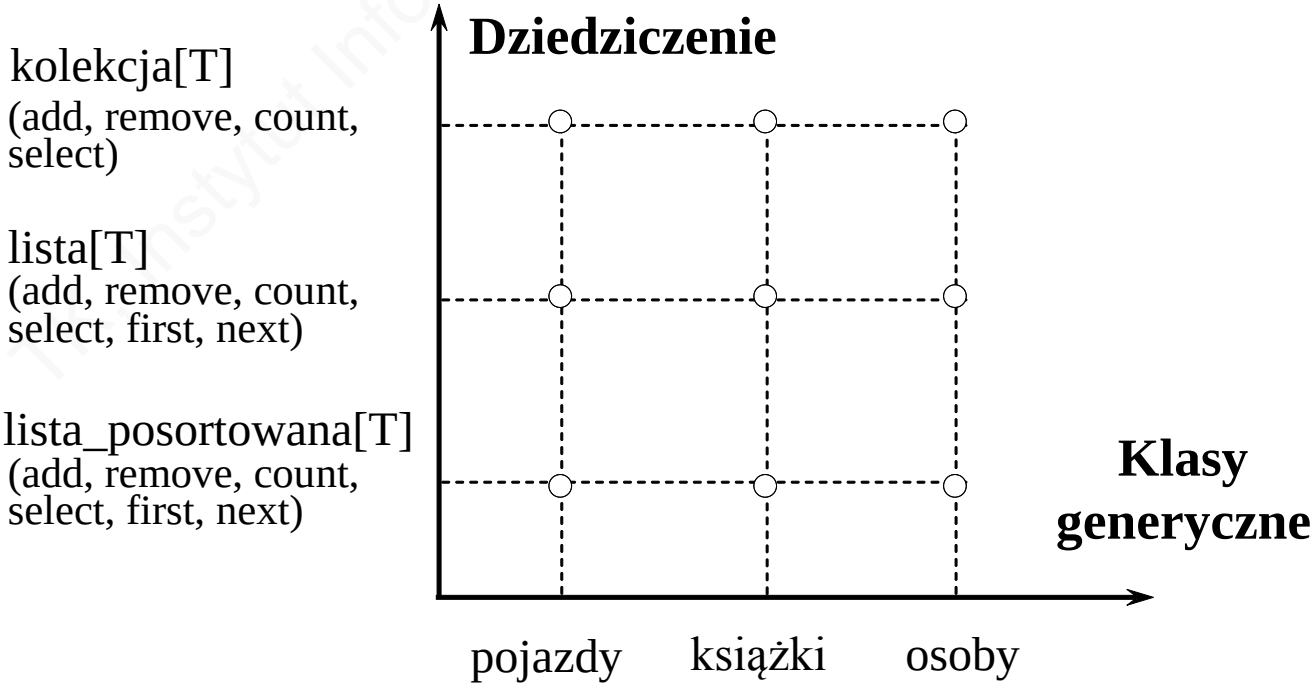
Klasy generyczne

Klasy generyczne są to **klasy o parametryzowanych typach danych**. **Klasy generyczne** posiadają kompletną implementację, jednak **nie definiują typów danych** wykorzystanych w tej implementacji. Klasy generyczne pozwalają na jednoczesne osiągnięcie elastyczności i bezpieczeństwa programów komputerowych. Pozwalają one tworzyć uniwersalne klasy o statycznie typowanych danych.

```
                          class ListaLiczba
class Lista [T]   ≡       class ListaOsób
                          class ListaNapisów
```

T- parametryzacja typu

Dziedziczenie (generalizacja klas) ↔ Parametryzacja typów (klasy generyczne)



Klasy generyczne - Java

Klasa generyczna jest klasą, której definicja zawiera **sparametryzowane typy danych**.

Zaimplementowany poniżej stos będzie poprawnie obsługiwał stosy jednorodnych elementów dowolnego typu podstawionego pod parametr T.

```
class Stos<T> { // Stos jest klasą generyczną
  private int index = 0;
  public static final int max = 10;
  private T[] data; // tablica elementów typu T

  public Stos(int rozmiar) {
    data = (T[])new Object[max];
  }
   public void push(T obj) {//wstaw element typu T
      data[index++] = obj;
  }

  public T pop() {  // pobierz element typu T
    return data[--index];
  }
}
```

