Dzisiejszy wykład

- Wyjątki

Grupowanie wyjątków

- Wyjątki często w sposób naturalny tworzą rodziny. Oznacza to, że dziedziczenie może być użyteczne w strukturalizacji wyjątków i ich obsłudze

```
class Matherr{ };
class Overflow: public Matherr{ };
class Underflow: public Matherr{ };
class Zerodivide: public Matherr{ };
// ...

void f()
{
try   {
      // ...
   }
   catch (Overflow) {
      // handle Overflow or anything derived from Overflow
   }
   catch (Matherr) {
      // handle any Matherr that is not Overflow
   }
}
```

Grupowanie wyjątków

- Organizowanie hierarchii wyjątków może być ważne z punktu widzenia niezawodności kodu

```
void g()
{
try   {
      // ...
   }
   catch (Overflow) { /* ... */ }
   catch (Underflow) { /* ... */ }
   catch (Zerodivide) { /* ... */ }
}
```

- Bez takiej możliwości
  - łatwo zapomnieć o którymś przypadku
  - konieczna modyfikacja kodu w wielu miejscach po dodaniu wyjątku do biblioteki

Wyjątki dziedziczone

- Wyjątki są zwykle wyłapywane przez procedurę obsługi przyjmującej argument typu klasy podstawowej, a nie pochodnej
- Obowiązująca w kodzie obsługi semantyka wyłapywania i nazywania wyjątków jest identyczna z semantyką funkcji przyjmującej argumenty
  - Argument formalny jest inicjowany wartością argumentu

```
class Matherr {
 // ...
 virtual void debug_print() const { cerr << "Math error"; }
};
class Int_overflow: public Matherr {
 const char* op;
 int a1, a2;
public:
 Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
 virtual void debug_print() const { cerr << op <<´(´<< a1 <<´,´<< a2 <<´)´;}
 // ...
};
void f()
{
 try  {
        g() ;
      }
      catch (Matherr m) {
        // ...
      }
}
```

- Po wejściu do procedury obsługi zmienna *m* jest typu *Matherr* - dodatkowa informacja z klasy pochodnej jest niedostępna.

Wyjątki dziedziczone

- Aby uniknąć trwałego gubienia informacji, można użyć referencji

```
int add(int x, int y)
{
 if ( (x>0 && y>0 && x>INT_MAX-y)
    || (x<0 && y<0 && x<INT_MIN-y) )
       throw Int_overflow("+",x,y) ;
 return x+y; // x+y will not overflow
}
void f()
{
 try {
       int i1 = add(1,2) ;
       int i2 = add(INT_MAX,-2);
       int i3 = add(INT_MAX,2) ; // here we go!
}
catch (Matherr& m) {
       // ...
       m.debug_print() ;
    }
}
```

- Wywołana zostanie metoda *Int_overflow*::*debug_print*( )

Złożone wyjątki

- Nie zawsze zgrupowanie wyjątków ma strukturę drzewa. Często wyjątek należy do dwóch grup, np.:

```
class Netfile_err : public Network_err, public File_system_err{ /* ...*/ };
```

- Taki błąd mogą wyłapać funkcje obsługujące wyjątki w sieci, jak i obsługujące wyjątki w systemie plików

