Wyjątki

Wyjątkiem nazywane zdarzenie pojawiające się podczas wykonywania programu, które przerywa wykonanie normalnego ciągu instrukcji.

Wyjątkiem nazywany jest także obiekt tworzony podczas obsługi zdarzeń wyjątkowych. Obiekt ten należy do klasy `Throwable` lub jej klas potomnych (`Exception` i `RuntimeException`).

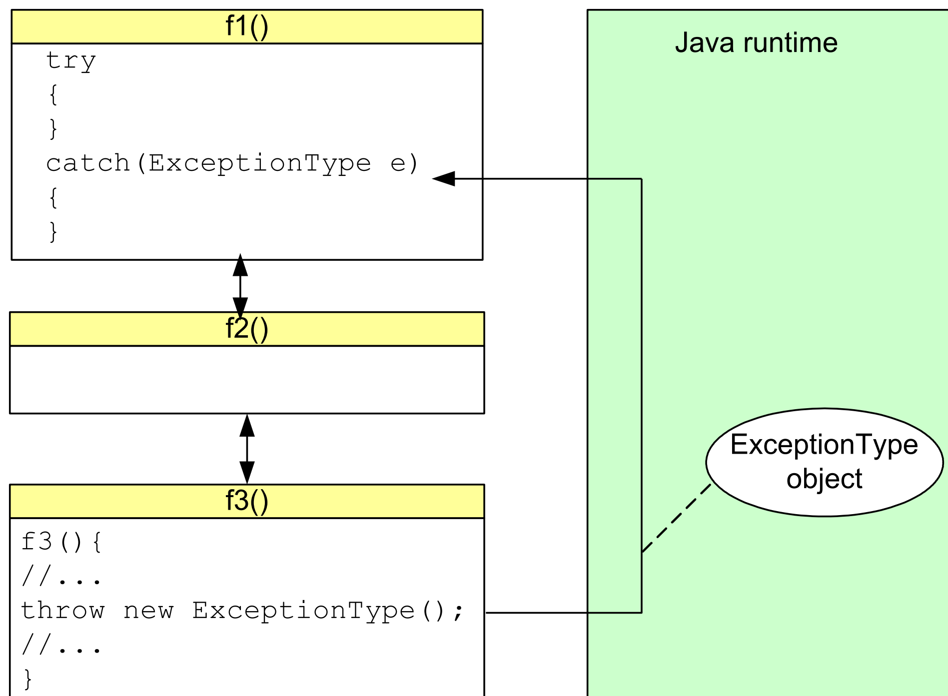
Wyjątki tworzone są jako reakcja na błędy – takie jak: dzielenie przez zero, dereferencja obiektu wskazywanego przez referencję o wartości `null`, próba dostępu do nieistniejącego elementu tablicy, błąd otwarcia pliku.

Wyrzucanie wyjątków. Jeżeli podczas wykonania metody zaistnieje błąd uniemożliwiający jej kontynuację, wówczas metoda jawnie lub niejawnie tworzy obiekt zawierający informacje o typie błędu i miejscu jego wystąpienia. Obiekt ten jest przekazywany do maszyny wirtualnej Java (czy szerzej środowiska, w którym program jest wykonywany – ang. *runtime system*). Operacja ta nazywana jest wyrzucaniem wyjątków.

- Wyjątki mogą być wyrzucane jawnie za pomocą instrukcji `throw`. Zazwyczaj generowane obiekty należą do klas wyjątków stworzonych przez programistę. Argumentem instrukcji jest referencja do obiektu, stąd typowe wywołanie ma postać: `throw new ExceptionType();` . Na stercie tworzony jest obiekt klasy sygnalizującej wyjątek, wołany jest jej konstruktor i opcjonalnie przekazywane argumenty konstruktora.
- Wyjątki mogą zostać wygenerowane niejawnie w wyniku działania mechanizmów zabezpieczających wbudowanych w maszynę wirtualną, które mają na celu wykrycie takich typowych błędów wykonania, jak odwołania do nieistniejącego obiektu za pomocą referencji o wartości `null` lub próba przekroczenia rozmiarów tablicy. Tego rodzaju wyjątki należą zazwyczaj do klas pochodnych `RuntimeException` (błędów wykonania).

Przechwytywanie wyjątków. Po pojawieniu się wyjątku przerywane jest normalne wykonanie metody, w której wyjątek został wyrzucony. System rozpoczyna poszukiwanie odpowiedniego fragmentu kodu, który jest odpowiedzialny za obsługę wyjątku (ang. *exception handler*).

Kod obsługi wyjątku ma postać: `catch(ExceptionType e){…}`



Poszukiwanie rozpoczyna się od metody, w której wyjątek został wyrzucony. Jeżeli w bieżącej metodzie kod obsługujący wyjątek nie zostanie odnaleziony, wówczas przeglądana jest w górę zawartość stosu wywołania metod, aż do znalezienia odpowiedniego handlera wyjątku. Handler może zostać dopasowany do wyjątku, jeżeli typem wyrzuconego wyjątku jest wyspecyfikowana klasa `ExceptionType` lub klasa potomna.

