PV168

Logging, Errors and Transactions

General debugging

- What is your go-to debugging method?
- Basic printed statements
- Debugger

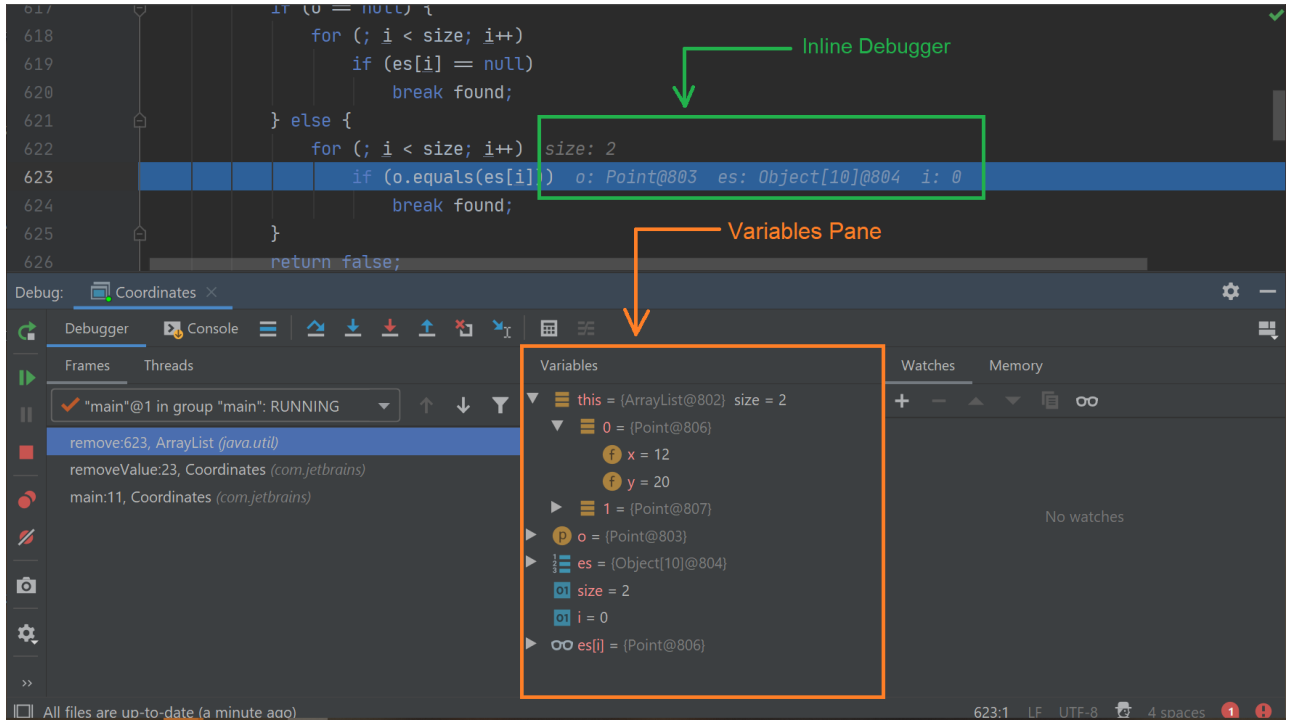
General debugging

- Printed statements are the easiest
- Don't even require much knowledge of the language
- Often effective for simple problems

```
static void transferMoney(User user1, User user2, int amount) {

    SomethingChecker.checkSomething(user1, user2);
    user1.deduct(amount);
    user1.doSomethingElse();

    System.out.println("Did we get here?");

    user2.add(amount);
    user2.doSomeCheck();
  }
```

General debugging

- Debugger is the most powerful
- Not always trivial to set up
- Generally want to avoid having to use it

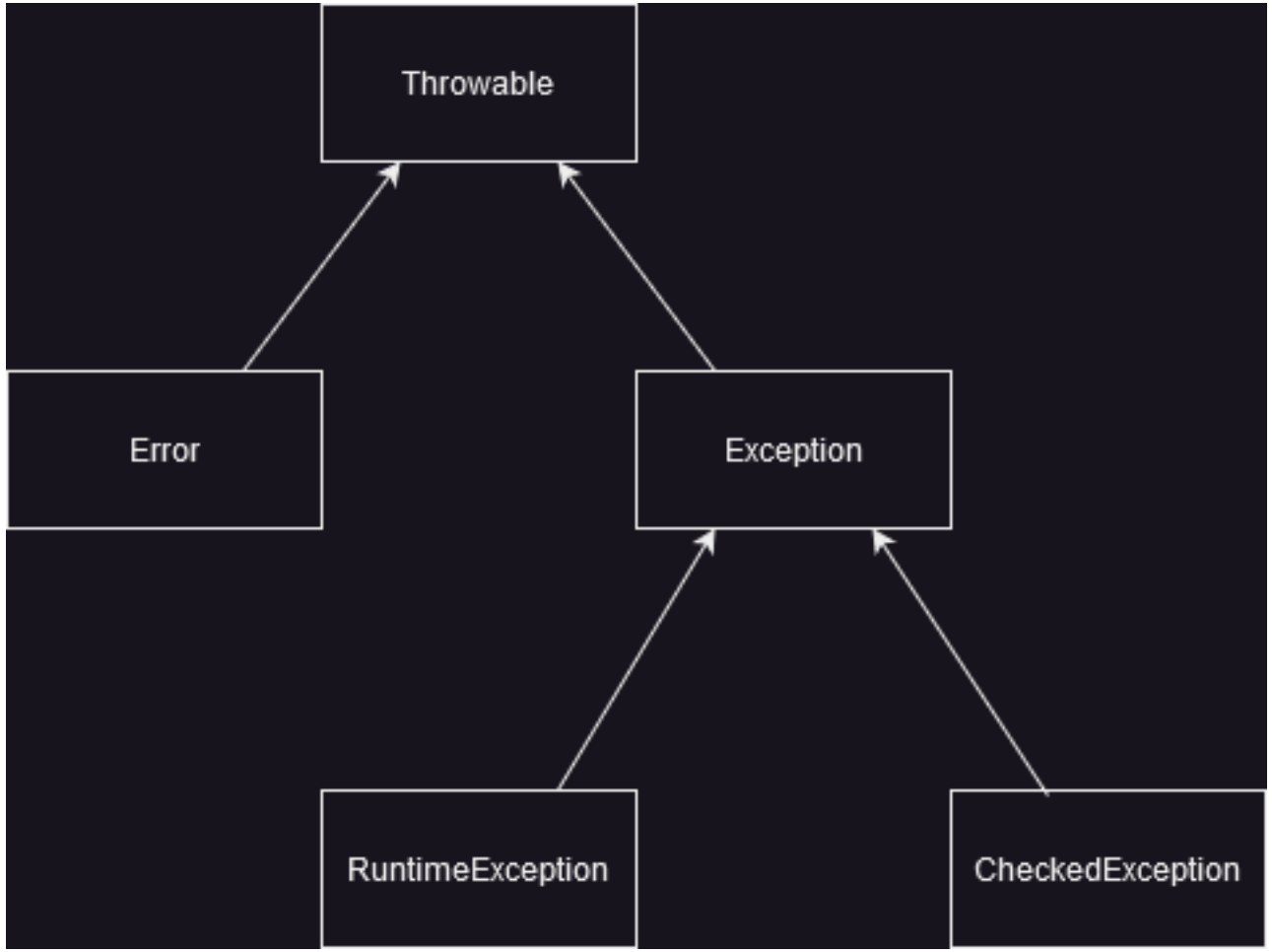

Inline Debugger
Variables Pane
for (; i < size; i++)
if (es[i] == null)
break found;
} else {
for (; i < size; i++) size: 2
if (o.equals(es[i])) o: Point@803 es: Object[10]@804 i: 0
break found;
}
return false;
Debug:
Coordinates
Debugger
Console
Frames
Threads
"main"@1 in group "main": RUNNING
remove:623, ArrayList (java.util)
removeValue:23, Coordinates (com.jetbrains)
main:11, Coordinates (com.jetbrains)
Variables
this = {ArrayList@802} size = 2
0 = {Point@806}
x = 12
y = 20
1 = {Point@807}
o = {Point@803}
es = {Object[10]@804}
size = 2
i = 0
es[i] = {Point@806}
Watches
Memory
No watches
All files are up-to-date (a minute ago)
623:1
LF
UTF-8
4 spaces

Errors & Exceptions

- Expected runtime exceptions (e.g. input validation, file not found, incorrect access rights...)
- Unexpected runtime exceptions (e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException)
- Unexpected unrecoverable errors (e.g. NoClassDefFoundError, OutOfMemoryError, IllegalAccessError)

Errors & Exceptions

- Error and Exception are subclasses of Throwable
- Indication that an unusual situation happened
- Errors not to be handled, thrown by JVM
- Exceptions handleable - Java has predefined Exceptions (SQLException, FileNotFoundException...)


Throwable
Error
Exception
RuntimeException
CheckedException

Errors & Exceptions

- You can make your own by extending the Exception class:

```
public class IncorrectFileNameException extends Exception {
  public IncorrectFileNameException(String errorMessage) {
    super(errorMessage);
  }
}
```

- Then throw it in a fitting situation:

```
try {

    // Attempt to do something where incorrect file name can be inputted.

} catch (FileNotFoundException e) {

    if (!isCorrectFileName(fileName)) {
        throw new IncorrectFileNameException("Incorrect filename : " + fileName);
    }

    // Continue handling the situation.

} finally {

    // Do this in any case.
}
```

```
try {

    // Attempt to do something where incorrect file name can be inputted.

} catch (FileNotFoundException e) {

    if (!isCorrectFileName(fileName)) {
        throw new IncorrectFileNameException("Incorrect filename : " + fileName, e);
    }

    // Continue handling the situation.

} finally {

    // Do this in any case.
}
```

Errors & Exceptions

- Handling exceptions should not serve to simply prevent the application from crashing.
- Errors and exceptions should thrown in appropriate situations.
- The message should be informative.
- The goal is to make debugging as easy as possible.

Logging

- A more sophisticated approach to tracking what is happening with the code and data.
- Can be outputted to the console, persisted in a file or re-directed to any other I/O stream.
- Again, the point is to make debugging easier.

Logging

- Events should be labeled with an appropriate level:
  - TRACE - the most granular information; used to "trace" the path through the code (line level)
  - DEBUG - variable value and other information of interest to the dev, not the admin (method level)
  - INFO - general statements about what the application is doing (feature level)

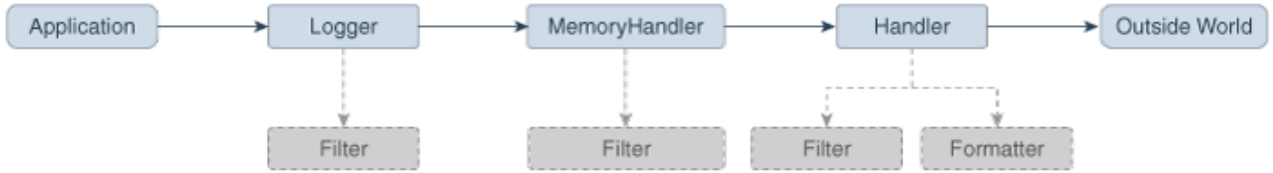
Logging

- WARN - statement about a potentially risky action (e.g. config file not found, using a backup server...)
- ERROR - something actually went wrong
- FATAL - something went so wrong the application needs to shut down
- Logs can be filtered based on that.
- Watch out for size of the log file!

Logging

- Java has a custom `java.util.logging` package.
- Allows for creation, filtering, memory handling and basic formatting of logs

source: [Oracle.com]



```java
public class Bank {

  // Obtain a logger object.
  private static Logger logger = Logger.getLogger("cz.muni.fi.bank");

  public static void main(String args[]) {

    // Log a TRACE level message
    logger.fine("Describing what I'm doing in detail.");

    try {
      user.withdraw(amount);
    } catch (Exception ex) {
      // Log the exception
      logger.log(Level.WARNING, "Cannot withdraw the inputted amount.", ex);
    }
    logger.fine("done");
  }
}
```

Logging

- Can be configured to an extent.
- Handlers can be set up for writing of logs to different streams.
- FileHandler, StreamHandler, ConsoleHandler, SocketHandler...

```
public static void main(String[] args) {

  Handler fileHandler = new FileHandler("%t/bank.log");
  Logger.getLogger("").addHandler(fileHandler);
  Logger.getLogger("cz.muni.fi").setLevel(Level.FINEST);
  ...
}
```

Logging

- The default Java logger is okay for basic use cases.
- Several logging frameworks generally used for finer control and extra features (e.g. automatic reloading of config files, graceful recovery from IO failures, better performance...)
- E.g.: Log4J 2, Logback, SLF4J...

Logging

- As external packages logging frameworks need to be included in your POM.xml.

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>${log-4j-api-version}</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>${log-4j-core-version}</version>
</dependency>
```

Logging

- Log4J is configured using the `log4j2.xml`.
- The end-user needs to define an appender, which contains information about the output stream, formatting...

```
<Appenders>
  <File name="logger" fileName="output-file.log" append="true">
    <PatternLayout>
      <Pattern>%d{yyyy-MM-dd HH:mm:ss} %p %m%n</Pattern>
    </PatternLayout>
  </File>
</Appenders>
```

- %d - date pattern, %p - log level, %m - message, %n - new line

Logging

- And then enable it by including it among Loggers:

```
<Loggers>
  <Root level="error">
    <AppenderRef ref="logger"/>
  </Root>
</Loggers>
```

Logging

- Usage is similar to the default Java logger:

```
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;

public class Something {

    private static Logger logger = LogManager.getLogger(Something.class);

    public static void main(String[] args) {
        logger.debug("Debug log message");
        logger.info("Info log message");
        logger.error("Error log message");
    }
}
```

Transactions

- Transactions are a way of preserving consistency in data.
- They ensure that when we are making a change in data, that we will either do it correctly or not at all.

Transactions

- ACID principles:
  - Atomicity - a transaction cannot be broken into smaller parts; either the whole action is completed or it is not done at all
  - Consistency - a transaction cannot being in or end in an invalid state
  - Isolation - a transaction is always done one at a time
  - Durability - once the transaction is finished, it stays finished forever (e.g. saved to disk)

Transactions

- The operation takes Runnable as an argument.
- Runnable object is an executable block of code, similar to Main.
- All calls in the block are executed or none of them are.

```
@Override
public void importData(String filePath) {
  transactionExecutor.executeInTransaction(() ->

        // delete data
        // validate import data
        // import all data
}
```