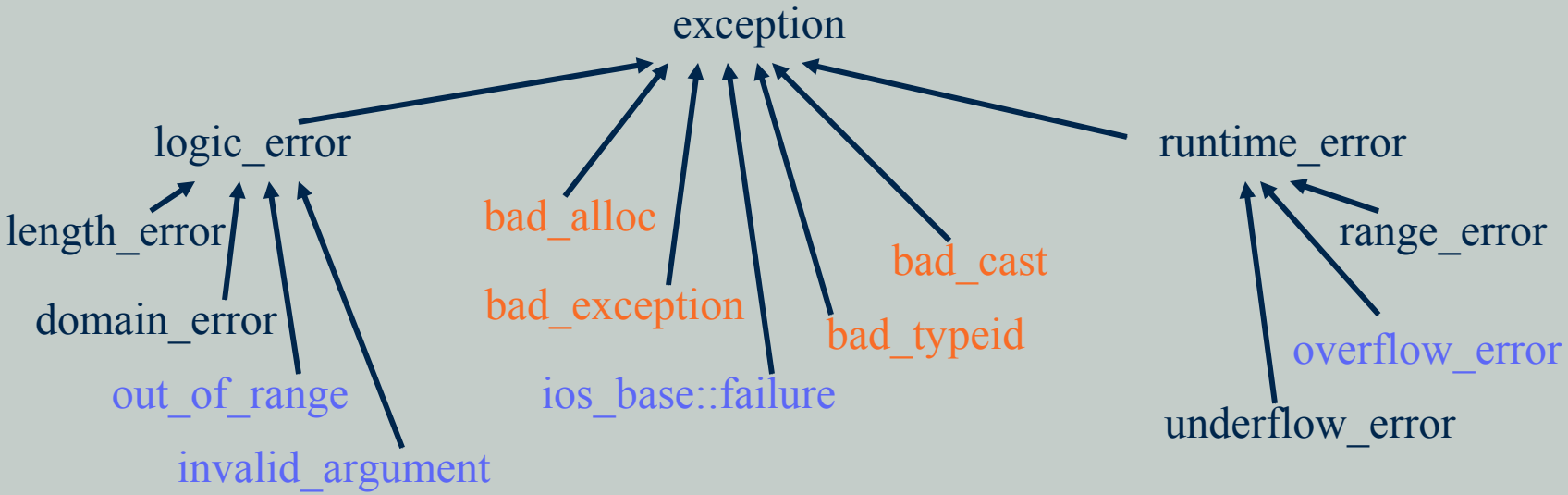
```
void f()
{
 try {
   // something
 }
 catch(Network_err& e) {
   // ...
 }
}

void g()
{
 try {
   // something else
 }
 catch(File_system_err& e) {
   // ...
 }
}
```

Standardowe wyjątki

- Wyjątki standardowe są częścią hierarchii klas zakorzenionej w klasie *exception*, przedstawionej w <*exception*>

```
class exception {
 public:
   exception() throw() ;
   exception(const exception&) throw() ;
   exception& operator=(const exception&) throw() ;
   virtual ~exception() throw() ;
   virtual const char*what() const throw() ;
 private:
 // ...
};
```



Standardowe wyjątki

- Błędy logiczne to takie, które w zasadzie można wyłapać albo przed rozpoczęciem wykonania programu, albo w wyniku testowania argumentów funkcji i konstruktorów
- Błędy czasu wykonania to wszystkie pozostałe
- Klasy wyjątków definiują we właściwy sposób wymagane funkcje wirtualne

```
void f()
try {
 // use standard library
}
catch (exception& e) {
 cout<< "standard library exception" << e.what() << ´\n´; // well, maybe
 // ...
}
catch (...) {
 cout << "other exception\n";
 // ...
}
```

- Operacje *exception* same nie zgłaszają wyjątków. Zgłoszenie wyjątku z biblioteki standardowej nie powoduje *bad_alloc*. Mechanizmy obsługi wyjątków trzymają dla siebie trochę pamięci na przechowywanie wyjątków. Można napisać taki kod, który zużyje w końcu całą pamięć dostępną w systemie, a w wyniku doprowadzi do katastrofy.

Wyłapywanie wyjątków

- Rozważmy przykład

```
void f()
{
 try {
    throw E() ;
 }
 catch(H) {
    // when do we get here?
 }
}
```

- Procedura obsługi wyjątku będzie wywołana, gdy
  - H jest tego samego typu, co E
  - H jest jednoznaczną publiczną klasą podstawową dla E
  - H i E są typami wskaźnikowymi, a dla typów, na które wskazują, zachodzi jeden z dwóch pierwszych przypadków
  - H jest referencją, a dla typu, na który wskazuje, zachodzi jeden z dwóch pierwszych przypadków
- Wyjątek jest kopiowany w chwili zgłoszenia, więc procedura obsługi dostaje kopię oryginału.
- Wyjątek może być skopiowany wiele razy, nim zostanie wyłapany.
- Nie można zgłaszać wyjątków, których nie można kopiować
- Implementacja musi zapewnić, że będzie dość pamięci na wykonanie *new* w celu zgłoszenia standardowego wyjątku braku pamięci, *bad_alloc*