Tworzenie wystąpień klas generycznych

Klasa generyczna pozwala tworzyć różniące się między sobą wystąpienia. Tworząc wystąpienie klasy generycznej należy podać wartość parametru typu.

```
// wartość parametru T równa int
Stos<Integer> si = new Stos<Integer>(12);
// wartość parametru T równa float
Stos<Float> sf = new Stos<Float>(50);
// wartość parametru T równa String
Stos<String> ss = new Stos<String>(100);
// wartość parametru T równa Point
Stos<Point> sp = new Stos<Point>(100);

si.push(17);
si.push(21);
sf.push(934.128F);
ss.push("Potężne kaczory");
ss.push("Brzydkie kaczątko");
Point p1 = new Point(2.13F, 1.47F);
sp.push(p1);
float f = sf.pop();
```

Kontrola typów

Klasy generyczne gwarantują statyczne typowanie danych. Kompilator będzie weryfkował poprawność elementów wstawianych na generyczny stos.

```
Stos<Integer> si = new Stos<Integer>();
Stos<Point> sp = new Stos<Point>();
Stos<Figure> sf = new Stos<Figure>();

int i, j;
Point p;
Figure f;
Wielokąt w;
Czworokąt c;
...
sf.push(f);          // OK
sf.push(908);        // kontrola typów - błąd kompilacji
int i = sf.pop(); // kontrola typów - błąd kompilacji
```

Dopuszczalne są polimorficzne podstawienia pod zmienne typu.

```
sf.push(w);    // OK; wielokąt jest podtypem figury
sf.push(c);    // OK; czworokąt jest podtypem figury
f = sf.pop(); // OK;
c = sf.pop(); // błąd typu, nie każda figura jest czworokątem
```

Ograniczona generyczność klas

W pewnych sytuacjach jest niezbędne odwoływania się do określonej funkcjonalności wystąpień sparametryzowanych typów danych.

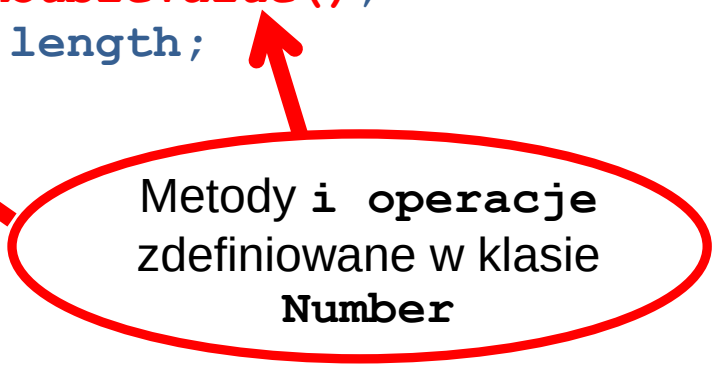
Czy implementacja poniższej klasy generycznej jest poprawna dla dowolnego typu parametru?

```
class Stos<T> {
  private int index = 0;
  private T[] data;
  public Stos(int rozmiar) {
    data = (T[])new Object[rozmiar]; }

   public void push(T obj) {
      data[index++]=obj; }

   public T pop() {return data[--index]; }

  double average() {
    double sum = 0.0;
    for(int i=0; i < data.length; i++)
    // odwołanie do funkcjonalności elementów
    // tylko elementy, które można dodawać
      sum += data[i];
    return sum / data.length;
  }
}
```

Ograniczona generyczność klas

Dla zagwarantowania poprawności przetwarzania, wartości parametrów aktualnych klasy generycznej muszą być ograniczone do klas o pożądanej funkcjonalności.

```
class Stos<T extends Number> {
  private int index = 0;
  private T[] data;
  ...
  double średnia() {
    double sum = 0.0;
    // odwołanie do funkcjonalności zdefiniowanej
    // w klasie Number (doubleValue())
    for(int i=0; i < data.length; i++)
      sum += data[i].doubleValue();
    return sum / data.length;
  }
}
```



Dynamiczne wiązanie wystąpień klas generycznych (Wildcard Type)

Co jest nadklasą wszystkich wystąpień klasy generycznej?

```
Stos<String> ss = new Stos<String>();
Stos<Integer> si = new Stos<Integer>();
Stos<Osoba> so = new Stos<Osoba>();
Stos<Object> sob = new Stos<Object>();

sob = ss; // błąd
sob = si; // błąd
sob = so; // błąd
```

Nadklasą jest klasa o nieokreślonym typie argumentu:

```
Stos<?>
```

Dzięki temu są możliwe podstawienia polimorficzne dla wystąpień klas generycznych.

```
Stos<?> sa = new Stos<Osoba>();
```

Nieokreślony typ argumentu klasy generycznej może być ograniczony do wybranej podhierarchii klas.

```
Stos<? extends Shapes> ss = new Stos<Circle>();
```

Wymazywanie typu

W kodzie wynikowym programów napisanych w języku Java, typ generyczny jest zastępowany – wymazywany (ang. type eraser).

