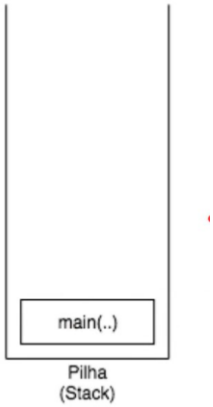
PILHA DE EXECUÇÃO

Para se lembrar do que ainda precisa ser executado e o que já foi executado. Começando com método MAIN

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

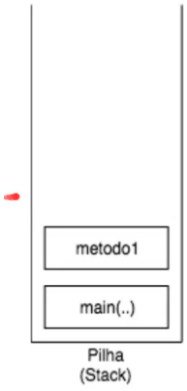


No console será exibido:

Início do main

Depois será chamado o método1, que o método1 foi chamado no main, ele interrompe sua execução e inicializa o metodo1:

private static void metodo1() {}



No console:

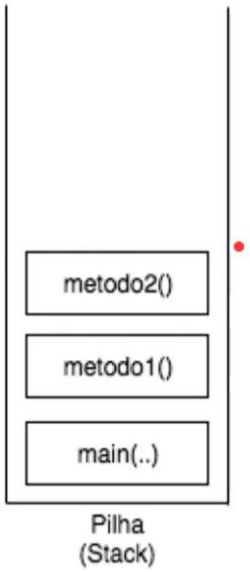
Início do main

Início do metodo1

(repare que o main não foi finalizado), só o metodo1 foi inicializado também.)

Foi feito uma chamada do metodo2, já no metodo1 para se finalizar ele precisar chamar o metodo2, ele foi chamado:

private static void metodo2() {}



No console:

Início do main

Início do metodo1

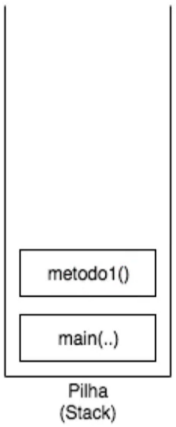
Início do metodo2

1 2 3 4 5

Fim do metodo2

Ele vai rodar todo o método2 pq ele tem uma finalização independente do main e do método1.

Ele **REMOVE** o metodo2 da ´pilha porque foi completamento executado:



Console:

Início do main

Início do metodo1

Início do metodo2

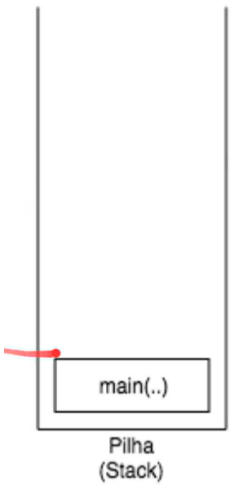
1 2 3 4 5

Fim do metodo2

Fim do método1

Ele vai terminar de executar o que sobrou do metodo1.

Remove o método1 da pilha e termina de executar o main:



No console:

Início do main

Início do metodo1

Início do metodo2

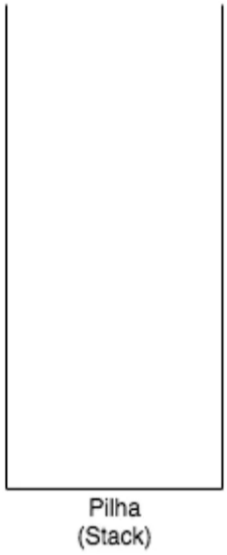
1 2 3 4 5

Fim do metodo2

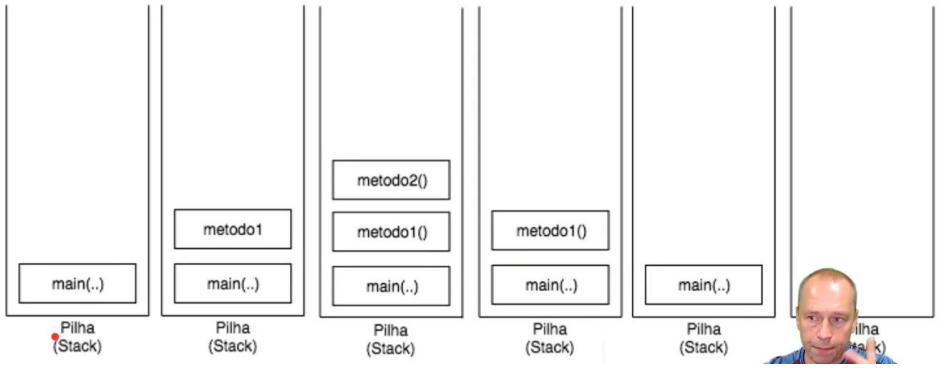
Fim do método1

Fim do main

Após terminar de executar todos os métodos o Java finaliza:



O Java será encerrado. Panorama geral da Pilha de Execução (é a mesma pilha):



Porque o Java (JVM) usa uma Stack(Pilha)?

1 - Para saber qual método está sendo executado.

2 - Para organizar a execução dos métodos.

Uma pilha Java faz parte da JVM e armazena os métodos que estão sendo executados. Além do bloco de código a pilha guarda as variáveis e as referências

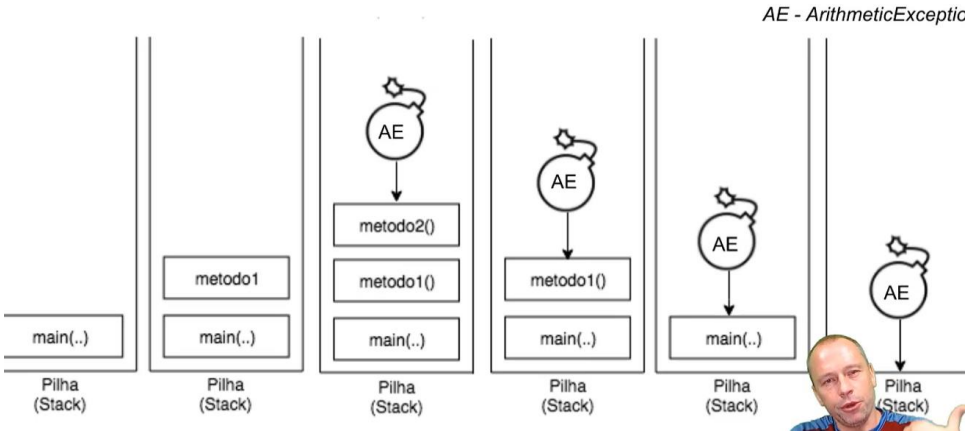
desse bloco. Assim a JVM organiza a execução e sabe exatamente qual método está sendo executado que é sempre o método no topo da pilha. A JVM também sabe quais outros precisam ser executados ainda, que são justamente os métodos abaixo.

DEPURAÇÃO

Debugar é olhar por dentro do sistema para visualizar seu processo de execução. Tem como selecionar linha por linha para ver a execução dos métodos de acordo com o método da pilha. Resume – Executa de acordo com a Stack planejou.

Usado para excepcionar linha por linha para entender o método de execução de um sistema.

TRATAMENTO DE EXECUÇÕES



Tem essa bomba o Java procura pra ver se os métodos sabem resolver algum problema, não tendo nenhum código, ele vai jogando os métodos fora e a bomba cai em cima do console, dando essa mensagem:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Fluxo.metodo2(Fluxo.java:32)

at Fluxo.metodo1(Fluxo.java:24)

at Fluxo.main(Fluxo.java:18)

O breakpoint ele não consegue dar continuidade ao fluxo.

Sobre exceções em Java:

1 - Exceções não tratadas caem na pilha de execução procurando por alguém que saiba lidar com ela.

2 - Toda exceção em Java possui um nome que a identifica. Essa abordagem torna seu entendimento mais fácil do que o uso de números mágicos (códigos de erros) como 15, 7012 ou 16.

TRY CATCH

try {} catch {} são comandos para capturar a exceções e permaneça o fluxo. Para retirar a bomba na pilha. Usada para quando código não é fixo e recebe algumas alterações, mas não mudar a base deles e evitar erros;

Try = algo cauteloso e definidor de bloco, uma exceção ao fluxo

Catch = Pega o nome dessa exceção dentro do fluxo e com uma variável.

```
try {
    int a = i / 0;
} catch(ArithmeticException ex) {
System.out.println("ArithmeticException");
}
```

No console

Inicio do main
Inicio do metodo1
Inicio do metodo2
1
ArithmeticException
2
ArithmeticException
3
ArithmeticException
4
ArithmeticException
5
ArithmeticException
Fim do metodo2
Fim do metodo1
Fim do main

No terminal ele trata essa exceção com try catch e continua o fluxo.

Sobre o Bloco TRY-CATCH

1 - Para tratarmos uma exceção, que pode ocorrer enquanto nosso programa esta sendo executado, precisamos tratá-la antecipadamente com um bloco de código específico.

2 - lógica de tratamento de erro no bloco catch só é disparada quando uma exceção é lançada dentro de um bloco try.

3 – Podem ter vários catches, mas todos devem ser especificados.

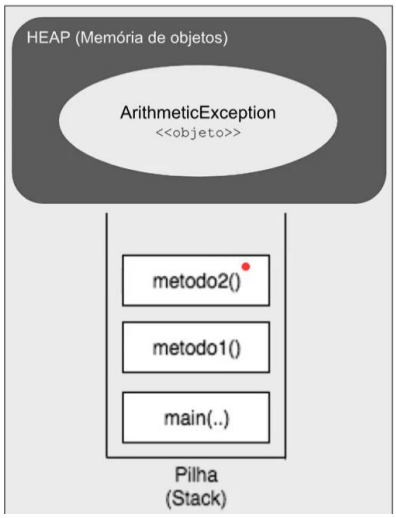
REVISANDO

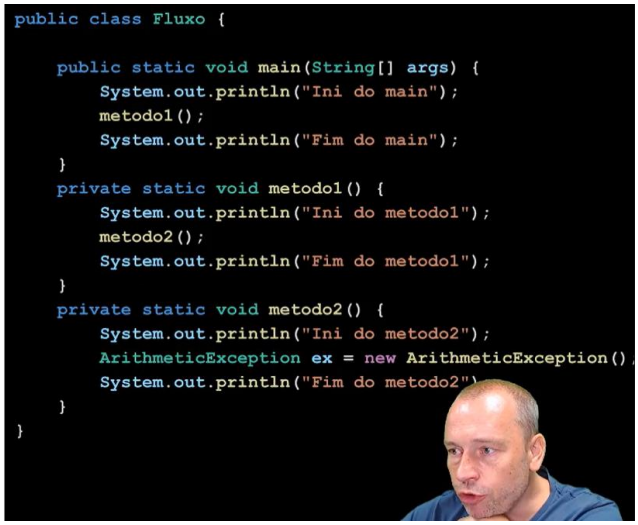
Erros excepcionais, imprevistos podem acontecer na produção de sistemas e por isso tem os **try catch** para permanecer com o fluxo do código.

1 - Para tratar a exceção, usa-se o bloco try e catch. Com o bloco try e catch, tratamos uma exceção que pode ocorrer enquanto nosso programa está sendo executado, tratando-a antecipadamente com um código específico.

2 - Caso não seja tratada, a exceção muda o fluxo de execução do programa, encerrando-o abruptamente.

LANÇANDO EXCEÇÕES





THROW

Throw – Comando que utilizado para sair abruptamente do fluxo da pilha. Funcionando apenas para exceções (só funciona com objetos).

Toda vez que instanciar (criar um objeto) é necessário lançar o throw:

Pode guardar em uma variável, fica mais verboso:

```
ArithmeticException exception = new ArithmeticException();
   throw exception;
}
```

Ou assim sem uma variável:

```
throw new ArithmeticException("Deu errado");
```

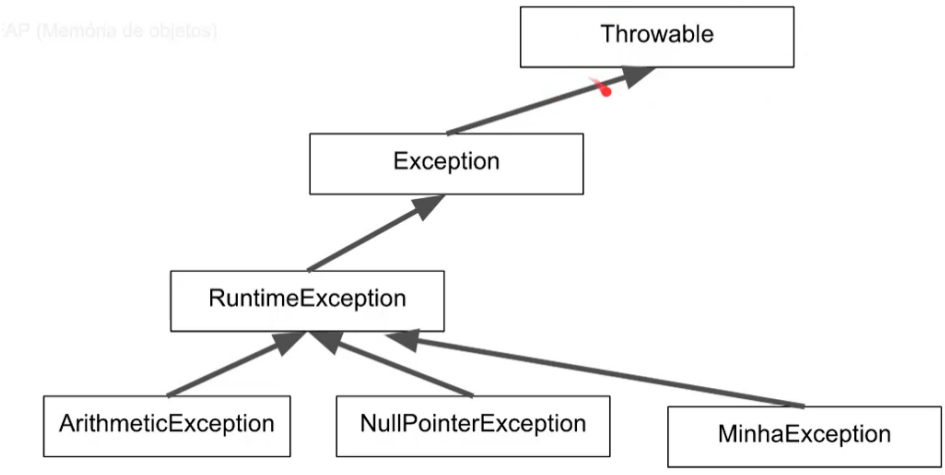
CHECKED E UNCHECKED

Criando as próprias exceções através do método runtimeexception. Exemplo:

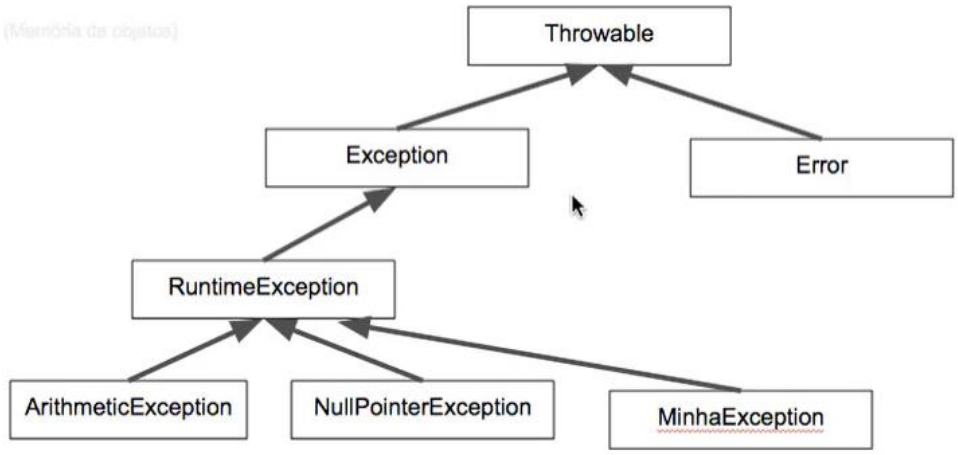
Para que isso ocorra a classe MinhaExcecao tem que herda de alguma classe na hierarquia de Throwable. Por exemplo RuntimeException:

```
public class EstoqueInsuficienteException extends RuntimeException {
}
```

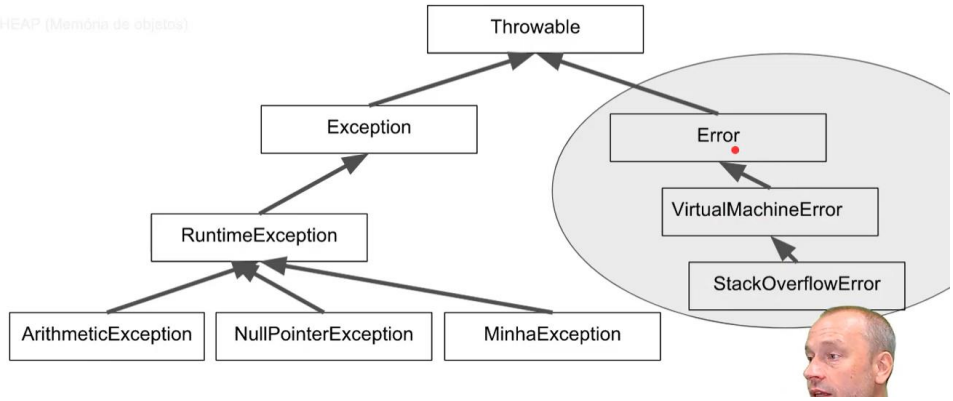
Hierarquia de exceções



Os Throwable são erros

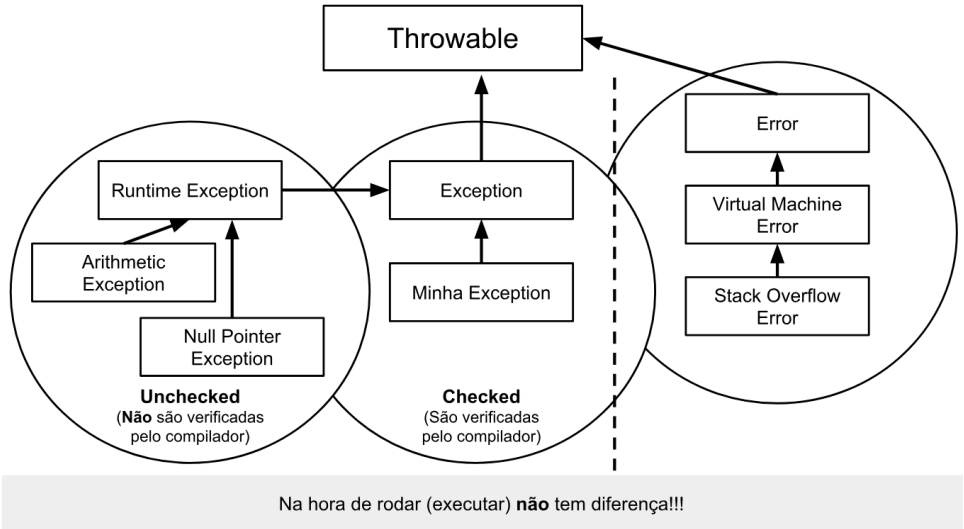


Os error são classes para os desenvolvedores da Máquina Virtual Java (MVJ), são lançando para os desenvolvedores da VMJ. Alguns erros do MVJ:



A classe Exception = para os desenvolvedores que usam a máquina virutal.

A classe Error = para os desenvolvedores da máquina virtual.



Unchecked - não verificados pelos compiladores;

Checked - São os verificados pelo compilador ao Extender a classe FluxoComError para Exception, ele exige uma identificação da exceção na assinatura do método:

Private static void metodo1( ) throws MinhaExececao {

System.out.println("Inicio do metodo2");

throw new MinhaExcecao(“Deu merda”);

}

O compilador dar duas opções: 1) transforma em uncheked ou identifica a exceção na assinatura. A diferença entre os dois são na hora da compilação, mas após os dois, na hora de rodar, são como uma bomba na pilha.

Sobre hierarquia de execeções e unchecked e checked:

1 - Existe uma hierarquia grande de classes que representam exceções. Por exemplo, **ArithmeticException** é filha de **RuntimeException**, que herda de **Exception**, que por sua vez é filha da classe mais ancestral das exceções, **Throwable**. Conhecer bem essa hierarquia significa saber utilizar exceções em sua aplicação.

2 - Throwable é a classe que precisa ser extendida para que seja possível jogar um objeto na pilha (através da palavra reservada throw).

3 - A hierarquia iniciada com a classe **Throwable** é dividida em exceções e erros. Contudo, classes que comunicam erros da máquina virtual herdam de **Error**.

4 - **StackOverflowError** é um erro da máquina virtual para informar que a pilha de execução não tem mais memória.

5 - Exceções são separadas em duas grandes categorias: aquelas que são obrigatoriamente verificadas pelo compilador e as que não são verificadas. As primeiras são denominadas checked e são criadas através do pertencimento a uma hieraquia que não passe por **RuntimeException**. As segundas são as **unchecked,** e são criadas como descendentes de **RuntimeException**.

RESUMO

- Existe uma hierarquia grande de classes que representam exceções. Por exemplo, **ArithmeticException é filha de RuntimeException**, **que herda de Exception**, **que por sua vez é filha da classe mais ancestral das exceções, Throwable.** Conhecer bem essa hierarquia significa saber utilizar exceções em sua aplicação.
- **Throwable é a classe que precisa ser extendida para que seja possível jogar um objeto na pilha (através da palavra reservada throw)**
- É na classe Throwable que temos praticamente todo o código relacionado às exceções, **inclusive getMessage() e printStackTrace().** Todo o resto da hierarquia apenas possui algumas sobrecargas de construtores para comunicar mensagens específicas
- A hierarquia iniciada com a classe **Throwable é dividida em exceções e erros**. Exceções são usadas em códigos de aplicação. **Erros são usados exclusivamente pela máquina virtual.**
- Classes que herdam de Error são usadas para comunicar erros na máquina virtual. Desenvolvedores de aplicação não devem criar erros que herdam de Error.
- StackOverflowError é um erro da máquina virtual para informar que a pilha de execução não tem mais memória.
- Exceções são s**eparadas em duas grandes categorias**: **aquelas que são obrigatoriamente verificadas pelo compilador e as que não são verificadas.**
- As primeiras são denominadas **checked e são criadas através do pertencimento a uma hierarquia que não passe por RuntimeException.**
- As segundas são as **unchecked, e são criadas como descendentes de RuntimeException.**

Criamos um catch genérico que captura qualquer exceção, incluindo exceções checked.

Isso pode parecer uma boa prática, mas normalmente não é. **Como regra geral, sempre tente ser mais específico possível no bloco catch favorecendo vários blocos catch** ou usando **multi-catch.**

```
try                                                                    {
   metodoPerigosoQuePodeLancarVariasExcecoes();
```

```
} catch(Exception ex) {
    ex.printStackTrace();
}
```

De que maneira as exceptions podem ajudar a melhorar o código de seu programa?

a - Exceções tem um nome e, caso esse nome seja expressivo, documenta o problema que está ocorrendo.

b - Exceções podem ter uma mensagem, ou seja, o problema e o estado das variáveis podem ser descritos na mensagem.

c - Exceções mudam o fluxo de execução, ou seja, evitam que o problema siga o fluxo "normal" quando algo excepcional acontece.

d - Exceções podem ser tratadas, ou seja, podemos voltar para a execução "normal" caso o "problema" esteja resolvido.

Bloco Finally

Um comando opcional para fechar uma exceção como de métodos da classe de conexão com o banco de dados que precisa de um método de fim da conexão. Garante que a exceção seja encerrada e pode ser feita sem capturar um erro como o catch (tratar a exceção). **Só pode ter 1 finally.**

Afirmações obre o bloco finally:

O bloco finally é opcional quando há bloco catch.

O bloco finally sempre será executado (sem ou com exceção).

O bloco finally é tipicamente utilizado para fechar um recurso como conexão ou transação.

Modelos de como o finally pode ser usado:

1) **temos um try com dois blocos catch (clássicos) e o bloco finally.**

```
try {}
catch(SacaException ex) {}
catch(DepositaException ex) {}
finally {}
```

2) um tratamento com try e multi-catch.

```
try {}
catch(SacaException | DepositaException  ex) {}
```

3) no tratamento o bloco catch é opcional quando tem o bloco finally (Mas é sempre bom usar o catch para capturar a exceção)

```
try {}
finally {}
```

Try-with-resources

Tratando exceções nos objetos:

```
try(Conexao con = new Conexao()) {
    con.leDados();
}
```

1 - O bloco finally é criado automaticamente. Automaticamente é criado um bloco finally. Nele é chamado o método close() do recurso.
2 - O recurso precisa implementar o método close().

Exceções padrões

IllegalStateException - que faz parte da biblioteca do Java e **indica que um objeto possui um estado inválido.** Você já deve ter visto outras exceções famosas, como a NullPointerException. Ambos fazem parte das exceções padrões do mundo Java que o desenvolvedor precisa conhecer.

IllegalArgumentException – Pode usar ele quando, por exemplo, uma agência tem valor negativo (não faz sentido nenhum). Então o IllegalArgumentException é uma exceção para indicar que algum argumento é ilógico, inválido.