JTP: Wyjątki

dr hab. Piotr Kosiuczenko
profesor WAT

1

Błędy i Wyjątki

- Programy zawierają błędy – im więcej linii kodu, tym prawdopodobieństwo wystąpienia błędu w kodzie jest większe.
- Typizacja pozwala na wykrycie części z nich, podobnie testowanie, ale nie wszystkich.
- Część błędów w kodzie zostaje ujawniona dopiero w czasie wykonania programu i wtedy trzeba coś z nim zrobić.
- Z drugiej strony, użytkownik może wywołać metodę w niewłaściwy sposób i też wymaga zaradzenia.
- W jawie rozróżnia się błędy (ang. `Error`) i wyjątki (ang. `Exception`).
- Oba są podklasami `Throwable`.

2

`Throwable`

- Gdy coś jest nie tak, np. program próbuje przeczytać wartość atrybutu obiektu `null`, wtedy jest rzucany obiekt klasy `Throwable`.
- Rzucenie takiego obiektu może prowadzić albo do przerwania wykonania programu, albo może być odpowiednio potraktowane.

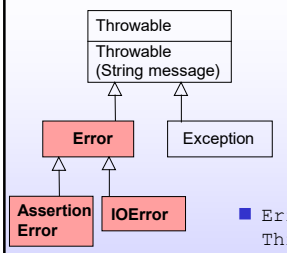
3

Wyjątki

- Rzucenie obiektu z klasy `Throwable` sygnalizuje pewną anomalię.
- Mechanizm wyjątków stosuje się do zaradzania pewnym sytuacjom; pozwala on też na oddzielenie kodu realizującego funkcjonalność od kodu odpowiadającego obsłudze błędów.
- Mechanizm błędów stosuje się do informowania o tym, że wystąpił błąd, którego program nie może usunąć.

4

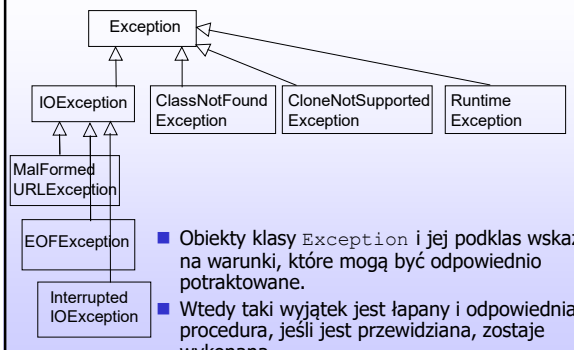
Throwable: Błędy i Wyjątki



- `Error`, czyli błąd, jest podklasą `Throwable`.
- Błąd oznacza poważny problem.
- Błędy nie powinny być łapane.
- W rezultacie metoda rzucająca błąd nie musi deklarować go w klauzuli `throws`.

5

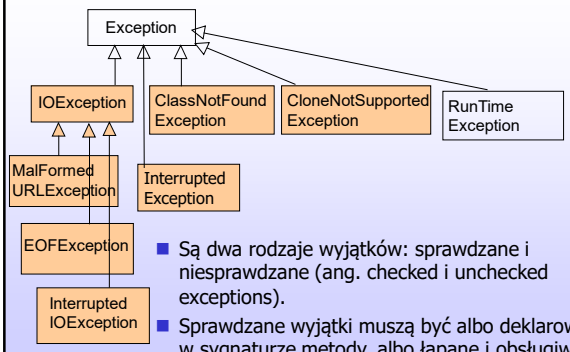
Wyjątki



- Obiekty klasy `Exception` i jej podklas wskazują na warunki, które mogą być odpowiednio potraktowane.
- Wtedy taki wyjątek jest łapany i odpowiednia procedura, jeśli jest przewidziana, zostaje wykonana.

6

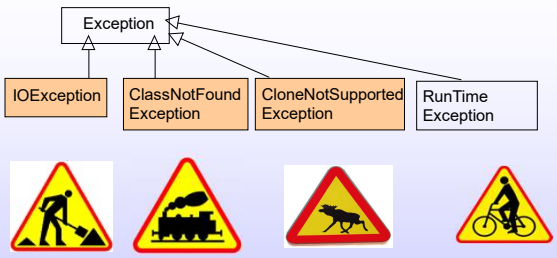
Sprawdzane Wyjątki



- Są dwa rodzaje wyjątków: sprawdzane i niesprawdzane (ang. checked i unchecked exceptions).
- Sprawdzane wyjątki muszą być albo deklarowane w sygnaturze metody, albo łapane i obsługiwane wewnątrz niej.

