Obsługa wyjątków

Klasyfikacja wyjątków

- Błędy w eksploatacji programów – np. błędy użytkowników.
- Niedostępność zasobów – żądanie przydziału zasobów nie może być zrealizowane np. w wyniku braku dostępnej pamięci operacyjnej, niedostępności serwera.
- Błędy programistyczne:
  - Błędy po stronie użytkowników modułów programistycznych;
  - Błędy po stronie twórców modułów programistycznych.
- Awarie sprzętowo-programowe.

Efekty uboczne błędów

Wynikiem błędów występujących w trakcie działania modułu programowego mogą być niepożądane efekty uboczne polegające na:

- błędnym wyniku działania kodu - wynik działania programu jest niezgodny z jego specyfikacją;
- niespełnieniu zdefiniowanych niezmienników - po fiasku przetwarzania stan zasobów i danych jest niepoprawny, np. wskaźnik stosu jest niespójny z liczbą elementów na stosie;
- pozostawieniu zmienionych wartości danych - nieukończone poprawnie przetwarzanie kodu pozostawia zmieniony stan obiektów;
- niezwalnianiu zasobów – działanie modułu programowego wiązało się z przydziałem zasobów, np. pamięci operacyjnej, systemu plików, procesów, sesji programów systemowych, lub kursorów, które po wystąpieniu błędu nie zostały zwolnione.

Bezpieczeństwo kodu

Fragment kodu programu jest ***bezpieczny*** (ang. ***exception-safe***) jeżeli błędy, które wystąpią w trakcie przetwarzania tego kodu nie będą powodowały niepożądanych efektów ubocznych. Można wyróżnić pięć podstawowych typów bezpieczeństwa kodu:

1. **Odporność na błędy** (ang. *failure transparency*) – przetwarzanie kodu zawsze kończy się poprawnie.
2. **Wysoki poziom bezpieczeństwa** (ang. *strong exception safety*) – fiasko przetwarzania nie powoduje żadnych efektów ubocznych, dzięki czemu dany program może dalej działać mimo wystąpienia błędów.
3. **Podstawowy poziom bezpieczeństwa** (ang. *basic exception safety*) - fiasko przetwarzania może pozostawić zmienione wartości danych, ale zdefiniowane niezmienniki są zachowane.
4. **Minimalny poziom bezpieczeństwa** (ang. *minimal exception safety*) – fiasko przetwarzania może powodować niespełnienie zdefiniowanych niezmienników, ale zwolnione są wszystkie przydzielone zasoby.
5. **Brak bezpieczeństwa** (ang. *no exception safety*) – fiasko przetwarzania nie daje żadnych gwarancji poprawności dalszego działania programu.

Przykłady obsługi wyjątków

Jaki jest poziom bezpieczeństwa poniższej klasy?

```
template <class T> class Stack {
  unsigned nelems;
  int top;
  T* v;
public:
  void push(T);
  T pop();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  ~Stack();
};
template void Stack::push(T element) {
  top++;
  if( top == nelems ){
    T* new_buffer = new T[nelems+=10];
    if( new_buffer == 0 )
      throw OutOfMemory();
    for(int i = 0; i < top; i++)
      new_buffer[i] = v[i];
    delete [] v;
    v = new_buffer;
  }
  v[top] = element;
}
```

Przykłady obsługi wyjątków

Przykład efektów ubocznych procedury obsługi wyjątków:

```
template <class T> class Stack {
  unsigned nelems;
  int top;
  T* v;
public:
  void push(T);
  T pop();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  ~Stack();
};
template void Stack::push(T element) {
  top++;
  if( top == nelems ){
    T* new_buffer = new T[nelems+=10];
    if( new_buffer == 0 )
   // pozostawienie zmienionych i niespójnych danych
      throw OutOfMemory();
    for(int i = 0; i < top; i++)
      new_buffer[i] = v[i];
    delete [] v;
    v = new_buffer;
  }
  v[top] = element;
}
```

Cele obsługi błędów

Dwie podstawowe strategie:

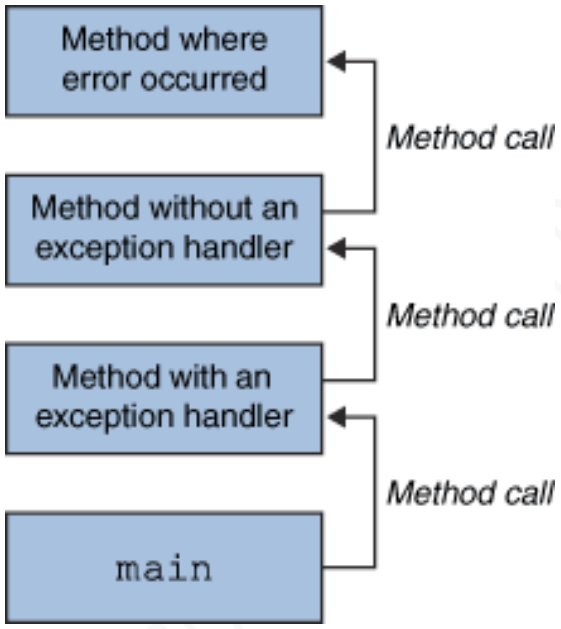
Unikanie degradacji systemu: Zadania, w których pojawiły się błędy, powinny być jak najszybszej wykryte i przerwane w celu uniknięcia propagacji błędów do innych powiązanych zadań. Zakończenie zadania powinno być poprzedzone odtworzeniem poprawnego stanu środowiska sprzętowo-programowego (*fail-fast* J. Grey, *zorganizowana panika* B. Meyer).

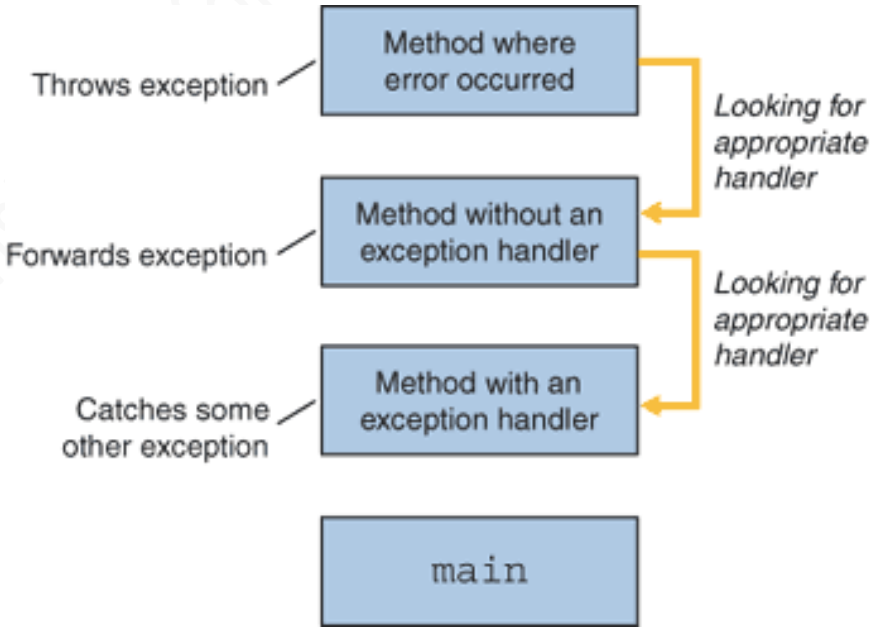
Odporność na błędy: Fiasko wykonywania jakiegoś modułu powoduje wycofanie wyników jego działania i powtórne uruchomienie tego samego lub alternatywnego modułu przy zmienionych warunkach wejściowych.

Powyższe cele mogą być osiągane łącznie. Na niższych poziomach działania programu - unikanie degradacji, na wyższych - odporność na błędy.

Architektura procedur obsługi wyjątków

Zagnieżdżone wywołania metod tworzą stos wyszukiwania procedur obsługi błędów.





Podstawowe lokalne schematy obsługi wyjątków

Budowa uniwersalnych modułów programowych wielokrotnego użytku wymaga stosowania uniwersalnych schematów obsługi błędów. Funkcjonalność obsługi wyjątków powinna zawierać następujące kroki:

1. Przywrócenie stanu spójnego przez wycofanie/kompensację wprowadzonych zmian i zwolnienie przydzielonych zasobów.
2. Zgłoszenie/przekazanie wyjątku do miejsca wywołania modułu.
3. Zmiana warunków i ponowne wywołanie kodu, który zgłosił wyjątek.