- Nieograniczony symbol typu ***T*** jest zastępowany przez typ ***Object***, a nie przez wartość sparametryzowanego typu.

```
class Stos<T> {...}
...
Stos<Osoba> so = new Stos<Osoba>();
```

  Parametrowi ***T*** jest przypisywana wartość ***Object***, nie ***Osoba***.

- Ograniczony symbol typu ***T*** *extends* ***S*** jest zastępowany przez typ ***S***.

```
class Stos<T extends Number> {...}
...
Stos<Float> so = new Stos<Float>();
```

  Parametrowi ***T*** jest przypisywana wartość ***Number***, nie ***Float***.

W kodzie wynikowym klasy generyczne są zwykłymi klasami. Ten sam kod jest współdzielony przez wszystkie wystąpienia klasy generycznej.

Wymazywanie typu

To kompilator jest odpowiedzialny za poprawną weryfikację typów danych, a nie kod wynikowy. Wystąpienia klas generycznych, nie przechowują informacji o typie danych.

```
Stos<Float> sf = new Stos<Float>(50);
Stos<String> ss = new Stos<String>(100);

sf.push(934.128F);
ss.push("Potezne kaczory");
ss.push("Brzydkie kaczatko");

Stos<?> gs = ss; // podstawienie polimorficzne
Stos<Float> ns = (Stos<Float>)o; // brak błędu
ns.push(934.128F); // niemożność weryfikacji typu
float f = ns.pop(); // poprawne
f = ns.pop(); // błąd rzutowania String na float
              // to jest błąd czasu wykonania
```

Ograniczenia mechanizmu generyczności

Nagłówki metod zawierających parametry typu muszą być rozróżnialne dla wszystkich wartości parametrów.

```
class KlasaGeneryczna<X,Y> {
  X ob1;
  Y ob2;
    // Wymazywanie typu spowoduje utworzenie
    // dwóch identycznych metod
  void set(X o) { // set(Object o)
    ob1 = o;
  }

  void set(Y o) { // set(Object o)
    ob2 = o;
  }
}
```

Kod klasy generycznej nie może zawierać wywołań konstruktorów sparametryzowanych typów danych. Wystąpienie klasy generycznej zawiera tylko jeden kod.

```
class KlasaGeneryczna2<X> {
  X ob;
  KlasaGeneryczna2() {
    ob = new X(); // Tylko jeden konstruktor
  }
}
```

Ograniczenia mechanizmu generyczności

Kolejne ograniczenia dotyczą metod i zmiennych klasowych klas generycznych. Klasa generyczna ma jeden zestaw zmiennych i metod niezależnie od liczby wystąpień tej klasy o różnej wartości sparametryzowanych typów.

```
public class KlasaGeneryczna3<X> {
//  Atrybuty klasy generycznej nie są generyczne
//  Jest tylko jedna zmienna ob
   static X ob;

//  Metody klasy generycznej nie są generyczne
//  Jest tylko jedna wersja metody getob()
   static X getob() {
     return ob;
   }
```

Uwaga:

Język Java umożliwia generowanie statycznych metod generycznych w klasach niegenerycznych. Metody te mogą być używane jako generyczne funkcje nie powiązane z klasami obiektów.

Szablony klas – C++

Szablon klasy opisuje grupę potencjalnych klas zgodnych z tym szablonem.

Definicja szablonu:

```
template <class T> class Tablica {
protected:
    unsigned rozmiar;
    T *tp;
public:
    Tablica (unsigned roz = 12);
    T &operator[](unsigned indeks)
           {return *(tp + indeks);}
    ~Tablica ( ) { delete [] tp; }
};
template <class T> Tablica<T>::
           Tablica ((unsigned roz) {
    tp = new T[rozmiar = roz];
    for (int i=0; i<rozmiar; i++)
       *(tp+i) = 0;
}
```

Powyższy kod źródłowy nie zostanie skompilowany. Kompilator nie posiada informacji wystarczających dla utworzenia kodu wynikowego dla szablonu.

Definicja wystąpień szablonu

Definicje wystąpień szablonu są rozwijane przez kompilator do postaci źródłowej (jedna wersja dla każdej wartości parametru aktualnego typu !!!). Może to prowadzić to do nadmiernego rozrastania kodu wynikowego. Jednak wydajność kodu jest taka sama jak dla klas nie-generycznych.

```
Tablica<char> ct(25); // kod programisty
// kod źródłowy generowany podczas kompilacji
Tablica<char>::Tablica (unsigned roz) {
   tp = new char[rozmiar = roz];
   for (int i=0; i<rozmiar; i++)
   *(tp+i) = 0; }

Tablica<double> dt(1024); // ==>
Tablica<double>::Tablica (unsigned roz)
{
   tp = new double [rozmiar = roz];
   for (int i=0; i<rozmiar; i++)
   *(tp+i) = 0; }

ct[0] = 'a'; // ==>
char& operator[](unsigned indeks) {
return *(tp + indeks);}

dt[0] = 1034905.79; // ==>
double& operator[](unsigned indeks) {
return *(tp + indeks);}
```