- Jeżeli handler zostanie odnaleziony, wówczas następuje przekazanie do niego sterowania z pominięciem dalszych instrukcji, które znajdują się w metodzie, gdzie nastąpiło wyrzucenie wyjątku. Mechanizm ten nazywany jest *przechwytywaniem wyjątków*.
- Jeżeli na stosie wywołań funkcji brak jest odpowiedniego kodu obsługi wyjątków, maszyna wirtualna kończy działanie programu. *Ściślej wątku; programu jeżeli jest programem jednowątkowym.*

Zalety wyjątków

Oddzielenie kodu obsługującego błędy od kodu realizującego podstawowy scenariusz.

```
try{   // podstawowy scenariusz
// (1)
// (2) …
}
catch(ExceptionType1 e){
}
catch(ExceptionType2 e){
}
```

Propagacja informacji o błędach w górę stosu wywołań funkcji.

W tradycyjnym modelu obsługi błędów informacja o ich pojawieniu się musi być przekazywana w górę stosu wywołanych funkcji za pośrednictwem zwracanych wartości.

Mechanizm wyjątków pozwala na automatyczną propagację błędów niezależnie od typów wartości zwracanych przez funkcje.

Język Java narzuca jednak pewne ograniczenia: specyfikacja wyrzucanych wyjątków jest częścią interfejsu funkcji. Jeżeli w wyniku działania funkcji może zostać wyrzucony wyjątek (bezpośrednio w funkcji lub w wyniku wywołania innej funkcji) musi zostać to zadeklarowane za pośrednictwem deklaracji `throws`.

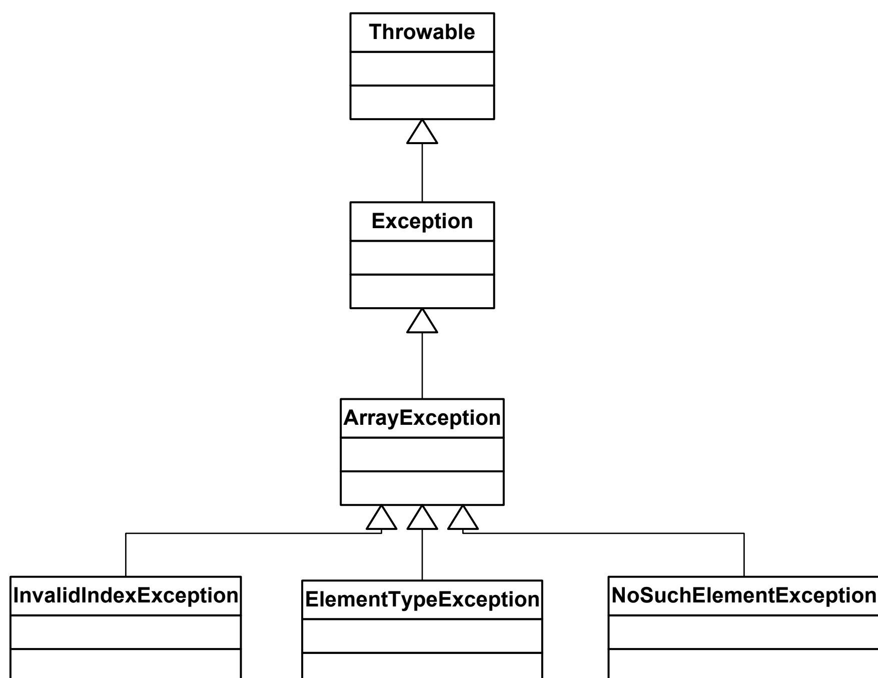
Przykład

```
void f1()
{
try{
   f2();
}
catch (ExceptionType e){/* … */}
}

void f2() throws ExceptionType {
   f3();
}

void f3() throws ExceptionType {
   throw new ExceptionType();
}
```

Grupowanie informacji o błędach i rozróżnianie błędów.

Obiekty, które mogą być argumentem instrukcji `throw` muszą być podklasami klasy `Throwable`.

Zazwyczaj klasy wyjątków zdefiniowane przez użytkownika dziedziczą po klasie `Exception`.



Jeżeli dysponujemy hierarchią klas opisujących możliwe błędy, wówczas możemy napisać

- handlery przechwytujące wyłącznie wyjątki określonego typu, np.:

```
catch (InvalidIndexException e){
}
```

- handlery przechwytujące grupę wyjątków

```
catch (ArrayException e){
}
```

- handlery przechwytujące wszystkie wyjątki

```
catch (Exception e){
}
```

Mogą być one używane jednocześnie, jednakże muszą wówczas wystąpić w porządku odwrotnym do miejsca w hierarchii. (Wpierw przechwytujące mniej, potem bardziej ogólne typy wyjątków.)

Przykład

```
try{
}
catch (Exception e){}
catch (ArrayException e){}
```

Kompilator odmówi skompilowania powyższego kodu – handler `ArrayException` nigdy nie zostanie wywołany, ponieważ wszystkie wyjątki przechwyci występujący wcześniej handler klasy bardziej ogólnej: `Exception`.