Unikanie degradacji działającego programu powinno obejmować kroki 1) i 2), dalsze działanie programu wymaga zastosowania kroków 1) i 3).

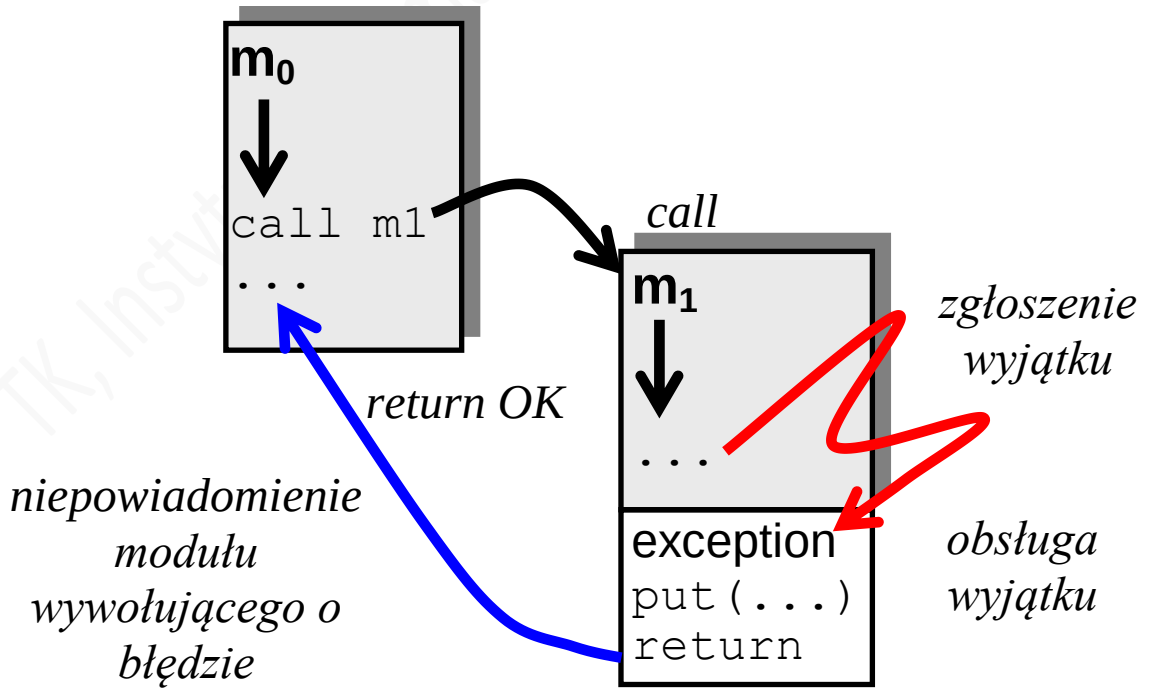
Brak wykonania kroku 1) pociąga za sobą konieczność zatrzymania pracy programu. Praca programu nie może być kontynuowana przy pozostawieniu niespójnych danych.

Za zatrzymanie działania programu powinien odpowiadać jedynie główny (startowy) moduł programu.

Jak nie należy obsługiwać wyjątków

Niezgłoszenie wyjątku.