Ponowne zgłoszenie wyjątku

- Zdarza się, że po wyłapaniu wyjątku procedura obsługi w gruncie rzeczy nie może do końca obsłużyć błędu
- Wówczas zazwyczaj wykonuje lokalne uporządkowanie stanu, po czym zgłasza wyjątek ponownie
- Bywa, że obsługę trzeba rozproszyć po kilku procedurach obsługi

```
void h()
{
 try {
   // code that might throw Math errors
 }
 catch (Matherr) {
   if (can_handle_it_completely) {
      // handle the Matherr
      return;
   }
   else {
      // do what can be done here
      throw; // rethrow the exception
   }
 }
}
```

- Konstrukcja *throw* bez argumentu oznacza ponowne zgłoszenie.
  - Próba ponownego zgłoszenia wyjątku, gdy żadnego nie wyłapano, powoduje wywołanie *terminate()*
- Ponownie zgłoszony wyjątek jest wyłapanym wyjątkiem, a nie jego częścią, dostępną jako *Matherr*

Wyłap każdy wyjątek

- Tak jak w funkcji wielokropek (...) oznacza "dowolny argument", tak `catch(...)` oznacza wyłapanie dowolnego wyjątku, np.:

```
void m()
{
 try {
   // something
 }
 catch (...) { // handle every exception
   // cleanup
   throw;
 }
}
```

Kolejność procedur obsługi wyjątków

- Ponieważ dziedziczony wyjątek może być wyłapany przez procedury obsługi dla więcej niż jednego typu wyjątku, więc porządek tych procedur w instrukcji *try* jest znaczący. Procedury sprawdza się po kolei

```
void f()
{
 try {
   // ...
 }
 catch (std::ios_base::failure) {
   // handle any stream io error
 }
 catch (std::exception& e) {
   // handle any standard library exception
 }
 catch (...) {
   //  handle any other exception
 }
}
```

Kolejność procedur obsługi wyjątków

- Kompilator zna hierarchię klas, może więc wyłapać wiele pomyłek logicznych

```
void g()
{
 try {
   // ...
 }
 catch (...) {
   // handle every exception
 }
 catch (std::exception& e) {
   // handle any standard library exception
 }
 catch (std::bad_cast) {
   // handle dynamic_cast failure
 }
}
```

- W podanym przykładzie nigdy nie będzie rozważany *exception*. Nawet gdybyśmy usunęli procedurę "wyłap wszystkie", to i tak nie będzie rozważany *bad_cast*, gdyż pochodzi od *exception*

Wyjątki w destruktorach

- Z punktu widzenia obsługi wyjątków, destruktor można wywołać na jeden z dwóch sposobów
  - normalne wywołanie - w wyniku normalnego wyjścia z zasięgu, wywołania *delete*
  - wywołanie podczas obsługi wyjątku - podczas zwijania stosu mechanizm obsługi wyjątku opuszcza zasięg zawierający obiekt z destruktorem
- W drugim przypadku wyjątek nie może uciec z samego destruktora, a gdy taka sytuacja jest możliwa, to traktuje się ją jako niepowodzenie mechanizmu obsługi wyjątków i wywołuje się *std::terminate()*

Wyjątki w destruktorach

- Jeśli destruktor wywołuje funkcje, które mogą zgłosić wyjątki, to może sam się ochronić, np:

```
X::~X()
try {
 f() ; // might throw
}
 catch (...) {
   // do something
}
```

- Funkcja *uncaught_exception()* ze standardowej biblioteki przekazuje true, gdy wyjątek zgłoszono, ale jeszcze nie wyłapano. Umożliwia to programiście specyfikowanie różnych działań w destruktorze, zależnie od tego, czy obiekt jest niszczony normalnie, czy w ramach zwijania stosu.