Implementacja szablonu klasy

Język C++ nie wspiera ograniczonej generyczności klas. Możliwe jest za to implementowanie przez programistę kodu dla poszczególnych wartości parametrów szablonu.

```
template <class T> class Tablica {
protected:
    unsigned rozmiar;
    T *tp;
public:
    ...
    Tablica& sortuj( );
};
template<class T> Tablica& Tablica<T>::sortuj ()
{
    for (int i=0; i<rozmiar-1; i++)
        for (int j=rozmiar-1; i < j; j--)
            if ( tp[j] < tp[j-1] ) {
                T pom = tp[j];
                tp[j] = tp[j-1];
                tp[j-1] = pom; }
}
```

Implementacja metody `sortuj()` dla wystąpienia szablonu dla parametru typu `char*`:

```
Tablica& Tablica<char*>::sortuj () {
    for (int i=0; i<rozmiar-1; i++)
        for (int j=rozmiar-1; i < j; j--)
            if ( strcmp(tp[j], tp[j-1] )) < 0 {
                T pom = tp[j];
                tp[j] = tp[j-1];
                tp[j-1] = pom; }
}
```

Inne parametry szablonów

W języku C++ klasy wzorce mogą być parametryzowane nie tylko za pomocą typów danych. Parametrami mogą być również dowolne inne zmienne.

W poniższym przykładzie dodatkowym parametrem klasy wzorca jest liczba reprezentująca wielkość tablicy.

```
template
    <class X, int rozmiar> class Tablica {
protected:
    X tp[rozmiar];
public:
    Tablica ( );
    X& operator[](unsigned indeks) {
        return tp[indeks];}
};

Tablica<char, 1024> t1; // są to definicje  dwóch
Tablica<char, 24> t2;   //  różnych klas
Tablica<Complex, 24> t3;
Tablica<char, 24> t4;

t2 = t3;      // błąd typu
t2 = t4;      // OK
t1 = t2;      // błąd typu
```

Generyczne metody i zmienne klasy w języku C++

Ponieważ implementacja szablonów klas polega na kreowaniu osobnych klas dla wszystkich różnych wartości parametru typu, w języku C++ można korzystać z generycznych zmiennych i metod klasy (nie tylko obiektów). Dla każdej klasy będącej implementacją szablonu, jest tworzona osobna kopia generycznych zmiennych i metod klasy.

```
template <class T>class Obiekt {
private:  T typ; int wielkosc;
public: static int l_Ob;
   Obiekt(int, T); T Typ() {return typ;}
static int PodajLiczbeObiektow() {return l_Ob;} };

template <class T>  int Obiekt<T>:: l_Ob = 0;
template <class T> Obiekt<T>::Obiekt(int w, T t) {
   ++ l_Ob;  typ = t;  wielkosc = w;}

int main() {
Obiekt<char> o1(7,'K');
Obiekt<char> o2(1,'M');
Obiekt<float> o3(7,5.5);
int i=Obiekt<char>::PodajLiczbeObiektow();
int j=Obiekt<float>::PodajLiczbeObiektow();
cout<<"char = "<<i<<" , float = "<<j;
}
```

Szablony funkcji

Definicja szablonu funkcji:

```
Template<class x25> x25 max(x25 a, x25 b)
    { return a > b ? a : b; }
```

Utworzenie implementacji szablonu funkcji:

```
int a, b;
char c, d;
...
int max1 = max (a, b);  // max (int, int);
char max2 = max (c, d); // max (char, char);
int max3 = max (a, c);  // błąd typu
```

Rozróżnianie przeciążonych funkcji szablonu:

```
int a, b;
short c, d;
...
int max1 = max(a, b);   // max (int, int);
short max2 = max(c, d); // max (short, short);
int max3 = max(int(c),int(d)); // max (int, int);
```

Biblioteka szablonów
Standard Template Library

Kontenery:

- Vectors

```
vector<int> V;
V.insert(V.begin(), 3);
assert(V.size() == 1 &&
     V.capacity() >= 1 &&
     V[0] == 3);
```

- Lists, Double-Ended Queues
- Maps, Multimaps
- Sets, Multisets

Algorytmy

- binary_search, sort, for_each

Iteratory

```
list<int> L;
L.push_front(3);
insert_iterator<list<int>>
   ii(L,L.begin());
*ii++ = 0;
*ii++ = 1;
*ii++ = 2;
copy(L.begin(), L.end(),
   ostream_iterator<int>(cout, " "));
```