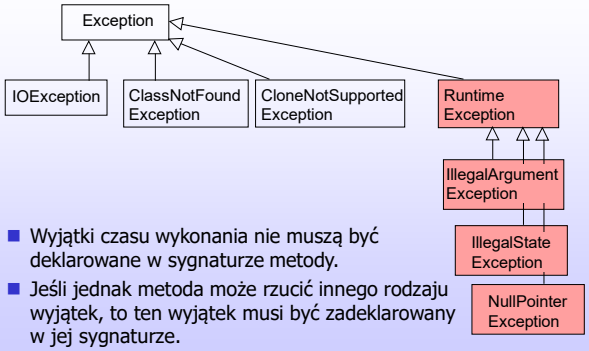
7

Wyjątki: sprawdzane i niesprawdzane



8

Wyjątki Czasu Wykonania



- Wyjątki czasu wykonania nie muszą być deklarowane w sygnaturze metody.
- Jeśli jednak metoda może rzucić innego rodzaju wyjątek, to ten wyjątek musi być zadeklarowany w jej sygnaturze.

9

Deklaracja Wyjątku (konto w banku cd.)

Deklaracja wyjątku nie jest potrzebna.

```
public void credit(double amount) {
  if(amount < 0)
     throw new IllegalArgumentException("amount < 0!");
  balance = balance + amount;
}
```

Nie ma klauzul `try` i `catch`.

Wiadomość o bledzie.

Ten wyjątek może być albo obsłużony przez metodę wywołującą, albo przekazany dalej.

10

Deklaracja Wyjątku Sprawdzanego w Sygnaturze Metody

Deklaracja wyjątku sprawdzanego w sygnaturze metody

```
public void run() throws InterruptedException {
      while (true) {
        System.out.println(word);
        sleep(delay);
      }
}
```

Nie ma klauzul `try` i `catch`, więc metoda wywołująca musi być przygotowana na obsługę.

`InterruptedException` musi być obsłużony przez metodę wywołującą lub zadeklarowany w jej sygnaturze.

11

Wyjątki: łapanie wyjątków w ciele metody

Tu może być rzucony wyjątek.

```
public void run() {
      try {
        while (true) {
          System.out.println(word);
          sleep(delay);
      }catch (InterruptedException e) {
        return;
      }
  }
```

Tu jest on łapany.

A tu jest on obsługiwany.

12

Finally

```
try { . . .  //execute method();
 } catch (ExceptionX e) {
  . . . //do something about it
 } catch (ExceptionY e) {
  . . . //do something about it
 } finally {
   . . . //do whatever
 }
```

Jest wykonywane niezależnie od tego, czy wyjątek zostanie rzucony.

A tu jest on obsługiwany.

Będzie wykonana pierwsza klauzula `catch`, która złapie rzucony wyjątek.

13

Wyjątki: obsługa

- Wyjątki są mechanizmem przekazania kontroli i informacji z pewnego punktu w programie do kodu obsługi wyjątków.

```
try {
  . . .  //execute method();
} catch (IllegalArgumentException e) {
  . . . //do something about it
} finally {
  . . . //do whatever
}
```

14

Użycie Wyjątku

```
public void transfer(BankAccount target, double amount) {
    this.debit(amount);
    target.credit(amount);
}
```

Te instrukcje mogą rzucać wyjątki.

Co z nimi zrobić?

Metoda wywołana przez inną metodę może rzucić wyjątek.

15

Obsługa Wyjątków

```
public void transfer(BankAccount target, double amount) throws
   Exception {
  final double oldBalanceThis = this.balance;
  final double oldBalanceTarget = target.balance;
  try{
    this.debit(amount);
    target.credit(amount);
  } catch (Exception e) {
    this.balance = oldBalanceThis;
    target.balance = oldBalanceTarget;
    System.out.println(e.toString());
   throw e;
  }
}
```

Zapisujemy wartość atrybutów.

Wykonujemy procedurę.

W razie rzucenia wyjątku przywracamy wartość atrybutom sprzed wykonania procedury.

Chcemy przywrócić stan sprzed rzucenia wyjątku.