Wsparcie dla procesu usuwania błędów

Generowane wyjątki są obiektami. Ich stan pozwala przekazać szczegółowe informacje o typie i miejscu wystąpienia błędów.

W szczególności obiekt może przechowywać komunikat (`String`) oraz zawartość stosu wywołań funkcji dostępną zazwyczaj dzięki metodzie `printStackTrace()`.

Wyjątki użytkownika

Programista projektujący bibliotekę może utworzyć własne klasy służące do sygnalizacji wyjątków.

Przykład

```
class MyException extends Exception {}

class BadArgException extends Exception
{
 BadArgException(String s){super(s);}
 BadArgException(){}
}
```

W klasie `BadArgException` konstruktor z parametrem pozwala na ustawienie pola `message` klasy `Throwable`. Konstruktor bezargumentowy inicjuje pole `message` pustym tekstem.

Przykład

```
void f()
  {
    class MyException extends Exception{}
    try {
      throw new MyException();
    }
    catch(MyException e){
      e.printStackTrace();
    }
}
```

Wyjątki są generowane i przechwytywane wewnątrz funkcji. Klasa `MyException` jest użyta tylko raz – w lokalnym kontekście.

W tego typu przypadku lepiej inaczej sterować wykonaniem instrukcji.

Przykład – obliczanie silni

```
public class Silnia
{
  public static int silnia(int n)
    throws BadArgException
  {
    if(n<0)
      throw new BadArgException("Negative:"+n);
    if(n==0)return 1;
    int sn_1=silnia(n-1);
    if(sn_1>(double)(Integer.MAX_VALUE)/n)
      throw new BadArgException("Too big:"+n);
    return n*sn_1;
  }

  public static void main(String args[])
  {
    try{
      int x = Integer.parseInt(args[0]);
      System.out.println(x+"!="+silnia(x));
    }
    catch(BadArgException e){
      e.printStackTrace();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Idea działania jest następująca. Użytkownik uruchamiając program podaje argument liczbowy, dla którego będzie następnie obliczana rekurencyjnie wartość silni.

Wywołanie> `java Silnia liczba`

Funkcja silnia generuje wyjątek dla ujemnych argumentów, a także w przypadku, kiedy wartość jest zbyt duża i wynik nie mieści się w zakresie `Integer.MAX_VALUE`.

Wywołanie `java Silnia -1` da rezultat

```
BadArgException: Negative:-1
   at Silnia.silnia(Silnia.java:13)
   at Silnia.main(Silnia.java:25)
```

Wywołanie `java Silnia 15` da rezultat