Wyjątki, które nie są błędami

- Mechanizm obsługi wyjątku można traktować jako szczególny przypadek struktury sterującej, np.:

```
void f(Queue<X>& q)
{
 try {
   for (;;) {
      X m = q.get() ; // throws 'Empty' if queue is empty
      // ...
   }
 }
 catch (Queue<X>::Empty) {
   return;
 }
}
```

- Obsługa wyjątków jest mniej strukturalnym mechanizmem niż lokalne struktury sterujące, jak *if* czy *for*, i często mniej efektywnym gdy rzeczywiście dojdzie do zgłoszenia wyjątku.
  - Wyjątków powinno używać się tylko tam, gdzie tradycyjne struktury sterujące są nieeleganckie lub nie można ich użyć

Wyjątki, które nie są błędami

- Stosowanie wyjątków jako alternatywnych metod powrotu może być elegancką techniką kończenia funkcji wyszukujących - zwłaszcza silnie rekurencyjnych funkcji wyszukujących, takich jak funkcje przeszukiwania struktur drzewiastych

```
void fnd(Tree* p, const string& s)
{
 if (s == p->str) throw p; // found s
 if (p->left) fnd(p->left,s) ;
 if (p->right) fnd(p->right,s) ;
}
Tree* find(Tree* p, const string& s)
{
 try {
   fnd(p,s) ;
 }
 catch (Tree* q) { // q->str==s
   return q;
 }
 return 0;
}
```

- Nadużywanie wyjątków do innych celów niż obsługa błędów prowadzi do nieczytelnego kodu.
- Jeżeli jest to uzasadnione, powinno się przestrzegać zasady, że obsługa wyjątków jest obsługą błędów.

Specyfikacje wyjątków

- Jako część deklaracji funkcji można zgłosić zbiór wyjątków, które funkcja może zgłosić

```
void f(int a) throw (x2, x3) ;
```

- Zapis oznacza, że *f()* może zgłosić jedynie wyjątki *x2*, *x3* oraz wyjątki wyprowadzone z tych typów, lecz żadne inne
- Próba zgłoszenia innego wyjątku z f() powoduje wywołanie *std::unexpected()*, domyślnie wywołującym *std::terminate()*, które z kolei wywołuje *abort()*

Specyfikacje wyjątków

- Zapis

```
void f() throw (x2, x3)
{
 // stuff
}
```

jest równoważny zapisowi

```
void f()
try
{
 // stuff
}
catch (x2) {throw; } // rethrow
catch (x3) {throw; } // rethrow
catch (...) {
 std::unexpected() ; // unexpected() will not return
}
```

- Jeżeli deklaracja funkcji nie zawiera specyfikacji wyjątków, funkcja może zgłosić każdy wyjątek

```
int f() ; // can throw any exception
```

- Funkcję, która nie zgłasza żadnych wyjątków, deklaruje się z listą pustą

```
int g() throw () ; // no exception thrown
```

Kontrola specyfikacji wyjątków

- Nie jest możliwe wyłapanie w czasie kompilacji każdego naruszenia specyfikacji interfejsu
- Wykonuje się większość kontroli czasu kompilacji
- Jeśli któraś deklaracja funkcji ma specyfikację wyjątków, to każda deklaracja (z definicją włącznie) musi mieć specyfikację z dokładnie tym samym zbiorem typów, np.:

```
int f() throw (std::bad_alloc) ;
int f() // error: exception specification missing
{
 // ...
}
```

- Nie wymaga się dokładnej kontroli specyfikacji wyjątków między granicami jednostek kompilacji.

Kontrola specyfikacji wyjątków

- Funkcję wirtualną można zastąpić tylko funkcją, której specyfikacja wyjątków jest co najmniej tak restrykcyjna, jak jej własna specyfikacja

```
class B {
 public:
   virtual void f() ; // can throw anything
   virtual void g() throw(X,Y) ;
   virtual void h() throw(X) ;
};
class D : public B {
 public:
   void f() throw(X) ; // ok
   void g() throw(X) ; // ok: D::g() is more restrictive than B::g()
   void h() throw(X,Y) ; // error: D::h() is less restrictive than B::h()
};
```

- Podobnie, można przypisać wskaźnik do funkcji, która ma bardziej restrykcyjną specyfikację wyjątków, na wskaźnik do funkcji z mniej restrykcyjną specyfikacją, ale nie na odwrót

```
void f() throw(X) ;
void (*pf1)() throw(X,Y) = &f; // ok
void (*pf2)()throw() = &f; // error: f() is less restrictive than pf2
void g() ; // might throw anything
void (*pf3)() throw(X) = &g; // error: g() less restrictive than pf3
```