16

Zwijanie stosu przy wyrzuceniu wyjątku

```
Exception in thread "main"
  java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
at java.util.ArrayList.rangeCheck(Unknown Source)
at java.util.ArrayList.get(Unknown Source)
at pseudo_states.src.Selected.cl(Selected.java:60)
at pseudo_states.src.Selected.<init>(Selected.java:39)
at pseudo_states.src.SemiStrSM.<init>(SemiStrSM.java:37)
at pseudo_states.src.Tester1.main(Tester1.java:66)
```

typ wyjątku

metoda wykonywana gdy wyjątek jest rzucony

Metoda na spodzie stosu

17

Dziedziczenie: Co będzie wynikiem kompilacji?

```
public class C {
  public void m() {
  }
}

public class D extends C {
  public void m() throws
     IOException {

    ...
  }
}
```

```
public class C {
  public void m()   throws
     IOException {
  }
}

public class D extends C {
  public void m() {
  }
}
```

Wynikiem będzie powstanie klas binarnych.

18

Wyjątki a Dziedziczenie

Podklasa musi zachowywać kontrakt nadklasy. W przypadku dziedziczenia metoda w podklasie może rzucać tylko takie sprawdzane wyjątki, które są deklarowane w nadklasie.

```
import java.io.IOException;
class Parent {
  public void m() throws IOException {
    //. . .
  }
}
```

```
import java.io.InterruptedIOException;
class Child2 extends Parent {
  public void m() throws InterruptedIOException {
    //. . .
  }
}
```

OK, rzuca wyjątek będący szczególnym przypadkiem wyjątku metody w nadklasie.

19

Pytanie: Co będzie wynikiem wykonania metody n?

```
public void m() {
  try {
     System.out.println("m throws runtime exception e.");
     if(2==1 + 1) throw new RuntimeException();
  } catch(RuntimeException e){
      System.out.println("m handles e.");
      throw e;
  }
  finally{
     System.out.println("m executes its finally-part.");
  }
  System.out.println("X");
}

public void n() {
  try {
     System.out.println("n calls m.");
     m();
  } catch(RuntimeException e){
     System.out.println("n handles e.");
  } finally{
     System.out.println("n executes its finally-part.");
  }
}
```


20

Odpowiedz

```
n calls m.
m throws rune time exception e.
m handles e.
m executes its finally-part.
n handles e.
n executes its finally-part.
```


21

JTP: Testowanie

dr hab. Piotr Kosiuczenko
prof. WAT

22

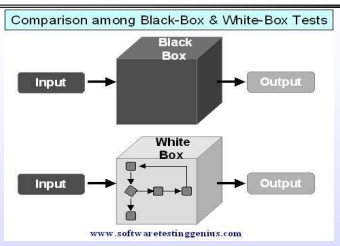
Poziomy testowania

Są zasadniczo cztery zasadnicze poziomy testowania, od najniższego licząc:

1) testy jednostkowe, np. typu czarnej skrzynki z użyciem JUnit (poziom implementacji)
2) testy integracyjne komponentów (poziom integracji)
3) testy systemowe (poziom systemu)
4) testy akceptacyjne (poziom akceptacji)

23

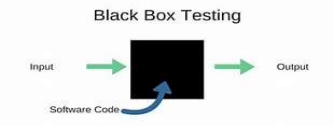
Testy jednostkowe: czarnej i białej skrzynki



- Metody testowania oprogramowania są zasadniczo dzielone na testowanie typu białej i czarnej skrzynki.
- Te dwa typy odpowiadają punkowi widzenia, jaki przyjmuje tester podczas projektowania przypadków testowych.
- W metodologii testowania oprogramowania można również zastosować podejście hybrydowe, zwane testowaniem "szaro-skrzynkowym".

24

Testowanie typu czarna skrzynka



- Testowanie "czarnej skrzynki" (CS), zwane też testowaniem funkcjonalnym, traktuje oprogramowanie jak "czarną skrzynkę", badając funkcjonalność bez żadnej wiedzy o implementacji, tj. bez wglądu w kod źródłowy.
- Testerzy wiedzą tylko, co oprogramowanie ma robić, a nie jak to robi.

25

Testowanie typu CS

- Przypadki testowe buduje się wokół specyfikacji i wymagań, czyli tego, co aplikacja ma robić.
- Do tworzenia przypadków testowych wykorzystuje się opisy użycia oprogramowania, w tym jego specyfikacje, związane z nim wymagania i projekty.
- Testowanie oparte na specyfikacji ma na celu sprawdzenie funkcjonalności oprogramowania zgodnie z odpowiednimi wymaganiami.
- Ten poziom testowania zazwyczaj wymaga dostarczenia testerowi dokładnych przypadków testowych, który następnie może po prostu sprawdzić, czy dla danego wejścia, wartość wyjściowa (lub zachowanie) "jest" lub "nie jest" taka sama jak oczekiwana wartość określona w przypadku testowym.

26

Testowanie funkcji: przypadki testowe

W najprostszym przypadku, gdy metoda jest funkcją, tj. nie zmienia stanu obiektów, przy testowaniu podajemy argumenty metody i wartości zwracane:

Dla wybranych argumentów możemy sprawdzić wartości, ale nie dla wszystkich

27

Testowanie funkcji: przypadki testowe

W najprostszym przypadku, gdy metoda jest funkcją, tj. nie zmienia stanu obiektów, przy testowaniu podajemy argumenty metody i wartości zwracane:

metoda	nazwa testu	wart. parametrów	oczekiwany wynik
square	testSquare0	0	0
	testSquare1	1	1
	testSquare2	2	4
	testSquare3	3	9
...			
factorial	testFactorial0	0	1
. . .			

28

JUnit

- *De facto* standard for developing unit tests in Java
  - One of the most important Java libraries ever developed
  - Made unit testing easy and popular among developers
- Two techniques:
  - Extending the TestCase class (prior to version 4)
  - Using Java annotations (after version 4)

29

JUnit: asercje i polecenia

`assertTrue( boolean )`	Asserts that a condition is true.
`fail( String )`	Asserts that a test fails, and prints the corresponding message.
`assertFalse(boolean )`	Asserts that a condition is false.
`assertEquals(Object, Object)`	Asserts that two objects are equal.
`assertNotNull( Object )`	Asserts that an object is *not* null.
`assertNull( Object )`	Asserts that an object is null.
`assertSame( Object, Object )`	Asserts that two references refer to the same object.
`assertNotSame( Object, Object )`	Asserts that two references do *not* refer to the same object.

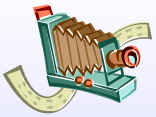
30

Przypadki testowe i odpowiadające im metody

metoda	nazwa testu	wart. parametrów	oczekiwany wynik
square	testSquare0	0	0
	testSquare1	1	1
	testSquare2	2	4
	testSquare3	3	9
	. . .	. . .	. . .

```
@Test
testSquare0{
   assertTrue( square(0) == 0 );
}
@Test
testSquare1{
   assertTrue( square(1) == 1 );
}
@Test
testSquare2{
   assertTrue( square(2) == 4 );
```

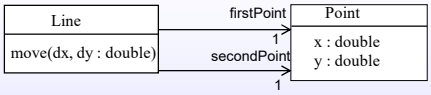

Testowanie zmiany stanu obiektów



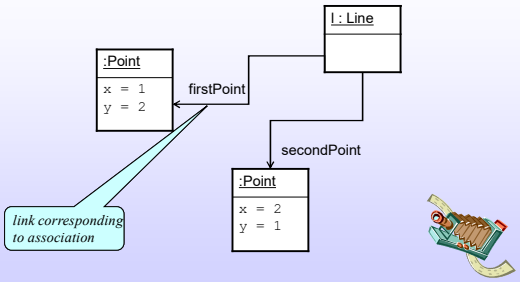
p1 : Point
x =10 y = 3

An object with certain attribute values at a given point in time.

Diagram Klas

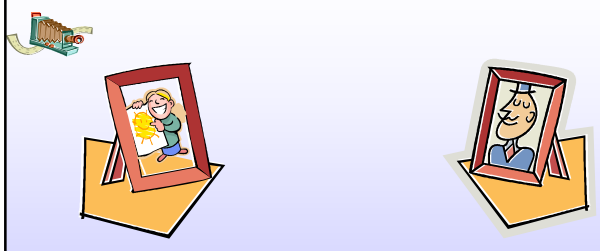


I odpowiadający mu diagram obiektowy



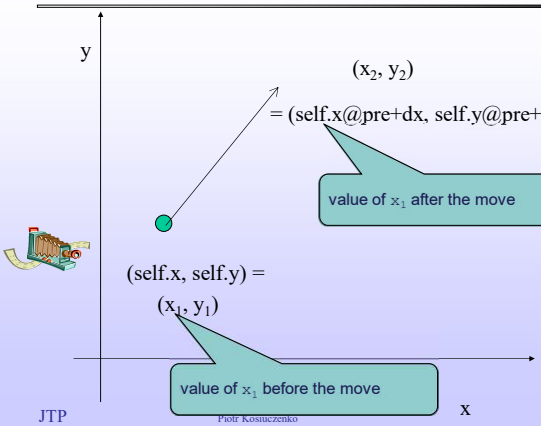
stan konkretnej linii `l` utrwalony w pewnym momencie

