# BAB IV
# ERROR HANDLING, LOGGING, AND DEBUGGING

## 4.1. Exception Handling / Error Handling

Pada kondisi tertentu, kode program yang ditulis oleh pemrogram atau *programmer* terkadang menghasilkan suatu kesalahan karena melanggar aturan penulisan kode program, seperti kesalahan penulisan *value* yang tidak memenuhi kaidah penulisan tipe data atau literal, kesalahan *casting* tipe data, file tidak ditemukan, index array yang tidak sesuai, dan bahkan kesalahan yang dihasilkan dari pengguna itu sendiri (*human error*). Namun demikian, seorang pemrogram akan sangat kesulitan menemukan kesalahan jika proses inspeksi *error* dilakukan secara manual. Ketika hal ini terjadi, maka diperlukan suatu penanganan khusus dalam mengatasinya, agar seorang pemrogram dapat mengetahui secara akurat dimana letak kesalahan pada bagian kode program, dan apa penyebab kesalahan tersebut.

Dalam bahasa pemrograman Java, pengecualian atau biasa kita sebut *Java Exception* merupakan suatu objek yang dapat menangkap, memproses dan mendeskripsikan suatu kondisi kesalahan atau *error* pada bagian kode program. Ketika kondisi ini muncul, maka objek *Exception* dibuat dan dilempar atau dikembalikan ke dalam metode yang menyebabkan kesalahan. Metode tersebut dapat memilih untuk menangani kesalahan/*error* itu sendiri, atau meneruskannya (ke pemanggil suatu *method*). Penanganan kesalahan/*error* ini dapat dibuat oleh sistem run-time Java, atau dapat kita buat sendiri secara manual dengan kode program sendiri.

Penanganan kesalahan ataupun error handling dapat dikelola menggunkan lima kata kunci, yaitu: *try*, *catch*, *throw*, *throws*, and *finally*. Kode program atau *statement* yang ingin kita pantau (jika terjadi kesalahan) kita letakkan di dalam blok *try*, jika terdapat kesalahan pada kode program dalam blok tersebut, maka kita dapat menangkap kesalahan dengan menggunakan kata kunci *catch*. Untuk membuat exception secara manual (misalkan dengan Bahasa kita sendiri), kita dapat menggunakan kata kunci *throw*. Sedangkan jika ingin mengeluarkan atau mengembalikan exception/kesalahan dari sebuah method, maka kita dapat

menggnakan kata kunci *throws*. Kode program yang ingin kita eksekusi (baik terjadi kesalahan atau tidak terjadi kesalahan pada blok *try*) kita letakkan pada blok *finally*.

Berikut merupakan bentuk umum sebuah blok untuk penanganan kesalahan atau pengecualian:

```
try {
    //blok kode program yang ingin di-monitor kesalahannya
} catch (Exception e) {
    //penangan kesalahan/pengecualian (sesuai tipenya)
} finally{
    //blok kode program yang ingin di-eksekusi setelah
    //kode program pada blok try selesai dieksekusi
}
```

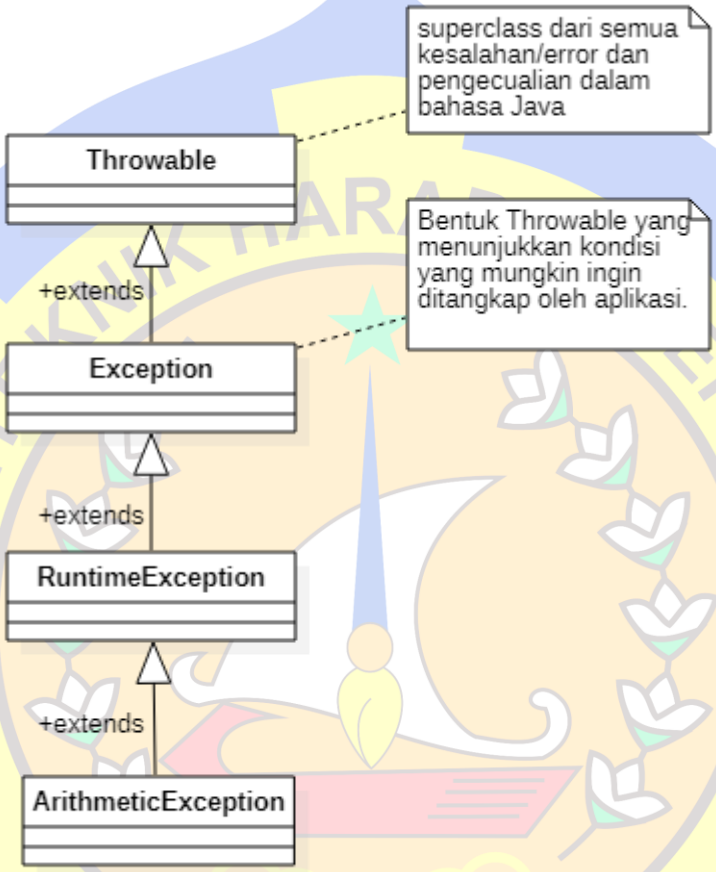
**Contoh:**

```
1   package except;
2
3   /**
4    *
5    * @author nishom
6    */
7   public class Contoh1 {
8
9       public static void main(String[] args) {
10          try {
11              int divider = 0;
12              int a = 10 / divider;
13              System.out.println(a);
14          } catch (Exception e) {
15              System.err.println("Error 001: " + e);
16          } finally {
17              System.out.println("\u2192 Done!");
18          }
19      }
20  }
```

Output:

```
compile-single:
run-single:
→ Done!
Error 001: java.lang.ArithmeticException: / by zero
BUILD SUCCESSFUL (total time: 1 second)
```

Pada baris kode ke-12, menghasilkan sebuah kesalahan/error, karena nilai pembagi adalah 0. Pesan kesalahan tersebut dihasilkan oleh kelas *ArithmeticException*. Seara otomatis blok *catch* dapat menentukan objek yang sesuai untuk menangani kesalahan/error. Hal ini dikarenakan kelas yang kita gunakan untuk menangkap kesalahan dalam blok *catch* adalah kelas *Exception* atau super-class dari kelas *ArithmeticException*. Lebih jelas lagi lihat pada pewarisan berikut:




Semua subclass dan detail dari kelas *Exception* dapat Anda lihat melalui laman berikut: https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

## 4.2. Exception Types

Pengecualian atau *Exception* di java terbagi menjadi dua cabang, yaitu:

1. **Exception**

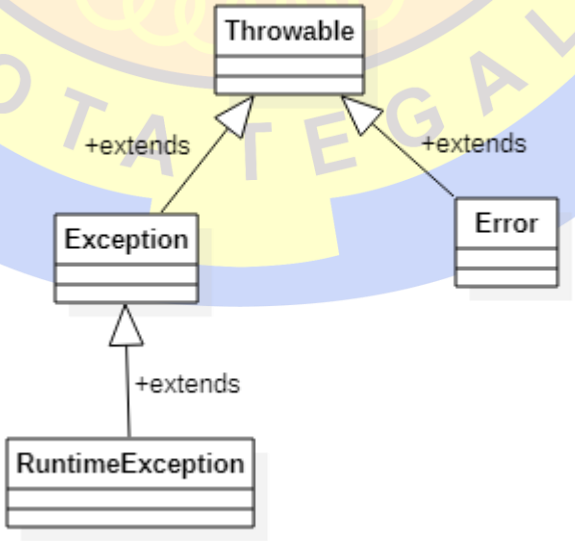
   Kelas ini digunakan untuk kondisi kesalahan tertentu yang harus ditangkap oleh program pengguna. Kelas ini juga dapat digunakan untuk membuat jenis pengecualian buatan kita sendiri. Ada subclass penting dari Exception,

biasa disebut RuntimeException. Pengecualian jenis ini secara otomatis ditentukan untuk program yang kita tulis dan menyertakan hal-hal seperti pembagian dengan nol dan pengindeksan array yang tidak valid.

2. **Error**

Mendefinisikan pengecualian yang tidak diharapkan untuk ditangkap dalam keadaan normal oleh sebuah program. Pengecualian tipe *Error* digunakan oleh sistem *run-time* Java untuk menunjukkan error yang berkaitan dengan lingkungan *run-time* itu sendiri. Stack overflow adalah contoh kesalahan tersebut. Pengecualian jenis ini biasanya dibuat sebagai respons terhadap kegagalan katastropik (*catastrophic*) yang biasanya tidak dapat ditangani oleh program kita sendiri. Contoh: program yang kita buat mencoba mengalokasikan memori dari *Java Virtual Machine* (JVM) namun memori yang tersedia tidak cukup untuk memenuhi permintaan tersebut. Contoh Error lainnya adalah kondisi dimana program kita mencoba untuk memuat sebuah kelas melalui metode `Class.forName()`, namun kelas tersebut ternyata rusak.

Pilihan pang tepat untuk memastikan bahwa program kita dapat berjalan dengan baik adalah dengan selalu melakukan monitoring pada setiap kode program yang akan kita eksekusi dengan memanfaatkan lima kata kunci untuk pengecekan kesalahan yang telah kita bahas di awal bab ini. Adapun hierarki pengecualian tingkat atas adalah seperti gambar berikut:



### 4.3. Logging

Standar dari sebuah aplikasi adalah memiliki sebuah log. Proses pencacatatan aktivitas atau pesan log selama eksekusi program ke suatu tempat ini biasa disebut *logging*. Pencatatan log ini memungkinkan kita untuk melaporkan dan mempertahankan pesan kesalahan dan peringatan serta pesan info sehingga pesan tersebut nantinya dapat diambil dan dianalisis. Java menyediakan fitur (*Java Logging API*) yang memungkinkan kita untuk mengkonfigurasi jenis pesan yang akan ditulis. Paket `java.util.logging` menyediakan kemampuan logging melalui kelas `Logger`. Ada beberapa alasan mengapa kita mungkin perlu merekam aktivitas aplikasi yang kita kembangkan, yaitu:

1. Mencatat keadaan yang tidak biasa atau kesalahan yang mungkin terjadi dalam program.
2. Mendapatkan info tentang apa yang terjadi di aplikasi untuk pengembangan aplikasi.

Detail yang dapat diperoleh dari log dapat bervariasi. Terkadang, kita mungkin menginginkan banyak detail tentang masalah tersebut, atau terkadang hanya beberapa informasi ringan. Seperti saat aplikasi sedang dalam pengembangan dan sedang menjalani fase pengujian, kita mungkin perlu menangkap banyak detail.

Level log berfungsi mengontrol detail logging, pelevelan tersebut menentukan sejauh mana kedalaman file log dihasilkan. Setiap level dikaitkan dengan angka dan terdapat 7 (tujuh) level log dasar dan 2 (dua) level khusus. Kita perlu menentukan tingkat pencatatan yang diinginkan setiap kali, kita berusaha berinteraksi dengan sistem log. Level logging dasar adalah sebagai berikut:

| LEVEL | VALUE | DETAIL/FUNGSI | KELOMPOK |
|---|---|---|---|
| **OFF** | 2147483647 | Level special yang digunakan untuk menon-aktifkan logging. | Spesial |
| **ALL** | -2147483648 | Level ini menunjukkan bahwa semua pesan harus dicatat, sepeti deklarasi field/atribut, pemanggilan metode, setiap penugasan, dan lain sebagainya. | Spesial |
| **SEVERE** | 1000 | Mengindikasikan kegagalan/kesalahan serius dan memungkinkan aplikasi tidak dapat melanjutkan lebih jauh. | Basic |
| **WARNING** | 900 | Berpotensi memunculkan masalah, contoh: seperti beberapa kali pengguna memberikan masukan atau kredensial yang salah. | Basic |

| LEVEL | VALUE | DETAIL/FUNGSI | KELOMPOK |
|---|---|---|---|
| **INFO** | 800 | Informasi umum, untuk penggunaan administrator atau pengguna tingkat lanjut. | Basic |
| **CONFIG** | 700 | Informasi pengaturan statis, seperti CPU apa yang menjalankan aplikasi, berapa banyak disk dan ruang memori, dan sebagainya. | Basic |
| **FINE** | 500 | Digunakan untuk memberikan informasi secara umum | Basic |
| **FINER** | 400 | Digunakan untuk memberikan informasi secara lebih detail | Basic |
| **FINEST** | 300 | Digunakan untuk memberikan informasi khusus (sangat detail) | Basic |

Contoh:

```
1   package logging;
2
3   import java.util.logging.Level;
4   import java.util.logging.Logger;
5
6   /**
7    *
8    * @author nishom
9    */
10  public class MyLogger {
11      static final Logger LOGGER = Logger.getLogger(MyLogger.class.getName());
12
13      public static void main(String[] args) {
14          try {
15              // atur LogLevel ke "Severe", hanya pesan severe yang akan ditulis
16              LOGGER.setLevel(Level.SEVERE);
17              LOGGER.severe("Severe Log 1");
18              LOGGER.warning("Warning Log");
19              LOGGER.info("Info Log");
20              LOGGER.finest("Finest Log");
21
22              // atur LogLevel ke "Info"
23              // hanya Log severe, warning dan info akan ditulis
24              LOGGER.setLevel(Level.INFO);
25              LOGGER.severe("Severe Log 2");
26              LOGGER.warning("Warning Log");
27              LOGGER.info("Info Log");
28              LOGGER.finest("Finest Log");
29              int a = 1/0;
30              System.out.println(a);
31          } catch (ArithmeticException e) {
32              // Membuat pesan log (level Severe)
33              LOGGER.severe("Severe Log 3: ".concat(e.getMessage()));
34          }
35      }
36  }
```

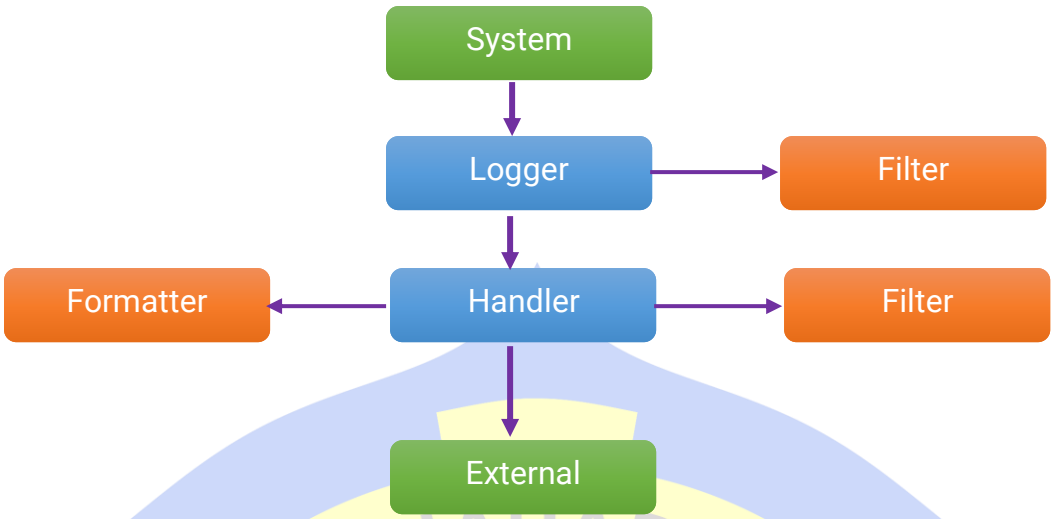
Output:

```
compile-single:
run-single:
Nov 24, 2020 1:26:48 AM logging.MyLogger main
SEVERE: Severe Log 1
Nov 24, 2020 1:26:48 AM logging.MyLogger main
SEVERE: Severe Log 2
Nov 24, 2020 1:26:48 AM logging.MyLogger main
WARNING: Warning Log
Nov 24, 2020 1:26:48 AM logging.MyLogger main
INFO: Info Log
Nov 24, 2020 1:26:48 AM logging.MyLogger main
SEVERE: Severe Log 3: / by zero
BUILD SUCCESSFUL (total time: 1 second)
```

Pada gambar hasil keluaran program dapat kita lihat bahwa Ketika logger diatur ke level Sereve (baris kode ke-16), maka semua logger yang nilainya di bawah sereve secara otomatis tidak akan proses maupun ditampilkan pada konsol. Begitupula Ketika logger diatur ke level Info (baris kode ke-24), maka pesan level info dan semua level di atas Info (Warning dan Sereve) akan ditulis. Selanjutnya, karena terdapat statement atau baris kode yang menugaskan sebuah operasi pembagian, dimana memungkinkan terjadinya kesalahan, maka diberikan logger pada blok catch agar dapat menangkap dan menampilkan kesalahan yang ditemukan. Namun demikian, jika kita ingin menampilkan semua jenis logger, maka kita cukup mengatur level menjadi `Level.ALL`.

Selain menulis pesan ke dalam konsol, logger juga dapat menulis pesan ke dalam file dengan berbagai format file, seperti html, xml, text, bahkan json sesuai dengan kebutuhan pengembang aplikasi.

Pada proses logging, objek `Logger` dialokasikan dengan objek `LogRecord` yang menyimpan pesan untuk dicatat. Objek `LogRecord` ini diteruskan ke semua *handler* yang ditetapkan ke objek `Logger`. Baik pencatat (*loggers*) maupun penangan (*handlers*) dapat secara opsional menggunakan `Filter` yang terkait dengannya, untuk memfilter pesan log. Kemudian, *handler* menerbitkan pesan yang dicatat ke sistem eksternal (baik konsol maupun berupa file yang tersimpan pada media penyimpanan). Kerangka logging dapat dilihat pada gambar di bawah ini:



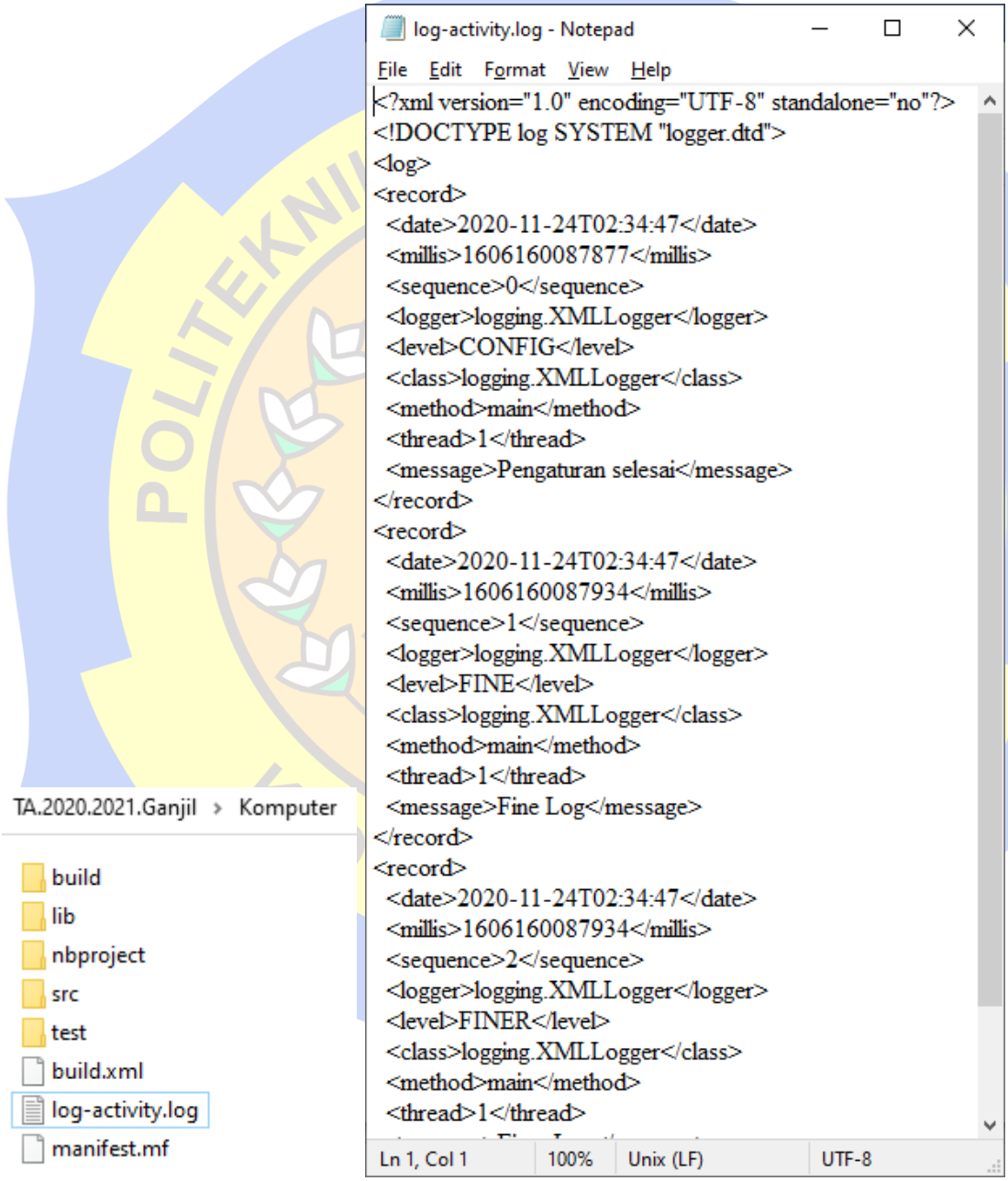

Contoh: Logger → handler (ke dalam konsol dan log file, xml format)

```
1  package logging;
2
3  import java.io.IOException;
4  import java.util.logging.ConsoleHandler;
5  import java.util.logging.FileHandler;
6  import java.util.logging.Handler;
7  import java.util.logging.Level;
8  import java.util.logging.Logger;
9
10 /**
11  *
12  * @author nishom
13  */
14 public class XMLLogger {
15     static final Logger LOGGER = Logger.getLogger(XMLLogger.class.getName());
16
17     public static void main(String[] args) {
18         Handler consoleHandler;
19         Handler fileHandler;
20         try{
21             consoleHandler = new ConsoleHandler();
22             fileHandler  = new FileHandler("log-activity.log");
23             LOGGER.addHandler(consoleHandler);
24             LOGGER.addHandler(fileHandler);
25
26             consoleHandler.setLevel(Level.ALL);
27             fileHandler.setLevel(Level.ALL);
28             LOGGER.setLevel(Level.ALL);
29
30             LOGGER.config("Pengaturan selesai");
31             LOGGER.removeHandler(consoleHandler);
32             LOGGER.log(Level.FINE, "Fine Log");
33             LOGGER.finer("Finer Log");
34         }catch(IOException e){
35             LOGGER.log(Level.SEVERE, "IOException", e);
36         }
37     }
38 }
```

Output (console log):

```
compile-single:
run-single:
Nov 24, 2020 2:34:47 AM logging.XMLLogger main
CONFIG: Pengaturan selesai
BUILD SUCCESSFUL (total time: 2 seconds)
```

Output (File Log):



Penjelasan:

- ✓ Baris kode ke-15 merupakan pembuatan objek LOGGER yang diberi nama sesuai nama kelas.
- ✓ Baris 18-19 merupakan deklarasi variable dengan tipe Handler, dimana kedua variable ini nantinya akan diinisialisasi dengan tipe handler-nya masing-masing.
- ✓ Baris kode ke 20-36 merupakan percobaan untuk monitoring kode program yang akan dieksekusi dengan menerapkan kata kunci `try` dan `catch`. Baris kode 21-33 merupakan kode program yang akan dimonitor ketika dieksekusi. Sedangkan baris 35 merupakan merupakan perintah untuk melakukan logging dengan level sereve jika pada blok `try` ditemukan kesalahan.
- ✓ Baris kode ke-21 merupakan instansiasi objek `consoleHandler`.
- ✓ Baris kode ke-22 merupakan instansiasi objek `fileHandler`, dengan konfigurasi penamaan *file* yang dihasilkan.
- ✓ Baris ke-23 menerapkan log handler ke objek LOGGER agar dapat menerima pesan dan menampilkannya ke layer (konsol).
- ✓ Baris ke-24 menerapkan log handler ke objek LOGGER agar dapat menerima pesan dan menuliskannya ke dalam *file*.
- ✓ Baris ke-26 merupakan pengaturan level logging pada `consoleHandler`. Level yang diterapkan adalah ALL, yang menandakan bahwa semua jenis pesan log akan ditulis pada layer konsol.
- ✓ Baris ke-27 merupakan pengaturan level logging pada `fileHandler`. Level yang diterapkan adalah ALL, yang menandakan bahwa semua jenis pesan log akan ditulis pada file yang telah ditentukan.
- ✓ Baris ke-28 merupakan pengaturan level logging ke `ALL`, agar semua pesan dapat diterima dan dicatat.
- ✓ Baris ke-30 merupakan pencatatan pesan log dengan level config.
- ✓ Baris ke-31 merupakan perintah untuk menghapus handler konsol, agar log apapun yang dituliskan berada di bawah kode ini tidak akan tercetak lagi pada konso, namun handler file tetap mencatat pesan ke dalam file.
- ✓ Exception pada clause blok `catch` berupa `IOException` merupakan tipe pengecualian yang digunakan untuk mendapatkan/menangkap informasi kesalahan input/output (I/O) yang mungkin terjadi.

## 4.4. Debugging

Secara umum, debugging merupakan proses mendeteksi dan memperbaiki kesalahan/error dalam suatu program. Keterampilan ini harus dimiliki oleh setiap pengembang Java karena membantu menemukan bug kecil yang tidak terlihat selama peninjauan kode atau yang hanya terjadi ketika kondisi tertentu terjadi saja. Dengan melakukan debugging, maka kemungkinan akan kita temukan berbagai jenis kesalahan. Beberapa di antaranya mudah ditangkap, seperti kesalahan sintaks, karena ditangani oleh kompilator. Kasus mudah lainnya adalah ketika kesalahan dapat dengan cepat diidentifikasi dengan melihat `stackTrace`, yang dengan mudah membantu kita mengetahui di mana kesalahan terjadi.

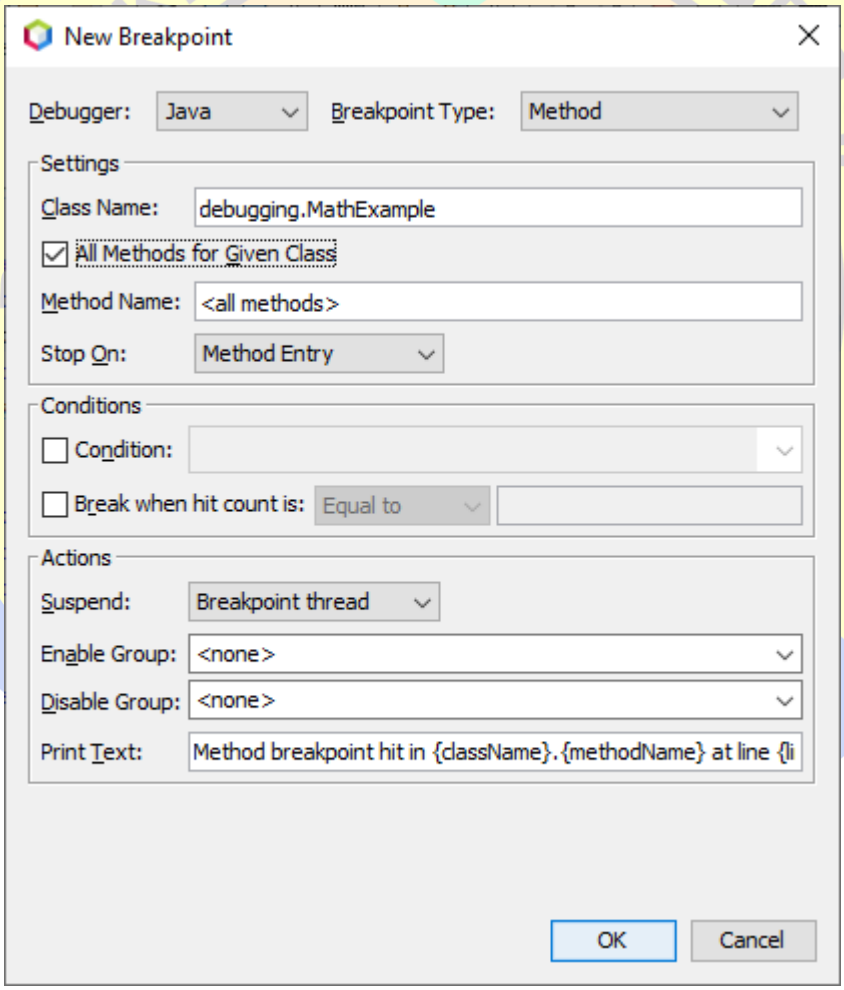
Namun, ada kesalahan yang bisa sangat rumit dan membutuhkan waktu lama untuk ditemukan dan diperbaiki. Misalnya, kesalahan logika yang hampir tidak dapat kita lihat, yang biasanya terjadi di awal program dan tidak muncul hingga sangat terlambat, dan terkadang merupakan tantangan yang sebenarnya bagi kita programmer untuk menyelesaikan masalah tersebut. Pada kondisi inilah debugger sangat tepat untuk digunaka, karena selain dapat menemukan kesalahan dengan akurat, debugging juga dapat melakukannya dengan cepat, sehingga sangat meringankan kerja programmer. Berikut merupakan contoh proses debugging tingkat dasar, pertama kita buat kode program yang akan kita jadikan kasus debugging.

```
1   package debugging;
2
3   /**
4    *
5    * @author nishom
6    */
7   public class MathExample {
8       public static void main(String[] args) {
9           String[] transkrip = {"82","84.5","90","89"};
10          double avg = findAverage(transkrip);
11          System.out.println("Rata-rata: " + avg);
12      }
13
14      private static double findAverage(String[] input) {
15          double result = 0;
16          for (String s : input) {
17              result += Double.valueOf(s);
18          }
19          return result;
20      }
21  }
```

Pada kode program di atas, terdapat 1 (satu) method yang telah kita buat, yang digunakan untuk menghitung nilai rata-rata. Method ini dipanggil pada method main, dengan parameter aktual berupa array yang berisikan item-item berupa nilai kuliah, bertipe string namun berisikan bilangan bulat dan pecahan.

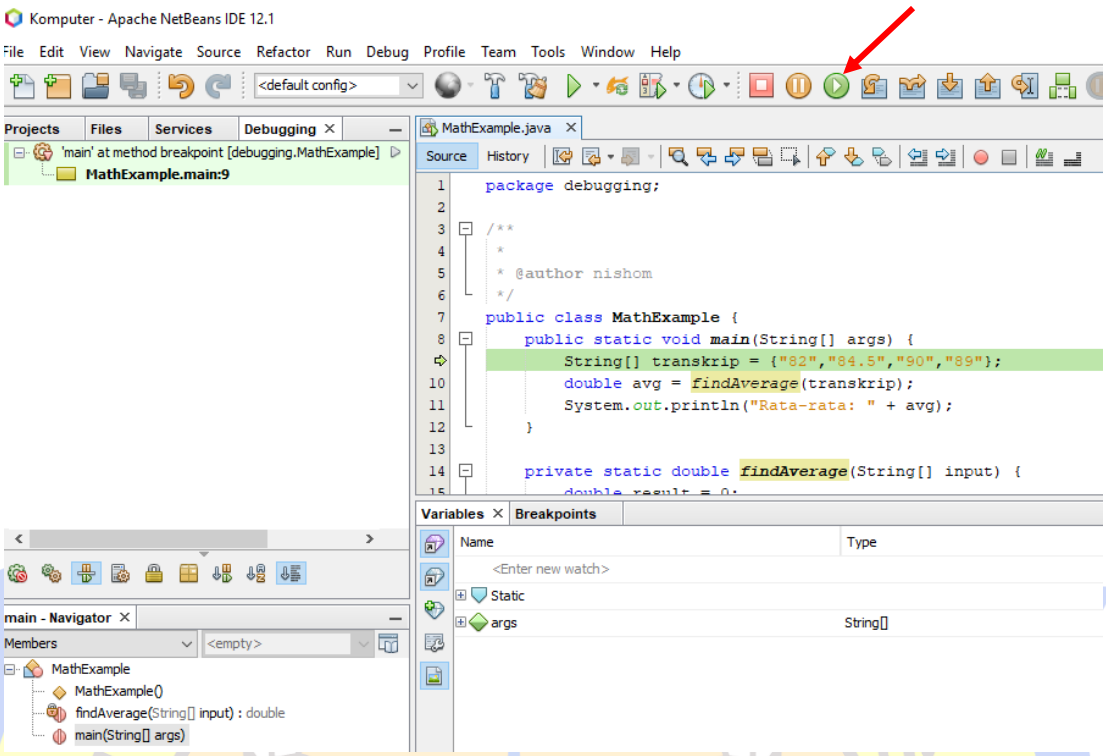
Untuk memeriksa bagaimana program beroperasi pada sebuah *runtime*, kita perlu menangguhkan eksekusinya sebelum bagian kode yang dicurigai. Hal tersebut dapat dilakukan dengan mengatur breakpoint. Breakpoint menunjukkan baris kode di mana program akan ditangguhkan agar kita dapat memeriksa statusnya. Di IDE Netbeans, untuk mengatur breakpoint dapat dilakukan dengan cara berikut:

- ✓ Pilih menu **Debug** → **New Breakpoint…**
- ✓ Pada blok **Settings**: isi Class Name → beri tanda centang pada "All Methods…" → Tekan tombol **OK**.

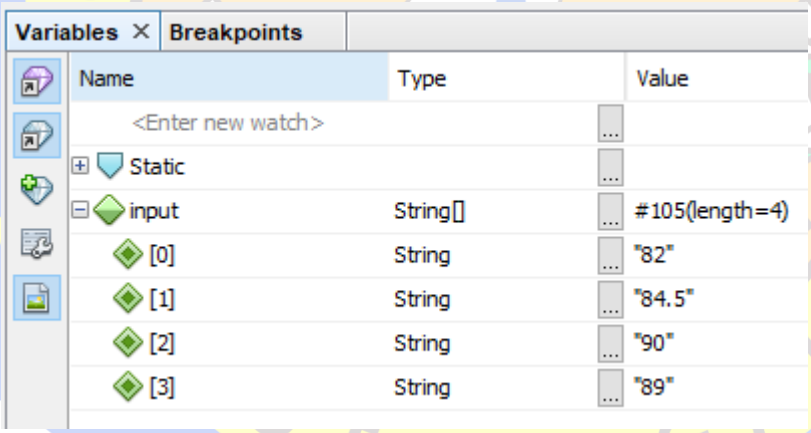



- ✓ Pilih menu **Debug** → **Debug File** (untuk debugging 1 kelas pada tab yang aktif pada editor netbeans); atau
- ✓ Pilih menu **Debug** → Debug Project (untuk debugging project)

✓ Klk tombol continue untuk melakukan dan melanjutkan debugging.




✓ Selama debugging, kita juga bisa mendapatkan informasi tentang semua variabel yang saat ini berada dalam lingkup di panel `Variable`.




## 4.5. Practice

Soon…

## 4.6. Exercise

Soon…