Pertemuan VII

EXCEPTION DAN ASSERTIONS

7.1. Pendahuluan

7.1.1. *Bug* dan *Exception*

Bugs dan *error* dalam sebuah program sangat sering muncul meskipun program tersebut dibuat oleh programmer berkemampuan tinggi. Untuk menghindari pemborosan waktu pada proses *error-checking*, Java menyediakan mekanisme penanganan *exception*.

Exception adalah singkatan dari *Exceptional Events*. Kesalahan (*errors*) yang terjadi saat *runtime*, menyebabkan gangguan pada alur eksekusi program. Terdapat beberapa tipe *error* yang dapat muncul. Sebagai contoh adalah *error* pembagian 0, mengakses elemen di luar jangkauan sebuah array, *input* yang tidak benar dan membuka file yang tidak ada.

7.1.2. *Error* dan *Exception Classes*

Seluruh *exceptions* adalah *subclasses*, baik secara langsung maupun tidak langsung, dari sebuah *root class Throwable*. Kemudian, dalam *class* ini terdapat dua kategori umum, yaitu *Error class* dan *Exception class*.

Exception class menunjukkan kondisi yang dapat diterima oleh user program. Umumnya hal tersebut disebabkan oleh beberapa kesalahan pada kode program. Contoh dari *exceptions* adalah pembagian oleh 0 dan error di luar jangkauan array.

Error class digunakan oleh Java *run-time* untuk menangani *error* yang muncul pada saat dijalankan. Secara umum hal ini di luar kontrol *user* karena kemunculannya disebabkan oleh *run-time environment*. Sebagai contoh adalah *out of memory* dan *harddisk crash.*

7.1.3. Contoh Kesalahan/Error

Perhatikan contoh program berikut :

```
class DivByZero {
  public static void main(String args[]) {
    System.out.print("3/0 = ");
    System.out.println(3/0);
  }
}
```

Jika kode tersebut dijalankan, akan didapatkan pesan kesalahan sebagai berikut :

```
3/0 = Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivByZero.main(DivByZero.java:4)
```

Pesan tersebut menginformasikan tipe *exception* yang terjadi pada baris dimana *exception* itu berasal. Inilah aksi *default* yang terjadi bila terjadi *exception* yang tidak tertangani. Jika tidak terdapat kode yang menangani *exception* yang terjadi, aksi *default* akan bekerja otomatis. Aksi tersebut pertama-tama akan menampilkan deskripsi *exception* yang terjadi. Kemudian akan ditampilkan *stack trace* yang mengindentifikasi metode dimana *exception* terjadi. Pada bagian akhir, aksi *default* tersebut akan menghentikan program secara paksa.

Bagaimana jika kita ingin melakukan penanganan atas *exception* dengan cara yang berbeda? Untungnya, bahasa pemrograman Java memiliki 3 *keywords* penting dalam penanganan *exception*, yaitu *try, catch* dan *finally*.

7.2. Menangkap Exception

7.2.1. *Try … Catch*

Seperti yang telah dijelaskan sebelumnya, keyword *try, catch* dan *finally* digunakan dalam menangani bermacam tipe exception. 3 Keyword tersebut digunakan bersama, namun *finally* bersifat opsional. Akan lebih baik jika memfokuskan pada dua keyword pertama, kemudian membahas *finally* pada bagian akhir.

Berikut ini adalah penulisan *try-catch* secara umum :

```
try {
    <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
    <handler if ExceptionType1 occurs>
}
...
} catch (<ExceptionTypeN> <ObjName>) {
    <handler if ExceptionTypeN occurs>
}
```

Blok *catch* dimulai setelah kurung kurawal dari kode *try* atau *catch* terkait penulisan kode dalam blok yang dimasukkan.