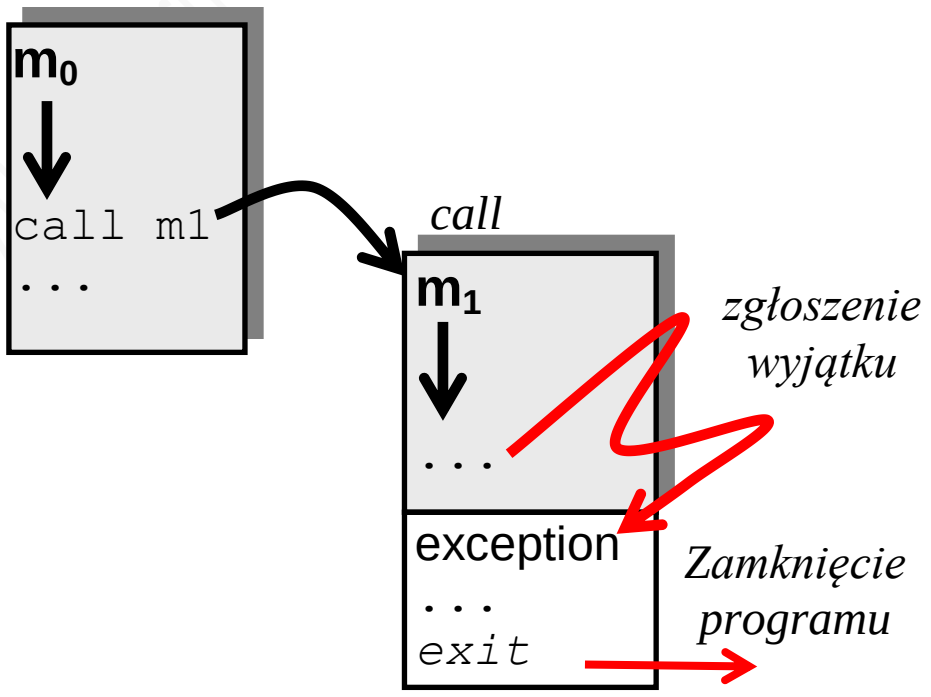
```
sqrt (n: REAL) return REAL is
 begin
   if x < 0.0 then
     raise Negative
   else
     ... - wyliczenie pierwiastka
   end
 exception
   when Negative =>
     put("Negative argument") -- komunikat
     return -- powrót bez zgłoszenia błędu
   when others => ...
 end -- sqrt
```



Jak nie należy obsługiwać wyjątków

Zamykanie działania całego programu przez zagnieżdżone moduły. **Moduł wielokrotnego użytku nie zna kontekstu swojego wywołania.**

```
sqrt (n: REAL) return REAL is
 begin
   if x < 0.0 then
     raise Negative
   end
 exception
   when Negative =>
     put("Negative argument") -- komunikat
     exit - zamknięcie działania programu
   when others => ...
 end -- sqrt
```



Wsparcie obsługi wyjątków przez język programowania

1. Rozdzielenie w strukturze programów przepływu sterowania dla normalnej pracy programu i dla obsługi wyjątków
2. Mechanizm zgłaszania wyjątków: jawnie przez programistów lub niejawnie przez środowisko sprzętowo-programowe. Obydwa rodzaje wyjątków powinny być obsługiwane w jednorodny sposób.
3. Mechanizm dla przekazywania sterowania do procedur obsługi błędów w przypadku zgłoszenia wyjątku. Powinny być zdefiniowane reguły przypisujące każdemu wyjątkowi określoną procedurę obsługi wyjątku.
4. Mechanizm umożliwiający zdefiniowanie zachowania programu po zakończeniu obsługi wyjątku na jeden z dwóch sposobów: podjęcie akcji dla zakończenia pracy systemu, lub ponowne wykonanie przerwanych operacji.
5. Powiązanie mechanizmu obsługi wyjątków z asercjami klas.

Większość języków programowania nie wspiera podstawowych schematów obsługi wyjątków.

Model obsługi wyjątków w C++

Dodatkowa składnia służąca do przekazywania sterowania i danych do wyodrębnionych procedur obsługi wyjątków.

1. **Definicja wyjątku**

   Wyjątek może być zdefiniowany jako:

   - typ prosty – ograniczona liczba wyjątków
   - typ wyliczeniowy – ograniczona taksonomia wyjątków
   - klasa

   Język C++ nie wspiera obsługi wyjątków systemowych

2. **Zgłoszenie wyjątku**

   ```
   Throw wyjątek – parametr aktualny
   ```

3. **Wyłapanie wyjątku**

   Kod, która ma wyłapywać zgłoszone wyjątki musi znajdować się w trybie nasłuchiwania:

   ```
   try { ... } // nasłuchiwanie
   catch(wyjątek) - parametr formalny
   ```

4. **Obsługa wyjątku**

   Procedura realizująca wybrana strategie obsługi wyjątków

Obsługa wyjątków

Przykład obsługi wyjątków:

```
class Tablica {
  int rozmiar;
  int *ip;
public:
class Zakres { };       // definicja klasa wyjątku
  Tablica (int roz)
  {ip = new int[rozmiar=roz];};
  int& operator[](unsigned);
  ~Tablica ( ) { delete ip; }
};
int& Tablica::operator[] (int indeks) {
  if (indeks >= 0 && indeks < rozmiar)
    return ip[indeks];
  else
    throw Zakres( );    // zgłoszenie wyjątku
}
```

W tym wypadku wyjątek jest pustą klasą bez zmiennych i metod. Informacja o typie błędu jest przekazywana przez typ obiektów.