- Specyfikacja wyjątków nie jest częścią typu funkcji i nie może zawierać jej instrukcja *typedef*

```
typedef void (*PF)() throw(X) ; // error
```

Nieoczekiwane wyjątki

- Niedbałe specyfikowanie wyjątków może prowadzić do wywołań funkcji *unexpected()*.
  - Można uniknąć takich wywołań, jeżeli starannie zorganizuje się wyjątki i wyspecyfikuje interfejsy
  - Można również przechwytywać wywołania tej funkcji i je neutralizować
- W dobrze zdefiniowanym podsystemie *Y* wszystkie wyjątki są wyprowadzone z klasy *Yerr*. Jeżeli obowiązują deklaracje

```
class Some_Yerr : public Yerr{ /* ... */ };
void f() throw (Xerr, Yerr, exception) ;
```

funkcja *f()* przekaże dowolny *Yerr* do funkcji wywołującej. W szczególności, *f()* obsłuży *Some_Yerr* poprzez przekazanie go do funkcji wywołującej. Żaden *Yerr* w *f()* nie spowoduje wywołania funkcji *unexpected()*

Odwzorowanie wyjątków

- Niekiedy strategia przerywania działania programu po napotkaniu nieoczekiwanego wyjątku jest zbyt drakońska. Wówczas tak trzeba zmodyfikować funkcję *unexpected()*, by uzyskać coś bardziej akceptowalnego
- Najprostszym sposobem jest dodanie wyjątku *std::bad_exception* z biblioteki standardowej do specyfikacji wyjątków. Funkcja *unexpected()* zgłosi wówczas *bad_exception*.

```
class X{ };
class Y{ };
void f() throw(X, std::bad_exception)
{
 // ...
 throw Y() ; // throw ``bad'' exception
}
```

Odwzorowanie wyjątków przez użytkownika

- Rozważmy funkcję I, napisaną dla środowiska niesieciowego. Załóżmy również, że *g()* zadeklarowano ze specyfikacją wyjątków, zgłosi więc ona jedynie wyjątki związane ze swoim "podsystemem Y"

```
void g() throw(Yerr) ;
```

- Teraz załóżmy, że musimy wywołać *g()* w środowisku sieciowym
  - *g()* w przypadku wystąpienia wyjątku sieciowego wywoła *unexpected()*
  - Żeby używać *g()* w środowisku rozproszonym, musimy dostarczyć kod do obsługi wyjątków sieciowych, lub napisać *g()* na nowo
  - Jeśli napisanie na nowo jest niewykonalne lub niepożądane, to możemy przedefiniować znaczenie *unexpected()*

Odwzorowanie wyjątków przez użytkownika

- Reakcję na nieoczekiwany wyjątek określa zmienna *_unexpected_handler* ustawiany przez *std::set_unexpected()* z *<exception>*

```
typedef void(*unexpected_handler)() ;
unexpected_handler set_unexpected(unexpected_handler) ;
```

- Żeby dobrze obsługiwać nieoczekiwane wyjątki, najpierw definiujemy klasę która umożliwia zastosowanie techniki "zdobywanie zasobów jest inicjowaniem" do funkcji *unexpected()*

```
class STC{ // store and reset class
 unexpected_handler old;
public:
 STC(unexpected_handler f) { old = set_unexpected(f) ; }
 ~STC() { set_unexpected(old) ; }
};
```

- Teraz definiujemy funkcję, która ma zastąpić funkcję *unexpected()*

```
class Yunexpected : Yerr{ };
void throwY() throw(Yunexpected) { throw Yunexpected() ; }
```

- *throwY()*, użyta jako *unexpected()*, odwzorowuje nieoczekiwany wyjątek w *Yunexpected*
- Wreszcie, dostarczamy wersję *g()* do użytku w środowisku sieciowym

```
void networked_g() throw(Yerr)
{
 STC xx(&throwY) ; // now unexpected() throws Yunexpected
 g() ;
}
```

- W ten sposób specyfikacja wyjątków funkcji *g()* nie jest naruszona

Odtwarzanie typu wyjątku

- Dzięki odwzorowaniu nieoczekiwanych wyjątków w *Yunexpected* użytkownik funkcji *networked_g()* wie, że dokonano odwzorowania. Nie wie natomiast, który wyjątek odwzorowano. Istnieje prosta technika pozwalająca na zapamiętanie i przekazanie tej informacji

```
class Yunexpected : public Yerr {
 public:
   Network_exception* pe;
   Yunexpected(Network_exception* p) :pe(p) { }
};
void throwY() throw(Yunexpected)
{
 try {
   throw; // rethrow to be caught immediately!
 }
 catch(Network_exception& p) {
   throw Yunexpected(&p) ; // throw mapped exception
 }
 catch(...) {
   throw Yunexpected(0) ;
 }
}
```

- Funkcja *unexpected()* nie może zignorować wyjątku i wykonać instrukcji powrotu, jeżeli spróbuje, to sama zgłosi wyjątek *bad_exception*

Niewyłapane wyjątki

- Jeśli wyjątek jest zgłoszony, ale niewyłapany, to woła się funkcję *std::terminate()*.
- Jest ona wywoływana również wtedy, kiedy mechanizm obsługi wyjątków stwierdzi, że stos jest zniszczony, i kiedy destruktor, wywołany podczas zwijania stosu spowodowanego wyjątkiem, próbuje zakończyć działanie przez zgłoszenie wyjątku.
- Reakcję na niewyłapany wyjątek określa *_uncaught_handler* ustawiany przez *std::set_terminate()* z *<exception>*

```
typedef void(*terminate_handler)() ;
terminate_handler set_terminate(terminate_handler) ;
```

- Wartością funkcji jest poprzednia funkcja przekazana do *set_terminate()*
- Domyślnie *terminate()* wywołuje *abort()*
- Zakłada się, że wywołanie *_uncaught_handler* nie powoduje powrotu do funkcji wywołującej, jeżeli spróbuje, to wywołana zostanie funkcja *abort()*

Niewyłapane wyjątki

- Od implementacji zależy, czy z powodu niewyłapanych wyjątków są wywoływane destruktory w sytuacji zakończenia programu
- Jeżeli programista chce mieć pewność, że po pojawieniu się nieoczekiwanego wyjątku będą zrobione porządki, powinien dodać do *main()* procedurę obsługi wszystkiego

```
int main()
try {
 // ...
}
catch (std::range_error)
{
 cerr << "range error: Not again!\n";
}
catch (std::bad_alloc)
{
 cerr << "new ran out of memory\n";
}
catch (...) {
 // ...
}
```

- Spowoduje to wyłapanie każdego wyjątku oprócz zgłoszonych podczas konstrukcji i destrukcji zmiennych nielokalnych. Nie ma możliwości wyłapania wyjątków zgłaszanych podczas inicjowania tych zmiennych.

Operator *new* i wyjątki

- Istnieją wersje standardowego operatora *new*, które nie zgłaszają wyjątków przy braku pamięci, ale zwracają 0, są one oznaczane dodatkowym argumentem *nothrow*

```
class bad_alloc : public exception{ /* ... */ };
struct nothrow_t {};
extern const nothrow_t nothrow; // indicator for allocation that
                                // doesn't throw exceptions
typedef void (*new_handler)() ;
new_handler set_new_handler(new_handler new_p) throw() ;
void* operator new(size_t) throw(bad_alloc) ;
void operator delete(void*) throw() ;
void* operator new(size_t, const nothrow_t&) throw() ;
void operator delete(void*, const nothrow_t&) throw() ;
void* operator new[](size_t) throw(bad_alloc) ;
void operator delete[](void*) throw() ;
void* operator new[](size_t, const nothrow_t&) throw() ;
void operator delete[](void*, const nothrow_t&) throw() ;
void* operator new (size_t, void* p) throw() { return p; } // placement
void operator delete (void* p, void*) throw() { }
void* operator new[](size_t, void* p) throw() { return p; }
void operator delete[](void* p, void*) throw() { }
```

```
void f()
{
 int* p = new int[100000] ; // may throw bad_alloc
 if (int* q = new(nothrow) int[100000]) { // will not throw exception
   // allocation succeeded
 }
 else {
   // allocation failed
 }
}
```