```
BadArgException: Too big:13
   at Silnia.silnia(Silnia.java:17)
   at Silnia.silnia(Silnia.java:15)
   at Silnia.silnia(Silnia.java:15)
   at Silnia.main(Silnia.java:25)
```

Podczas wykonania programu mogą powstać inne wyjątki i zostać przechwycone przez handler `catch(Exception e){}` w funkcji `main`.

W wyniku wywołania `java Silnia` raportowany będzie wyjątek:

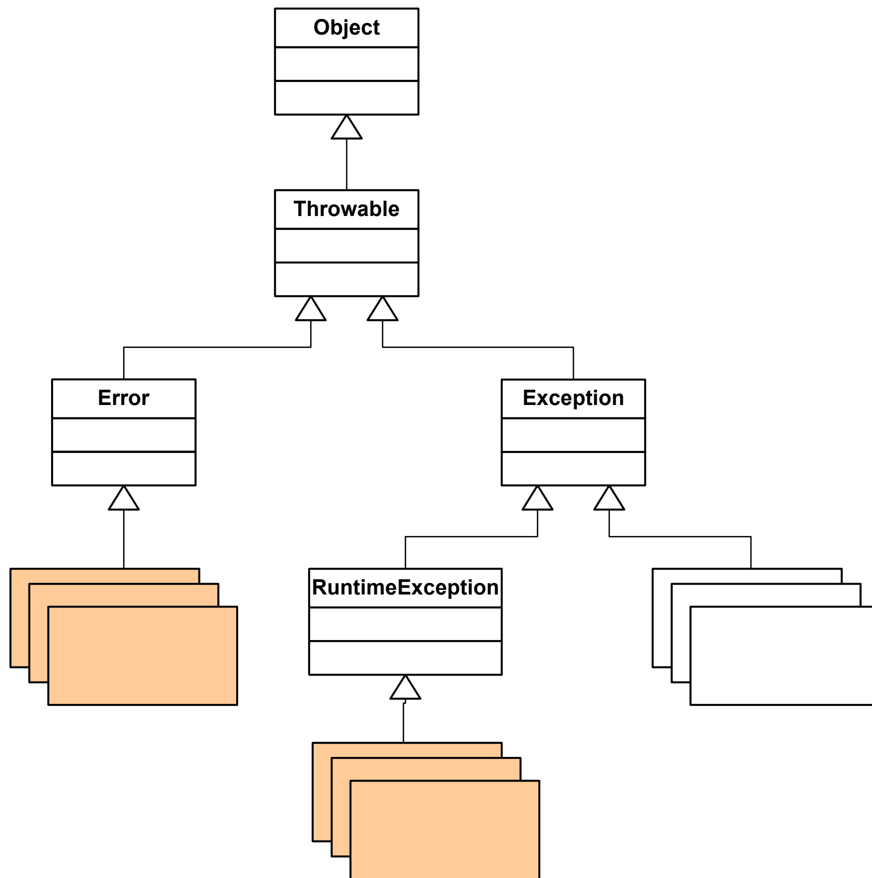
```
java.lang.ArrayIndexOutOfBoundsException: 0
    at Silnia.main(Silnia.java:24)
```

Błędny format liczby, np.: wywołanie "`java Silnia a`", wygeneruje wyjątek `java.lang.NumberFormatException` w funkcji `parseInt()`.

Hierarchia wyjątków

Klasa `Throwable` jest nadklasą wszystkich klas reprezentujących błędy i wyjątki. Argumentem instrukcji `throw` musi być referencja typu `Throwable` lub jej podklas.

Klasa `Throwable` ma dwie klasy pochodne `Error` i `Exception`.



- `Error` – wyrzucenie wyjątku tego typu sygnalizuje błąd, który nie powinien się nigdy pojawić (ang. *abnormal condition*) mający swoją przyczynę w konfiguracji maszyny wirtualnej, np.: fałszywa asercja w kodzie, błąd biblioteki graficznej, błąd linkowania, brak zasobów uniemożliwiający pracę maszyny wirtualnej.

  Wyjątków tego typu nigdy nie przechwytuje się, ponieważ aplikacja nie może skorygować błędów w środowisku wykonania powstałych nie z jej winy. Błędów nie deklaruje się też w sekcji `throws` metody.

- `Exception` – wyjątki tego typu mogą być wyrzucane i przechwytywane w programach.

  Zazwyczaj wszystkie wyjątki zadeklarowane przez programistę są podklasami klasy `Exception`.

- `RuntimeException` – wyjątki tego typu są generowane przez maszynę wirtualną podczas działania programu. Typowe przykłady to:
  - `NullPointerException` – generowane przy próbie dostępu do obiektu za pomocą referencji `null`
  - `ArithmeticException` – np.: dzielenie przez 0
  - `ArrayStoreException` – dodawanie do tablicy obiektów nieodpowiedniego typu
  - `IndexOutOfBoundsException` – dostęp do nieistniejącego elementu tablicy
  - `NoSuchElementException` – wysyłane przez `Enumeration.nextElement()` przy przekroczeniu zakresu kolekcji
  - `ClassCastException` – generowane przy próbie rzutowania do niewłaściwego typu
  - i wiele innych...

Tego typu wyjątki mogą pojawić się podczas wykonania programu praktycznie wszędzie. Niemal każda metoda pośrednio lub bezpośrednio może wygenerować wyjątki typu `RuntimeException`.

Koszt specyfikacji, że metoda może generować wyjątków tego typu oraz ich przechwytywania byłby bardzo duży i owocowałby wieloma wierszami nieczytelnego i trudnego w zarządzaniu kodu.

Z tego powodu projektanci języka zadecydowali, że pochodne klasy `RuntimeException` nie muszą być specyfikowane na liście wyjątków wyrzucanych przez metody, a także nie muszą być przechwytywane.

Weryfikowane wyjątki

Programiści mogą posługiwać się dwoma typami wyjątków:

- Jeżeli wyjątek jest podklasą `Exception` ale nie `RuntimeException`, wówczas jest to wyjątek weryfikowany (ang. *checked exception*).
- Jeżeli wyjątek jest podklasą `RuntimeException`, wówczas jest on nieweryfikowany (ang. *unchecked exception*).

W przypadku wyjątków weryfikowanych narzucane są bardzo silne ograniczenia: kompilator weryfikuje przepływy wyjątków i lokalizację ich handlerów. Jeżeli metoda woła inną metodę wyrzucającą weryfikowany wyjątek, wówczas musi go przechwycić lub zadeklarować w sekcji `throws`, że może go wyrzucić.

<u>Przykład</u>

```
class CheckedException extends Exception{}
class UnCheckedException extends RuntimeException{}

public class ExceptionTest {
  void f() throws CheckedException  {
    double d = new java.util.Random().nextDouble();
    if(d<0.5)throw new CheckedException();
    else throw new UnCheckedException();
  }
  void g() throws CheckedException  {
    f(); // f generuje CheckedException
  }
  void h() {
    try{
      f();
    }
    catch(CheckedException e){  e.printStackTrace();}
    // niewymagane
    //catch(UnCheckedException e){e.printStackTrace();}
   }
  public static void main(String args[]) {
    new ExceptionTest().h();
  }
}
```

Wyjątki i dziedziczenie

Specyfikacja weryfikowanych wyjątków wyrzucanych przez metodę jest częścią jej publicznego interfejsu – podobnie jak typ zwracanej wartości.

Jeżeli klasa przedefiniowuje polimorficzną metodę w klasie bazowej, wówczas ten publiczny interfejs musi zostać zachowany – lista wyrzucanych wyjątków w nie może zostać rozszerzona w metodzie klasy potomnej.

Przykład