Snapshot Before and After (motivation)

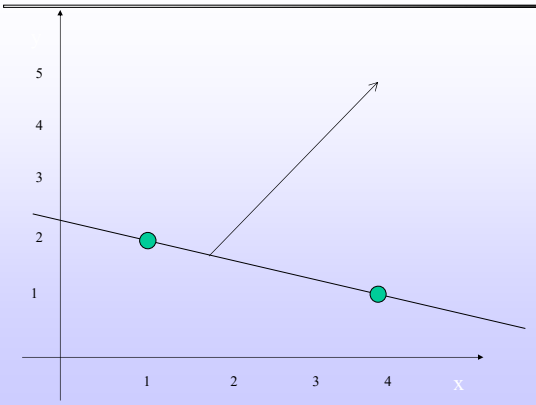


We can compare system states before and after a method execution.

Testowanie: przesunięcia punktu



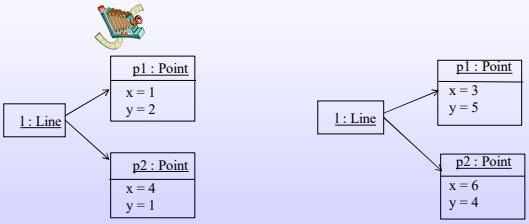
Przesuwanie linii



37

Diagram obiektowy

Objects with their attribute values and links form a system snapshot.

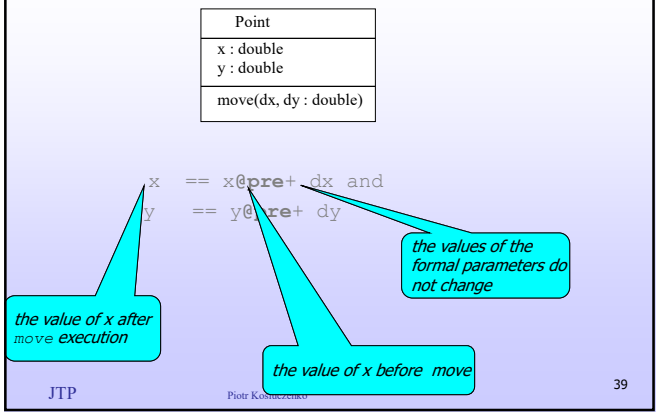


zdjęcie przed wykonaniem `move`

po wykonaniu `move`

38

Specification with Pre and Post-Conditions

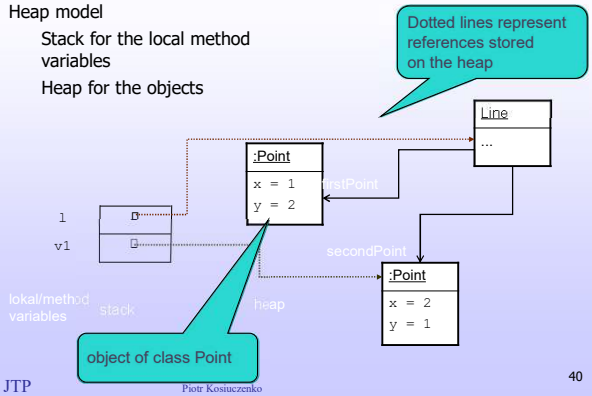


39

Heap (Machine Level Representation)

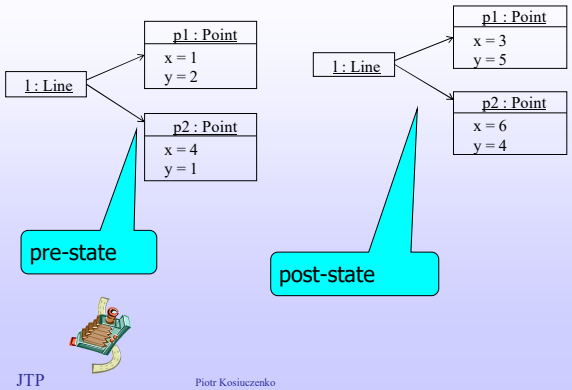
Heap model

- Stack for the local method variables
- Heap for the objects



40

Post-conditions



41

Testowanie stanu obiektów

- CD na laboratoriach.

42