Wywołanie konstruktora klasy Zakres

```
void main(void) {
  n=21;
  Tablica t1(12);
  try {                 // nasłuchiwanie wyjątków  zakresu
    for (int i=0; i<n; i++ )
    t1[i] = 2*i + 1;
  }
  catch (Tablica::Zakres) // wyłapywanie wyjątku
  {   // procedura obsługi wyjątku
    cout << "Przekroczenie zakresu" << endl;
    // i jeszcze coś sensownego !
   }
}
```

Wyłapanie wyjątku na podstawie jego typu

Wyjątki zwracające informacje

W tym wypadku wyjątki są wystąpieniami klas zawierającymi dodatkowe informacje

```
class Tablica {
  ...
class Zakres { // definicja wyjątku
public:
  int indeks;
  Zakres(int ind) { indeks = ind; } };
class Rozmiar { // definicja wyjątku
public:
  int rozmiar;
  Rozmiar(int rozm) {rozmiar = roz;} };
  ...
};
Tablica::Tablica (int roz) {
   if (roz >= 0 && roz <= MaxRozmiar)
     ip = new int[rozmiar=roz]
   else throw Rozmiar(roz); }

/* zwrócenie wartości przez z definicji beztypowy konstruktor */
try {
   ... // tworzenie tablic i operowanie na tablicach
}
catch (Tablica::Rozmiar r) {//parametr formalny
  cout << "Zły rozmiar tablicy:  "
   << r.rozmiar<< endl; }
}
catch (Tablica::Zakres z) {
  cout<<"Przekroczenie zakresu: "<<z.indeks
  << endl; }
}
```

Wyjątki, a klasy generyczne

Różne wyjątki dla różnych wystąpień klasy wzorca

```
template<class T> class Tablica {
    class Zakres { };
    ...
};

try {
    Tablica<int> ti(24);
    Tablica<float> tf(12);
    ...
}
catch(Tablica<int>::Zakres) {...}
catch(Tablica<float>::Zakres) {...}
```

Zdefiniowanie wyjątku jako klasy zagnieżdżonej wewnątrz klasy generycznej pozwala na rozróżnienie różnych wystąpień klasy generycznej.

Definicja wyjątku wspólnego dla wszystkich wystąpień klasy wzorca

```
class Zakres { };
template<class T> class Tablica {
    ...
};

try {
    Tablica<int> ti(24);
    Tablica<float> tf(12);
    ...
}
catch(Zakres) {...}
```

Klasyfikacja wyjątków

Typy wyliczeniowe:

```
enum BłądMatemat {Nadmiar, Niedomiar, ... };
try {
  ...
}
catch (BłądMatemat bm) {
  switch (bm) {
    case Nadmiar:
    ...
    case Niedomiar:
    ...
  }
}
```

Zdefiniowanie zbioru wyjątków jako hierarchii klas umożliwia ich elastyczną obsługę: unikalne metody obsługi poszczególnych wyjątków lub wspólną obsługę wielu typów wyjątków.

Hierarchia wyjątków:

```
class BłądMatemat { };
class Nadmiar: public BłądMatemat { };
class Niedomiar: public BłądMatemat { };
...
try {
  ...
}
catch (Nadmiar) {
  ... // obsługa błędu Nadmiar
}
catch (BłądMatemat bm) {//polimorficzne dopasowanie wyjątku
  ... //  obsługa błędów matematycznych, innych niż Nadmiar
}
```

Polimorficzna obsługa wyjątków z późnym wiązaniem

Hierarchia wyjątków:

```
class BłądMatemat {
    virtual informacja_o_błędzie() {...};
};
class Nadmiar: public BłądMatemat {
protected:
     const char* operacja;
     int operand1;
     int operand2;
public:
    virtual informacja_o_błędzie() {
    ...  // wykorzystuje zdefiniowane lokalnie zmienne
    };
};

try {
     ...
}
     catch (BłądMatemat bm) { // niejawne przycięcie!!!
     bm.informacja_o_błędzie();
        /* wiązanie statyczne - informacje specyficzne
          dla błędu nadmiaru są niedostępne  */
}

try {
     ...
}
     catch (BłądMatemat& bm) { // oryginał obiektu
     // wiązanie dynamiczne
     bm.informacja_o_błędzie();
}
```