```
class Ex1 extends Exception{}
class Ex2 extends Exception{}


class A {
 void f() throws Ex1 , Ex2 {
 }
}


class A1 extends A {
 void f() throws Ex1 {
   throw new Ex1();
 }
}

class A2 extends A {
 void f() throws Ex2 {
   throw new Ex2();
 }
}
```

```
public static void
main(String args[])
{
  A1 a1= new A1();
  A2 a2= new A2();

  try{
    a1.f();
  }
  catch(Ex1 e) { }

  try{
    a2.f();
  }
  catch(Ex2 e){ }

  try{
    A a = a1;
    a.f();
  }
  catch(Ex1 e){ }
  catch(Ex2 e){ }
}
```

Z powyższego przykładu widać, że:

- metoda może deklarować wyjątki, ale nie musi ich wyrzucać
- specyfikacja listy wyjątków polimorficznej metody w klasie bazowej musi być najszersza – wywołując metodę za pośrednictwem referencji typu bazowego musimy być przygotowani na wszystkie możliwe wyjątki, jakie mogą zostać wygenerowane w metodach klas potomnych.

Programista decydując, że polimorficzna metoda w nowej klasie należącej do pewnej hierarchii klas powinna wyrzucać nowy typ weryfikowanego wyjątku:

- umieszcza go na liście `throws` metody implementowanej klasy
- dodaje go do listy `throws` w klasie bazowej, nie modyfikując specyfikacji funkcji w pozostałych klasach.

Natomiast jest możliwe grupowanie.

```
class ExBase extends Exception{}
class Ex1 extends ExBase{}
class Ex2 extends ExBase{}
class Ex3 extends ExBase{}

class A{
    void f()throws ExBase{}
}

class B extends A{
    @Override
    void f() throws Ex1, Ex2, Ex3{}
}
```

Funkcja `f()` w klasie bazowej A deklaruje, że wyrzuca wyjątek typu `ExBase`. Tym samym handler przechwyci **`Ex1, Ex2, Ex3.`**

```
void callF(A a){
    try{
        f();
    }
    catch(ExBase e){
        // przechwyci kazdy wyjątek potomny
    }
}
```

Można więc bezpiecznie wywołać dla klasy potomnej `A`, np: `callF(new B()).`

W oficjalnej dokumentacji pojawiło się kilka reguł dotyczących użycia nieweryfikowanych wyjątków:

- W metodach można wykrywać i wyrzucać wyjątki typu `RuntimeException` po napotkaniu błędów w trakcie wykonania programu. Jest jednak prościej pozostawić wykrywanie i generowanie tego typu wyjątków maszynie wirtualnej. Metody powinny raczej wyrzucać wyjątki typu `Exception`, a nie `RuntimeException`.
- Generacja wyjątków typu `RuntimeException` powinna wiązać się z błędami i problemami wykrytymi podczas działania maszyny wirtualnej.
- Programista nie powinien tworzyć klas potomnych `RuntimeException`, ani wyrzucać wyjątków tego typu tylko po to, aby uniknąć konieczności ich specyfikacji podczas definiowania metod.

Mechanizm weryfikowanych wyjątków budzi wiele kontrowersji. Krytycy zauważają, że jest on efektywny w przypadku małych projektów, zwłaszcza dla kodu o charakterze demonstracyjnym.

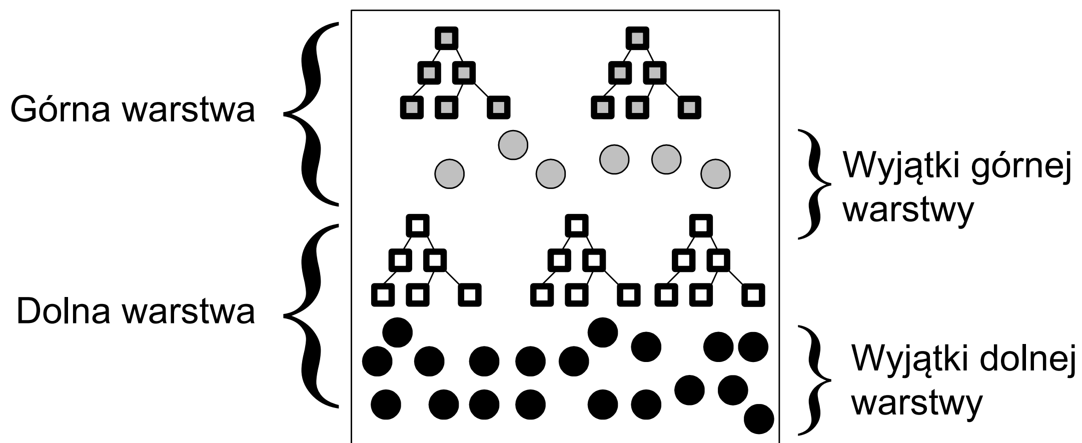
W większych programach konieczność propagacji informacji o wyrzucanych weryfikowanych wyjątkach jest dużym narzutem.

Dlatego programiści uruchamiając kod często wbrew zaleceniom używają klas wyjątków pochodnych po `RuntimeException` lub tworzą handlery wyjątków, w których „nic się nie dzieje”, nie są podejmowane żadne akcje zaradcze.

Daje to złudne poczucie bezpieczeństwa: aplikacja reaguje na wszystkie wyjątki, ale ich tak naprawdę nie obsługuje.

Wyjątki w architekturach wielowarstwowych

Duże programy są konstruowane w postaci architektury wielowarstwowej.



Na granicy warstw zazwyczaj zachodzi filtrowanie wyjątków. Na ogół warstwy te są rozwijane w miarę niezależnie, z własnymi zestawami wyjątków i nie jest wskazane, aby wyjątki dolnej warstwy znalazły się w specyfikacjach wyrzucanych wyjątków metod warstwy górnej.

Programista implementując metody na granicy warstw może:

- napisać handler dla wyjątku ze standardową akcją informacyjną typu wydruk na konsoli
- przechwycić wyjątek i zamienić go na rezultat funkcji (np.: `boolean`)
- przekonwertować wyjątek dolnej warstwy na wyjątek górnej warstwy (przeprowadzając np.: grupowanie)

Składowe klasy Throwable

Klasa stanowi klasę bazową dla wszystkich klas wyjątków, stąd dziedziczą one jej składowe.

Komunikat

Tworząc obiekt klasy `Throwable` możemy w konstruktorze przekazać dodatkowy opis tekstowy (komunikat). Metoda `String getMessage()` pozwala odczytać komunikat niesiony przez przechwycony wyjątek.

Informacje o stosie

Obiekt klasy `Throwable` przechowuje informację o stosie wywołań funkcji (ang. `stack trace`) wątku, w którym został wygenerowany.

- Stos ten może zostać wydrukowany za pomocą metody `printStackTrace()`
- Metoda `getStackTrace()` zwraca informację o stosie w postaci tablicy elementów opisujących poszczególne funkcje (wsparcie dla debuggowania)
- Metoda `fillInStackTrace()` tworzy kopię obiektu typu `Throwable,` ale zawierającą bieżącą informację stosie

Powtórne wyrzucanie wyjątków (ang. *rethrowing*). W handlerze wyjątku można powtórnie wyrzucić wyjątek, jeżeli stwierdzimy, że nie jesteśmy w stanie go obsłużyć. Metoda `fillInStackTrace()` pozwala zmodyfikować informację o miejscu generacji wyjątku.

Przykład

```
void foo() throws Excpt{
   try{
      f(); //zakladamy void f() throws Excpt{…}
   }
   catch (Excpt e){
      System.out.println("Can't handle it");
      //throw e;
      throw (Excpt) e.fillInStackTrace();
   }
}
```

Łańcuchy wyjątków

Klasa `Throwable` zawiera składową `cause` (przyczyna) typu `Throwable`. Służy ono do przechowywania informacji o wyjątku, który spowodował wygenerowanie danego wyjątku.

- Referencja może być ustawiona w konstruktorze klasy, ale także za pośrednictwem metody:
  `Throwable initCause(Throwable cause).`
- Metoda `Throwable getCause()` zwraca zapisaną wartość.

Mechanizm tworzenia łańcuchów wyjątków polega na ich przechwytywaniu i powtórnym wyrzucaniu.

Wyjątki mogą zostać wygenerowane w wyniku pojawienia się innego wyjątku jako rezultat powtórnego wyrzucania.

Na granicy dwóch warstw architektury aplikacji, np.: operacji na plikach i interfejsu graficznego bardzo często zachodzi konieczność filtrowania i konwersji wyjątków. Wyjątek w niższej warstwie jest przechwytywany i w jego miejsce wyrzucany wyjątek warstwy wyższej.

W starszej wersji biblioteki (przed 1.4) informacja o wyjątku warstwy niższej będącym przyczyną wystąpienia wyjątku w warstwie wyższej nie mogła zostać przekazana.

Przykład

```
class LowLevelException extends Exception{}

class LowLevel
{
 void f()throws LowLevelException{
   throw new LowLevelException();
 }
}
```

```
class HighLevelException extends Exception{}

class HighLevel
{
 void g()throws HighLevelException{
   try{
     new LowLevel().f();
   }
   catch(LowLevelException e){
     // throw new HighLevelException();
     throw (HighLevelException)
     new HighLevelException().initCause(e);
  }
 }
}
```

Wywołanie

```
public static void main(String args[])
{
  try{
    new HighLevel().g();
  }
  catch(HighLevelException e){
      e.printStackTrace();
  }
}
```

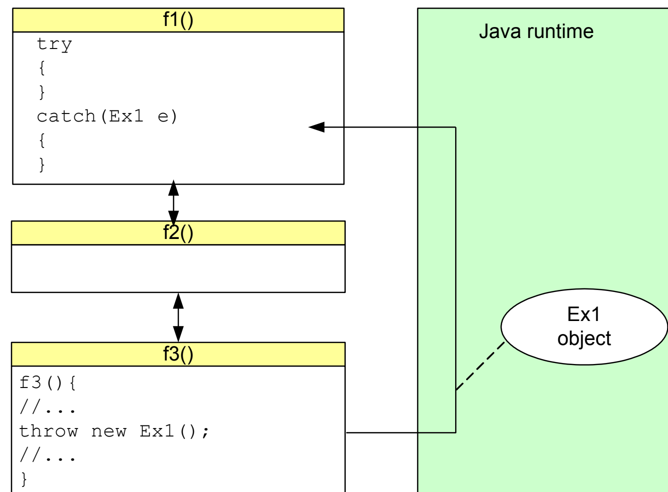
Wydrukuje pełną informację o wyjątku `HighLevelException` i jego przyczynie: `LowLevelException`

```
HighLevelException
    at HighLevel.g(ExceptionTest.java:43)
    at ExceptionTest.main(ExceptionTest.java:122)
Caused by: LowLevelException
    at LowLevel.f(ExceptionTest.java:31)
    at HighLevel.g(ExceptionTest.java:40)
    ... 1 more
```

Jeżeli aplikacja zawiera kilka warstw, wówczas tak generowane i konwertowane podczas powtórnego wyrzucania wyjątki tworzą łańcuch.

Blok finally

W języku C++ podczas obsługi wyjątków następuje oczyszczanie stosu. Wołane są destruktory wszystkich obiektów automatycznych, dla których pamięć przydzielana jest na stosie.



W języku Java brak jest destruktorów, pamięć dla obiektów przydzielana jest na stercie i jej zwalnianie odbywa się pod kontrolą mechanizmu *garbage collection*.

Projektując język przewidziano specjalne miejsce, w którym należy umieścić kod przywracający pożądany stan otoczenia programu, np.: zwalniający zasoby systemowe przydzielone w trakcie wykonania metody, usuwający obiekty interfejsu użytkownika, itd. Jest nim blok `finally{...}` , który może pojawić się na końcu metody, po bloku `try` i blokach `catch`.

Instrukcje w bloku `finally` są wykonywane **zawsze** podczas oczyszczania stosu przy przeskoku do handlera wyjątku, a także podczas normalnego wykonania metody.

```
try{
  // blok, w którym mogą zostać wygenerowane wyjątki
}
catch(A a){ // handler wyjątku A
}
catch(B b){ // handler wyjątku B
}
finally{
  // blok wykonywany zawsze
}
```

Przykład

```
class ExceptionTest{
void f3() throws Ex1
{
  try{
    throw new Ex1();
  }
  finally{
    System.out.println(
    "finally in f3");
  }
}
void f2() throws Ex1
{
  try{
    f3();
  }
  finally{
  System.out.println(
    "finally in f2");
  }
}
```

```
void f1()
{
  try{
    f2();
  }
  catch(Exception e){
    e.printStackTrace();
  }
  finally{
    System.out.println(
    "finally in f1");
  }
}
public static void
  main(String args[]){
    new ExceptionTest().f1();
}
}
```

Wykonanie programu spowoduje wypisanie:

```
finally in f3
finally in f2
Ex1
    at ExceptionTest.f3(ExceptionTest.java:94)
    at ExceptionTest.f2(ExceptionTest.java:103)
    at ExceptionTest.f1(ExceptionTest.java:112)
    at ExceptionTest.main(ExceptionTest.java:162)
finally in f1
```

Może się wydawać, że w metodach działających na plikach blok `finally` jest idealnym miejscem dla zamknięcia pliku.

Pliki powinny w języku Java być zamykane jawnie. Wywołanie metody `close()` zamykającej plik znajduje się w finalizatorze klasy, ale brak jest gwarancji określających kiedy i czy w ogóle dojdzie do finalizacji.

Niestety wadą jest to, że metoda `close()` wyrzuca weryfikowany wyjątek typu `IOException`, który musi być przechwycony.

Przykład

```
public void read()
{
  FileReader in=null;
  try{
    in = new FileReader("a.txt");
      for(;;){
        int c = in.read();
        if(c<0)break;
        System.out.print((char)c);
      }
  }
  catch(IOException e){
    e.printStackTrace();
  }
  finally{
    try{
      if(in!=null)in.close();
    }
    catch(IOException e){
      e.printStackTrace();
    }
  }
}
```

Kolejną wadą mechanizmu jest to, że wyjątki generowane w sekcji `finally` mogą zastąpić aktualnie obsługiwany wyjątek, podobnie jak dzieje się to przy powtórnym wyrzucaniu wyjątków.

Przykład

```
class TestFinally
{
  void f() throws Ex1 {
    throw new Ex1();
  }
  void g() throws Ex1, Ex2 {
    try{
      f();
     }
     finally{
      System.out.println("finally g");
      throw new Ex2();
      }
  }
}
```

```
public static void main(String args[])
{
  try{
    new TestFinally().g();
  }
  catch(Ex1 e){
    System.out.println("Ex1 caught");
  }
  catch(Ex2 e){
    System.out.println("Ex2 caught");
  }
}
```

Rezultatem wykonania powyższego kodu jest: `finally g Ex2 caught`.

Gdyby jednak wyjątek generowany w sekcji `finally` funkcji `g()` był przechwytywany, wówczas uruchomiony zostanie handler `Ex1` w funkcji `main()`.

```
  finally{
    try{
      System.out.println("finally g");
      throw new Ex2();
    }
    catch(Ex2 e){}
  }
```

Wyjątki w konstruktorach

Zadaniem konstruktora jest wprowadzenie obiektu w poprawny stan początkowy. Kiedy czas życia obiektu dobiega końca, wówczas – jeżeli jest to konieczne – użytkownik może wywołać specjalną metodę, która zwalnia zasoby nie znajdujące się pod kontrolą mechanizmu *garbage collection* (np.: zamyka otwarte pliki).

Podczas wykonania konstruktorów mogą również zostać wygenerowane wyjątki wewnątrz wołanych funkcji. Zazwyczaj są one dalej wyrzucane, wówczas operator `new` tworzący obiekt danej klasy zwraca referencję `null`. Nie można się nią posłużyć dla ewentualnego zwolnienia zasobów, a równocześnie częściowo skonstruowany obiekt może pozostawać w niespójnym stanie, np.: pozostawić otwarty plik.

Zwalniania zasobów nie należy w takim przypadku umieszczać w bloku `finally`, ponieważ byłby on wykonany zawsze. Można oczywiście posłużyć się dodatkowymi flagami ustawianymi po przechwyceniu wyjątków i powtórnie je wyrzucać, lub też kod sprowadzający do poprawnego stanu umieścić w handlerach.

<u>Przykład (Thinking In Java)</u>

```
class InputFile {
  private BufferedReader in;
  public InputFile(String fname) throws Exception {
    try {
      in = new BufferedReader(new FileReader(fname));
    }
    catch(FileNotFoundException e) {
      throw e;      // powtórnie wyrzuć wyjątek
    }
    catch(Exception e) {
        myClose(); // zamkninij plik
        throw e;   // powtórnie wyrzuć wyjątek
      }
    finally {  }
   }
   void myClose(){
     try { in.close(); }
     catch(IOException e2) {}
   }
}
```

Try-with-resource

W wersji 1.7 Javy wprowadzono nowy mechanizm: połączenie bloku try z deklaracją zasobu (ang. *try with resource*).

Jako zasób może być użyta dowolna klasa implementująca interfejs `AutoCloseable`

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

lub interfejs potomny (wprowadzony we wcześniejszej wersji JDK)

```
package java.io;
import java.io.IOException;
public interface  Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

Zasób ten zostanie automatycznie zamknięty (wywołana zostanie metoda `close()`, przy opuszczaniu bloku `try`).

Przykład

```
static void displayAsCSV(String path){
    try(BufferedReader rd = new BufferedReader(
        new FileReader(path))
    ){
        for(;;){
            String line = rd.readLine();
            if(line==null)break;
            String[] elements = line.split("\\s+");

            for(int i =0;i<elements.length;i++){
                if(i!=0)System.out.print(";");
                System.out.print(elements[i]);
            }
            System.out.println();
        }
        // rd.close() Nie musimy zamykać pliku
    }
    catch (FileNotFoundException ex) {
    }
    catch (IOException ex) {
    }
}

public static void main(String[]args){
    displayAsCSV("data.txt");
}
```

Plik wejściowy:

```
X Y F
1.12 2.34 1.11
2.13 4.15 0.45
```

Plik wyjściowy:

```
X;Y;F
1.12;2.34;1.11
2.13;4.15;0.45
```

Przykład: kopiowanie pliku tekstowego połączone ze zmianą kodowania. Dwa obiekty typu `AutoCloseable` zostały zadeklarowane jako zasoby w bloku `try`.

```
void copy(String src,String trg){
  try(
    InputStreamReader in =new InputStreamReader(
                     new FileInputStream(src), "Cp1250");
    OutputStreamWriter out = new OutputStreamWriter(
                     new FileOutputStream(trg), "UTF-8")
     ){
    for(;;){
      int c = in.read();
      if(c<0)break;
      out.append((char)c);
    }
  }
  catch (FileNotFoundException ex) {
  }
  catch (IOException ex) {
  }
}
```

Inny przykład (Java tutorial)

```
public static void viewTable(Connection con) throws
SQLException {

  String query = "select COF_NAME, SUP_ID, PRICE, SALES,
                  TOTAL from COFFEES";

  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);

    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");

        System.out.println(coffeeName + ", " + supplierID +
       ", " + price + ", " + sales + ", " + total);
    }
  }
  catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  }
}}
```

Podsumowanie

W jakim celu można stosować wyjątki:

- Aby uprościć obsługę błędów w programie
- Zwiększyć bezpieczeństwo biblioteki i aplikacji – zarówno podczas uruchamiania jak i późniejszej eksploatacji
- Aby obsłużyć błąd na odpowiednim poziomie wywołań funkcji. Należy unikać przechwytywania wyjątków w miejscu, gdzie dalej nie wiadomo, jak poradzić sobie z zaistniałym problemem.
- Skorygować dane wejściowe i powtórnie wywołać metodę, w której pojawił się wyjątek.
- Przesłać informację o błędzie do górnej warstwy aplikacji
- Przeprowadzić możliwe operacje (np.: przywracanie poprawnego stanu) na danym poziomie i przesłać wyjątek w górę.
- Dokonać konwersji wyjątków na granicy warstw aplikacji