Gunakan contoh kode tersebut pada program DivByZero yang telah dibuat sebelumnya :

```
class DivByZero {
  public static void main(String args[]) {
    try {
      System.out.print("3/0 = ");
      System.out.println(3/0);
     } catch (ArithmeticException exc) {
      //Reaksi atas kejadian
      System.out.println(exc);
    }
    System.out.println("Setelah Exception.");
  }
}
```

Kesalahan pembagian dengan bilangan 0 adalah salah satu contoh dari *ArithmethicException*. Tipe exception kemudian mengindikasikan klausa *catch* pada class ini. Program tersebut menangani kesalahan yang terjadi dengan menampilkan deskripsi dari permasalahan.

Output program saat eksekusi akan terlihat sebagai berikut :

```
3/0 = java.lang.ArithmeticException: / by zero
Setelah Exception.
```

Bagian kode yang terdapat pada blok *try* dapat menyebabkan lebih dari satu tipe *exception*. Dalam hal ini, terjadinya bermacam tipe kesalahan dapat ditangani menggunakan beberapa blok *catch.* Perlu dicatat bahwa blok *try* dapat hanya

menyebabkan sebuah *exception* pada satu waktu, namun dapat pula menampikan tipe *exception* yang berbeda di lain waktu.

Berikut adalah contoh kode yang menangani lebih dari satu exception :

```
class MultipleCatch {
  public static void main(String args[]) {
    try {
      int den = Integer.parseInt(args[0]); // baris ke-4
      System.out.println(3/den); // baris ke-5
    } catch (ArithmeticException exc) {
      System.out.println("Nilai pembagi 0.");
    } catch (ArrayIndexOutOfBoundsException exc2) {
      System.out.println("Parameter/argument tidak ada.");
    }
    System.out.println("Setelah exception.");
  }
}
```

Pada contoh ini, baris ke-4 akan menghasilkan kesalahan berupa *ArrayIndexOutOfBoundsException* bilamana seorang *user* alpa dalam memasukkan *argument*, sedang baris ke-5 akan menghasilkan kesalahan *ArithmethicException* jika pengguna memasukkan nilai 0 sebagai sebuah *argument*.

Contoh output/keluaran dari program diatas adalah :

```
>java MultipleCatch
Parameter/argument tidak ada.
Setelah exception.

>java MultipleCatch 0
Nilai pembagi 0.
Setelah exception.

>java MultipleCatch 3
1
Setelah exception.
```

Penggunaan *try* bersarang diperbolehkan dalam pemrograman Java.

```
class NestedTryDemo {
  public static void main(String args[]){
    try {
      int a = Integer.parseInt(args[0]);
      try {
        int b = Integer.parseInt(args[1]);
        System.out.println(a/b);
      } catch (ArithmeticException e) {
        System.out.println("Error, pembagi bernilai nol!");
      }
    } catch (ArrayIndexOutOfBoundsException e2) {
      System.out.println("Dibutuhkan dua parameter/argument!");
    }
  }
}
```

Contoh output/keluaran dari program diatas adalah :

```
>java NestedTryDemo 6
Dibutuhkan dua parameter/argument!

>java NestedTryDemo 6 0
Error, pembagi bernilai nol!

>java NestedTryDemo 6 2
3
```

Kode berikut menggunakan *try* bersarang tergabung dengan penggunaan method.

```
class NestedTryDemo2 {
  static void nestedTry(String args[]) {
    try {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      System.out.println(a/b);
    } catch (ArithmeticException e) {
      System.out.println("Error, pembagi bernilai nol!");
    }
  }

  public static void main(String args[]){
    try {
      nestedTry(args);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Dibutuhkan dua parameter/argument!");
    }
  }
}
```

Bagaimana output program tersebut jika diimplementasikan terhadap argument–argument berikut :

a. Tidak ada argumen
b. 15
c. 15 3
d. 15 0

7.2.2. *Keyword Finally*

Saatnya kita mengimplementasikan *finally* dalam blok *try-cacth*. Berikut ini cara penggunaan keyword tersebut :

```
try {
    <kode monitor exception>
} catch (<ExceptionType1> <ObjName>) {
    <penanganan jika ExceptionType1 terjadi>
}
...
} finally {
    <kode yang akan dieksekusi saat blok try berakhir>
}
```

Penggunaan *finally* dimulai setelah kurung kurawal penutup blok *catch* terkait. Blok *finally* mengandung kode penanganan setelah penggunaan *try* dan *catch.* Blok kode ini selalu tereksekusi walaupun sebuah *exception* terjadi atau tidak pada blok *try.* Blok kode tersebut juga akan menghasilkan nilai *true* meskipun *return, continue* ataupun *break*

tereksekusi. Terdapat 4 kemungkinan skenario yang berbeda dalam blok *try-catch-finally*. Pertama, pemaksaan keluar program terjadi bila control program dipaksa untuk melewati blok *try* menggunakan *return*, *continue* ataupun *break*. Kedua, sebuah penyelesaian normal terjadi jika *try-catch-finally* tereksekusi secara normal tanpa terjadi error apapun. Ketiga, kode program memiliki spesifikasi tersendiri dalam blok *catch* terhadap *exception* yang terjadi. Yang terakhir, kebalikan skenario ketiga. Dalam hal ini, *exception* yang terjadi tidak terdefinisikan pada blok *catch* manapun. Contoh dari skenario–skenario tersebut terlihat pada kode berikut ini :

```
class FinallyDemo {
  static void myMethod(int n) throws Exception{
    try {
      switch(n) {
        case 1: System.out.println("case pertama");
          return;
        case 3: System.out.println("case ketiga");
          throw new RuntimeException("demo case ketiga");
        case 4: System.out.println("case keempat");
          throw new Exception("demo case keempat");
        case 2: System.out.println("case Kedua");
      }
    } catch (RuntimeException e) {
      System.out.print("RuntimeException terjadi : ");
      System.out.println(e.getMessage());
    } finally {
      System.out.println("Blok finally");
    }
  }

  public static void main(String args[]){
    for (int i=1; i<=4; i++) {
      try {
        FinallyDemo.myMethod(i);
      } catch (Exception e){
        System.out.print("Exception terjadi : ");
        System.out.println(e.getMessage());
      }
      System.out.println();
    }
  }
}
```

Output/keluaran dari program diatas adalah :

```
>java FinallyDemo
case pertama
Blok finally

case Kedua
Blok finally

case ketiga
RuntimeException terjadi : demo case ketiga
Blok finally

case keempat
Blok finally
Exception terjadi : demo case keempat
```

7.3. Melempar Exception

7.3.1. *Keyword Throw*

Disamping menangkap exception, Java juga mengijinkan seorang user untuk melempar sebuah exception. Sintaks pelemparan exception cukup sederhana.

```
throw <exception object>;
```

Perhatikan contoh berikut ini.

```
/* Melempar exception jika terjadi kesalahan input */
class ThrowDemo {
  public static void main(String args[]){
    String input = "invalid input";
    try {
      if (input.equals("invalid input")) {
        throw new RuntimeException("throw demo");
      } else {
        System.out.println(input);
      }
      System.out.println("After throwing");
    } catch (RuntimeException e) {
      System.out.println("Exception caught here.");
      System.out.println(e);
    }
  }
}
```

Output/keluaran dari program diatas adalah :

```
Exception caught here.
java.lang.RuntimeException: throw demo
```

7.3.2. *Keyword Throws*

Jika sebuah metode dapat menyebabkan sebuah *exception* namun tidak menangkapnya, maka digunakan keyword *throws*. Aturan ini hanya berlaku pada *checked exception*. Kita akan mempelajari lebih lanjut tentang *checked exception* dan *unchecked exception* pada bagian selanjutnya (Kategori Exception).

Berikut ini penulisan *syntax* menggunakan keyword *throws* :

```
<type> <methodName> (<parameterList>) throws <exceptionList> {
    <methodBody>
}
```

Sebuah metode perlu untuk menangkap ataupun mendaftar seluruh *exceptions* yang mungkin terjadi, namun hal itu dapat menghilangkan tipe *Error*, *RuntimeException*, ataupun *subclass*-nya.

Contoh berikut ini menunjukkan bahwa method *myMethod* tidak menangani *ClassNotFoundException*.

```
class ThrowingClass {
  static void myMethod() throws ClassNotFoundException {
    throw new ClassNotFoundException ("just a demo");
  }
```

```
}

class ThrowsDemo {
  public static void main(String args[]) {
    try {
      ThrowingClass.myMethod();
    } catch (ClassNotFoundException e) {
      System.out.println(e);
    }
  }
}
```

7.4. Kategori *Exception*

7.4.1. *Exception Classes* dan Hierarki

Seperti yang disebutkan sebelumnya, *root class* dari seluruh *exception classes* adalah *Throwable class*. Yang disebutkan dibawah ini adalah *exception class* hierarki. Seluruh *exceptions* ini terdefinisi pada *package java.lang*.

Tabel 7.1. Hirarki *Exception Class*

Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException, (IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

Sekarang Anda sudah cukup familiar dengan beberapa *exception classes*, saatnya untuk mengenalkan aturan : *catch* lebih dari satu harus berurutan dari *subclass* ke *superclass*.

```
class MultipleCatchError {
  public static void main(String args[]){
    try {
      int a = Integer.parseInt(args [0]);
      int b = Integer.parseInt(args [1]);
      System.out.println(a/b);
    } catch (Exception e) {
      System.out.println(e);
    } catch (ArrayIndexOutOfBoundsException e2) {
      System.out.println(e2);
    }
```

```
    System.out.println("After try-catch-catch.");
  }
}
```

Setelah mengkompilasi kode tersebut akan menghasilkan pesan *error* jika *Exception class* adalah *superclass* dari *ArrayIndexOutOfBoundsException class*.

```
MultipleCatchError.java:9: exception ArrayIndexOutOfBoundsException has
already been caught
    } catch (ArrayIndexOutOfBoundsException e2) {
      ^
1 error
```

Agar tidak ada kesalahan, dapat dilakukan dengan menukar posisi catch.

7.4.2. *Checked* dan *Unchecked Exceptions*

Exception terdiri atas *checked* dan *unchecked exceptions*. *Checked exceptions* adalah *exception* yang diperiksa oleh Java *compiler*. *Compiler* memeriksa keseluruhan program apakah menangkap atau mendaftar *exception* yang terjadi dalam *sintax throws*. Apabila *checked exception* tidak didaftar ataupun ditangkap, maka *compiler error* akan ditampilkan.

Tidak seperti *checked exceptions*, *unchecked exceptions* tidak berupa *compile-time checking* dalam penanganan *exceptions*. Pondasi dasar dari *unchecked exception classes* adalah *Error*, *RuntimeException* dan *subclass*-nya.

7.4.3. User Defined Exceptions

Meskipun beberapa *exception classes* terdapat pada *package java.lang* namun tidak mencukupi untuk menampung seluruh kemungkinan tipe *exception* yang mungkin terjadi. Sehingga sangat mungkin bahwa kita perlu untuk membuat tipe *exception* tersendiri.

Dalam pembuatan tipe *exception*, kita hanya perlu untuk membuat sebuah *extended class* terhadap *RuntimeException class*, maupun *Exception class* lain. Selanjutnya tergantung pada kita dalam memodifikasi *class* sesuai permasalahan yang akan diselesaikan. *Members* dan *constructors* dapat dimasukkan pada *exception class* milik Anda.

Berikut ini contohnya :

```
class HateStringException extends RuntimeException{
  /* Tidak perlu memasukkan member ataupun construktor */
}

class TestHateString {
  public static void main(String args[]) {
    String input = "invalid input";
    try {
      if (input.equals("invalid input")) {
        throw new HateStringException();
      }
      System.out.println("String accepted.");
    } catch (HateStringException e) {
      System.out.println("I hate this string: " + input +".");
    }
  }
}
```

7.5. Assertions

7.5.1. User Defined Assertions

Assertions mengijinkan programmer untuk menentukan asumsi yang dihadapi. Sebagai contoh, sebuah tanggal dengan area bulan tidak berada antara 1 hingga 12 dapat diputuskan bahwa data tersebut tidak valid. Programmer dapat menentukan bulan harus berada diantara area tersebut. Meskipun hal itu dimungkinkan untuk menggunakan contructor lain untuk mensimulasikan fungsi dari assertions, namun sulit untuk dilakukan karena fitur assertion dapat tidak digunakan. Hal yang menarik dari assertions adalah seorang user memiliki pilihan untuk digunakan atau tidak pada saat runtime.

Assertion dapat diartikan sebagai extensi atas komentar yang menginformasikan pembaca kode bahwa sebagian kondisi harus terpenuhi. Dengan menggunakan assertions, maka tidak perlu untuk membaca keseluruhan kode melalui setiap komentar untuk mencari asumsi yang dibuat dalam kode. Namun, menjalankan program tersebut akan memberitahu Anda tentang assertion yang dibuat benar atau salah. Jika assertion tersebut salah, maka AssertionError akan terjadi.

7.5.2. Mengaktifkan dan Menonaktifkan Assertions

Penggunaan *assertions* tidak perlu melakukan *import package java.util.assert*. Menggunakan *assertions* lebih tepat ditujukan untuk memeriksa parameter dari *nonpublic methods* jika *public methods* dapat diakses oleh *class* lain. Hal itu mungkin terjadi bila penulis dari *class* lain tidak menyadari bahwa mereka dapat menonaktifkan *assertions*. Dalam hal ini program tidak dapat bekerja dengan baik.

Pada *non-public methods*, hal tersebut tergunakan secara langsung oleh kode yang ditulis oleh programmer yang memiliki akses terhadap metode tersebut. Sehingga mereka menyadari bahwa saat menjalankannya, *assertion* harus dalam keadaan aktif.

Untuk mengkompilasi file yang menggunakan *assertions*, sebuah tambahan parameter perintah diperlukan seperti yang terlihat dibawah ini :

```
javac –source 1.4 MyProgram.java
```

Jika kita ingin untuk menjalankan program tanpa menggunakan fitur *assertions*, cukup jalankan program secara normal.

```
java MyProgram
```

Namun, jika kita ingin mengaktifkan *assertions*, kita perlu menggunakan parameter –enableassertions atau –ea.

```
java –enableassertions MyProgram
```

7.5.3. Sintaks Assertions

Penulisan *assertions* memiliki dua bentuk.

a. Bentuk yang paling sederhana terlihat sebagai berikut :

```
assert <expression1>;
```

dimana <expression1> adalah kondisi dimana *assertion* bernilai *true*.

b. Bentuk yang lain menggunakan dua ekspresi, berikut ini cara penulisannya :

```
assert <expression1> : <expression2>;
```

dimana <expression1> adalah kondisi *assertion* bernilai *true* dan <expression2> adalah informasi yang membantu pemeriksaan mengapa program mengalami kesalahan.

Contoh penggunaan assertion :

```
class AgeAssert {
  public static void main(String args[]) {
    int age = Integer.parseInt(args[0]);
    assert(age>0);
    /* jika masukan umur benar (misal, age>0) */
    if (age >= 18) {
      System.out.println("Congrats! You're an adult!");
    }
  }
}
```

Referensi:

1. Hariyanto, Bambang, (2007), *Esensi-esensi Bahasa Pemrograman Java*, Edisi 2, Informatika Bandung, November 2007.
2. Utomo, Eko Priyo, (2009), *Panduan Mudah Mengenal Bahasa Java*, Yrama Widya, Juni 2009.
3. Tim Pengembang JENI, JENI 1-6, Depdiknas, 2007