Specyficzne konstrukcje obsługi wyjątków

Wyrażenie **throw** umieszczone w procedurze obsługi błędów i nie posiadające argumentu, oznacza ponowne zgłoszenie wyłapanego wyjątku.

```
try {
    ...
}
catch (Nadmiar) {
    ... // częściowa obsługa błędu Nadmiar
    throw; //ponowne zgłoszenie wyjątku Nadmiar
}
catch (BłądMatemat bm) {
    ... /* częściowa obsługa błędów
    matematycznych, innych niż Nadmiar */
    throw; //ponowne zgłoszenie wyjątku BłądMatemat
}
catch (...) { // "..." oznacza dowolny argument
    ... /* obsługa wszystkich pozostałych
    wyjątków */
}
```

Specyficznym typem formalnym wyrażenia wyłapującego wyjątki jest typ: "**...**". Oznacza on dowolny typ wyjątku. Umieszczenie wyrażenia **catch**(…) na końcu kodu obsługi błędów gwarantuje wyłapywanie wszystkich wyjątków.

Specyfikacja wyjątków w deklaracji klasy

Specyfikacja wyjątków zgłaszanych przez klasę jest częścią deklaracji klasy. Wiedza o wyjątkach zgłaszanych przez klasę jest niezbędna do poprawnego korzystania z wystąpień danej klasy.

1. Funkcja/metoda **`funkcja1`** może zgłosić jedynie wyjątki: **`Zakres`** i **`BłądMatemat`**:

```
void funkcja1 (const Tablica& )
  throw(Tablica::Zakres, BłądMatemat);
```

2. Funkcja/metoda **`funkcja2`** może zgłosić każdy wyjątek:

```
void funkcja2 (const Tablica& );
```

3. Funkcja/metoda **`funkcja3`** nie zgłasza żadnych wyjątków:

```
void funkcja3 (const Tablica& ) throw( );
```

Funkcje związane z obsługą wyjątków

1. Nieoczekiwane wyjątki

   Funkcja **`unexpected()`** jest wywoływana jeśli funkcja z wyspecyfikowaną listą wyjątków zgłasza wyjątek spoza tej listy.

   Funkcja `unexpected()` domyślnie wywołuje funkcję `terminate()`.

   Programista może samodzielnie zdefiniować znaczenie funkcji `unexpected()` za pomocą funkcji `set_unexpected()`

2. Nie wyłapane wyjątki

   Funkcja **`terminate()`** jest wywoływana kiedy:

   - mechanizm obsługi wyjątków nie może znaleźć procedury obsługi dla zgłoszonego wyjątku;
   - przed przejściem do obsługi wyjątku nastąpi uszkodzenie stosu;
   - kiedy destruktor wywołany przed przejściem do obsługi wyjątku próbuje zgłosić wyjątek.

   Funkcja `terminate()` domyślnie wywołuje funkcję `abort()`.

   Programista może samodzielnie zdefiniować znaczenie funkcji `terminate()` za pomocą funkcji `set_terminate()`.

Wzorce obsługi wyjątków

Zorganizowana panika:

```
class klasa {
    ...
    try {
        ...
        throw Wyjątek_1( );
    }
    catch (Wyjątek_1) {
    ...
    // kompensacja wykonanych działań
    throw Wyjątek_2( );
    // zgłoś wyjątek do miejsca wywołania
};
```

Odporność na błędy:

```
class klasa {
    ...
    jeszcze_raz:
    try {
        ...
        throw Wyjątek( );
    }
    catch (Wyjątek_1) {
    ...
    // kompensacja wykonanych działań
    // zmiana warunków wywołania
    goto jeszcze_raz;
};
```

Obsługa wyjątków w języku Java

W języku Java zintegrowano wyjątki systemowe i wyjątki definiowane przez użytkownika. Wszystkie wyjątki użytkowników muszą być bezpośrednimi lub pośrednimi wystąpieniami systemowej klasy **`Exception`**.

Hierarchia klas wyjątków i błędów: