Programação Orientada a Objetos

Aula 11 - Tratamento de Exceções

Apresentação

Até aqui, aprendemos muitos conceitos e elementos da linguagem Java que permitem ao programador escrever programas OO de uma maneira bem simples. Nesta aula, iremos aprender como tratar situações anormais de execução dos programas. Em particular iremos aprender como capturar erros que podem acontecer em situações não comuns e fazer o tratamento adequado dos mesmos, evitando assim paradas abruptas do programa ou exibição de detalhes dos erros do programa para o usuário. Boa aula!!!

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Entender a hierarquia básica das exceções.
- Saber como os métodos avisam ao compilador e ao programador que o programa pode causar riscos.
- Saber como capturar e tratar as exceções quando elas acontecerem.
- reconhecer os tipos mais comuns de exceções.

Comportamento do Risco

Coisas podem acontecer, o arquivo pode não está lá. O servidor pode cair. Não importa o quão bom você é programador, você não pode controlar tudo. Quando você escreve um método que pode causar risco, é necessário código para capturar as coisas ruins que podem acontecer. Mas, como saberemos quando um método poderá causar riscos? E o mais importante, onde colocaremos o código para capturar as situações de exceções?

Um Pouco de Conversa!

Uma regra antiga no desenvolvimento de software diz que 80% do tempo gasto é referente ao esforço requerido para checar e capturar erros, o restante é relativo ao desenvolvimento. A **detectação** e a **captura** de erros podem ser o ingrediente mais importante para construção de uma aplicação robusta.

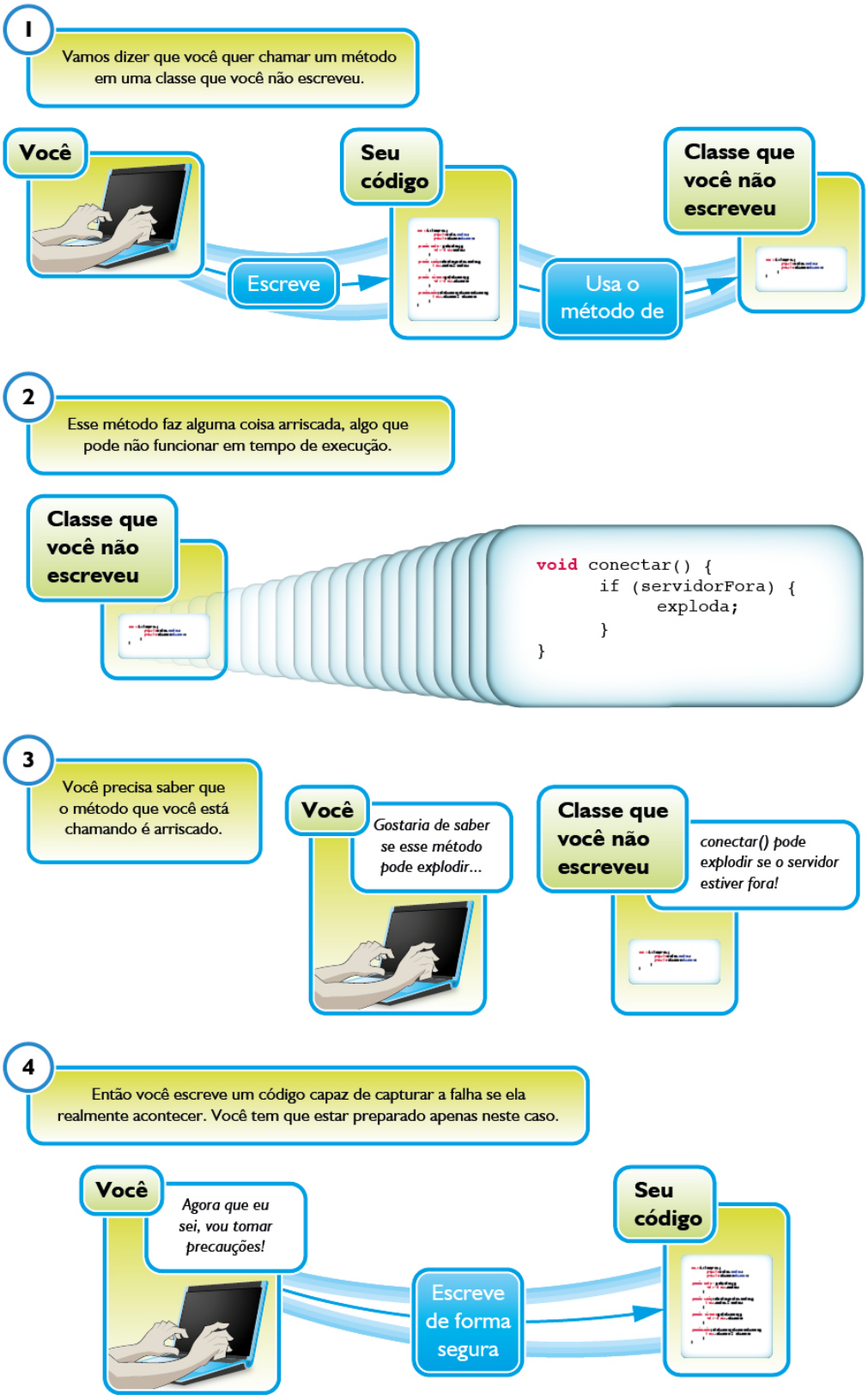
Felizmente, JAVA nos mune de excelentes equipamentos para capturar e organizar códigos para tratamento dos erros, mecanismo conhecido como *Exception handling*, que nos permite detectar erros facilmente sem ter que escrever códigos especiais para verificar os tipos de retornos. E ainda melhor, nos permite criar códigos que tratam as exceções *separadas dos códigos que as geram*. E a cereja do bolo ainda estar por vir... ele também nos permite capturar vários tipos de exceções e tratá-los em um único ponto.

Encarando o Problema de Frente!

Vamos dar uma olhada na seguinte situação e entender melhor por que capturar exceções é tão importante.

O que acontece quando um método que você deseja chamar (provavelmente de uma classe que você não escreveu) pode causar riscos?

Figura 01


1
Vamos dizer que você quer chamar um método em uma classe que você não escreveu.
Você
Escreve
Seu código
Usa o método de
Classe que você não escreveu
2
Esse método faz alguma coisa arriscada, algo que pode não funcionar em tempo de execução.
Classe que você não escreveu
void conectar() {
if (servidorFora) {
exploda;
}
}
3
Você precisa saber que o método que você está chamando é arriscado.
Você
Gostaria de saber se esse método pode explodir...
Classe que você não escreveu
conectar() pode explodir se o servidor estiver fora!
4
Então você escreve um código capaz de capturar a falha se ela realmente acontecer. Você tem que estar preparado apenas neste caso.
Você
Agora que eu sei, vou tomar precauções!
Escreve de forma segura
Seu código

Agora sim, entendemos que coisas ruins podem acontecer caso não nos preocupemos em tratá-las. Como vimos na ilustração acima, não podemos controlar, no caso, o programador estava chamando um código que não foi escrito por ele mesmo, e que poderia provocar grandes problemas se não tratado da forma correta, mas que para isso temos que saber previamente que um determinado código pode provocar tal exceção. É sobre isso e muito mais que iremos falar de agora em diante, mas antes vamos dar uma olhada na **hierarquia de exceções**.

Hierarquia de Exceções

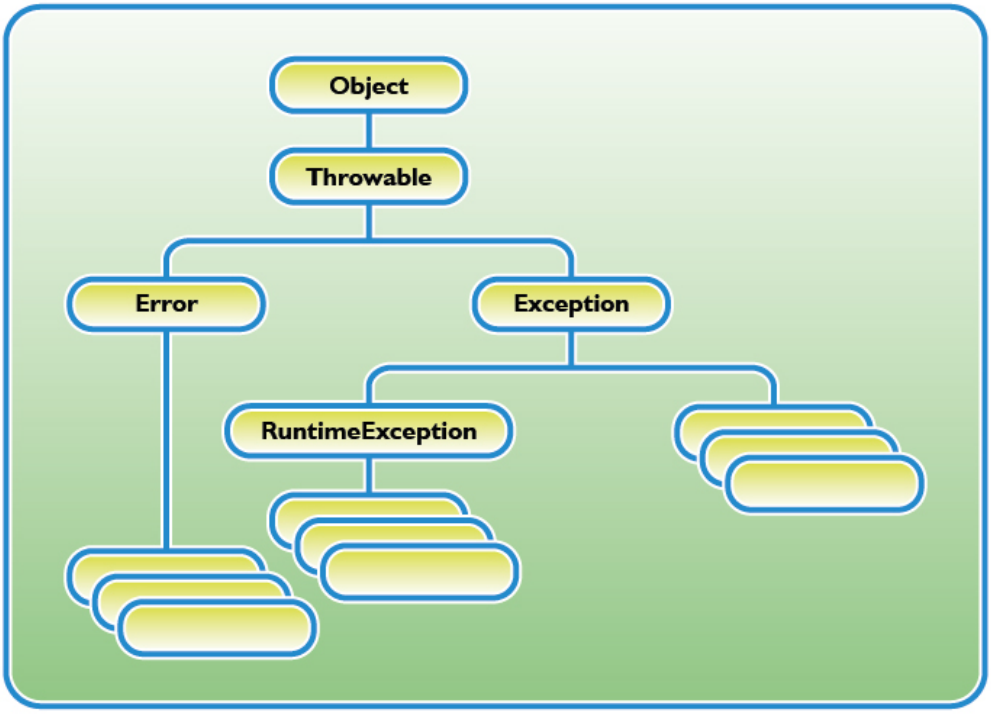
Primeiramente, o que é uma exceção em Java? De onde ela vem? Opa... Uma pergunta de cada vez!

Inicialmente, vamos dar uma olhada na terminologia **"exception"**, que significa **"condição excepcional"** a qual é uma ocorrência que altera o fluxo normal de execução do programa. Um monte de coisas pode causar exceções, incluindo falha de hardware, exaustão de recursos, ou até mesmo velhos **Bugs**.

Agora, respondendo a primeira pergunta: um **"exception"** em Java é um Objeto do tipo **Exception**. Sendo assim, como a exceção é um objeto, o que você receberá quando ocorrer uma condição de falha será um Objeto. Veremos mais adiante como é a sintaxe para capturar uma exceção em Java!

Não fiquem ansiosos... Vamos agora responder a segunda pergunta (de onde ela vem?), analisando a sua árvore "genealógica", temos o seguinte figura:

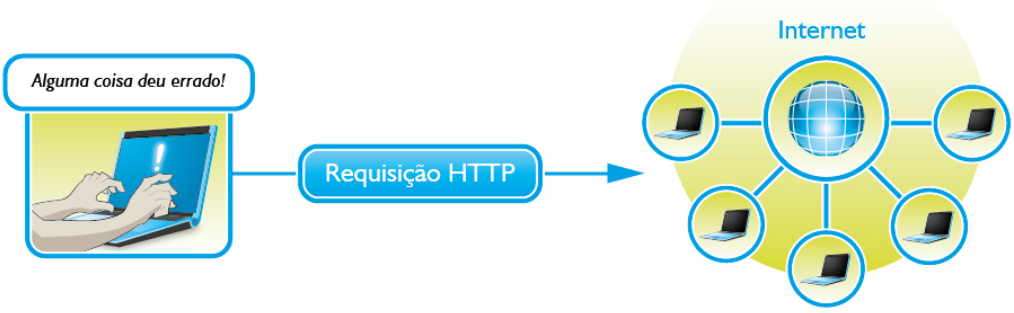
Figura 02 - Hierarquia das classes de exceções



Como você pode ver, existem duas classes que derivam diretamente de **Throwable: Exception** e **Error**. Classes que derivam de Error representam situações não usuais as quais não são causadas por erro de programa, e indicam coisas que não acontecem normalmente durante a execução do programa, tal como o famoso **Out of memory** da JVM. Geralmente, a nossa aplicação não será capaz de se recuperar de um erro proveniente de Error, sendo assim, não vamos esquentar a cabeça com isso!

Vamos então ao que interessa. Em geral, uma exceção representa algo que acontece NÃO como resultado de um erro de programação, mas sim devido a algum recurso que não está disponível ou quando alguma outra condição requerida para a execução correta do programa não está presente no momento da execução. Por exemplo, se a sua aplicação está querendo se comunicar com outra aplicação ou computador que não está respondendo, temos então um caso em que a exceção não foi causada por um Bug, nesse caso, será lançada uma **RuntimeException**, como visto na figura **Hierarquia das Classes de Exceções**.

Figura 03



Muito bom, aprendi o que é uma Exception em Java, sei a diferença entre um Erro e uma Exceção, sei que ela é um Objeto normal e até que herda de Throwable! Ótimo, tudo! Mas, não sei nada da sintaxe, como fico sabendo que um método pode causar uma exceção? Como eu trato uma exceção? Calma, calma... Agora, é que vai começar a parte boa da história, vamos falar sobre a SINTAXE das exceções! Então, vamos lá...

Exceptions (Sintaxe)

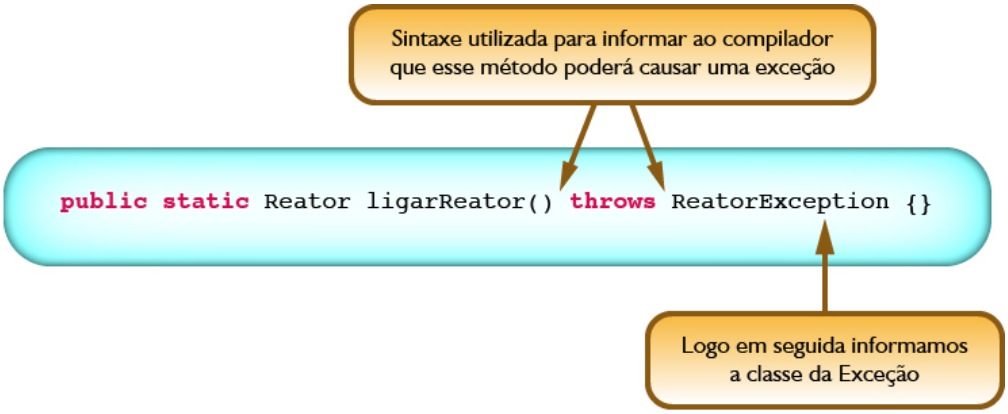
Primeiramente, um pouco de marketing sobre o mecanismo de tratamento de exceções em Java. Devo falar que ele é totalmente “Clean”, ou seja, é feito para que seja totalmente transparente e intuitivo para o programador capturar e tratar as condições adversas.

É baseado de maneira que você saiba quando está chamando um método que possa levantar uma exceção, assim você pode escrever códigos para lidar com essas possibilidades. Se você sabe que um método poderá causar algum problema antecipadamente, podemos então nos precaver e estar preparado para o pior! E se o pior vir a acontecer, podemos contornar a situação, já que era sabido que algo de ruim poderia acontecer!

Vídeo 02 - Exceções

Vamos começar pelo mais básico, ou seja, como finalmente sabemos que um método o qual desejamos chamar pode causar algum tipo de problema? Em Java, os métodos utilizam a cláusula **throws** na declaração do método!

Figura 04



O método **ligarReator()** acima pode causar uma exceção em tempo de execução, sendo assim é nossa obrigação informar que tipo de exceção pode ocorrer para que os usuários dessa função possam então se precaver.

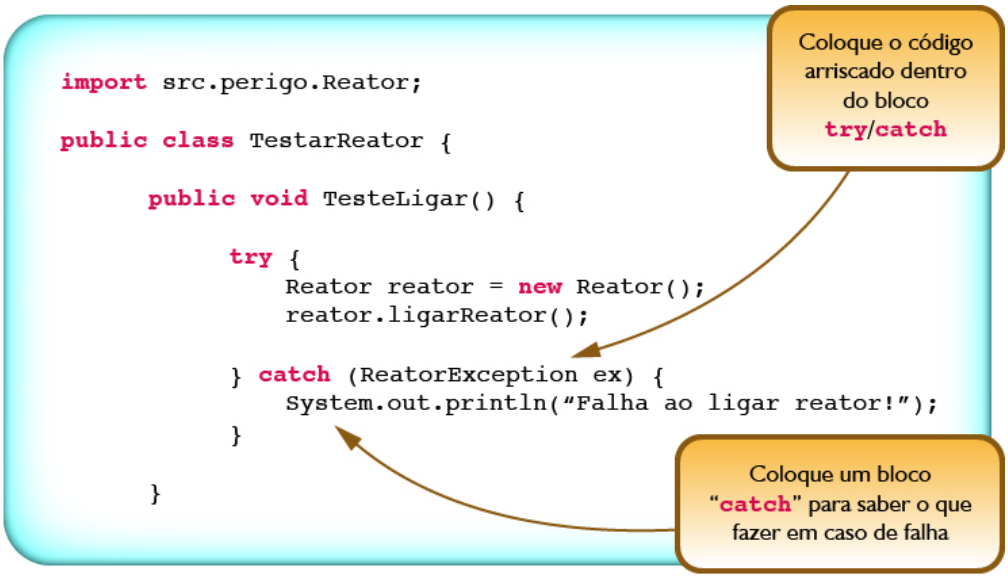
Para os mais apressadinhos e curiosos, notaram algo de especial no método acima? De onde vem esse tipo de exceção **ReatorException?** Parece-me algo criado pelo programador especialmente para tratar de um tipo específico de risco! E você está certo, o mecanismo para tratamento de exceções do Java é bastante completo e nos permite criar nossas próprias exceções, mas vamos devagar com a missa! Esse assunto será tratado mais tarde, foi apenas uma pequena pausa para deixá-los com água na boca!

Muito bom mesmo! Agora sei que se eu chamar um método e ele tiver uma cláusula denominada **throws** na sua declaração, significa que eu tenho que me precaver caso o pior aconteça! Massa! Mas, como eu faço isso?

Figura 05

Calma! Para isso que existe o **try/catch**! Vamos aprender agora como tratar a chamada a um método que possa causar uma exceção! Vejamos o código a seguir!

Figura 06



E para que tudo isso? O **compilador** tem que **saber** que **você** está chamando um método de risco!

Ou seja, o compilador não irá descansar enquanto você não colocar o seu código arriscado em um bloco **try/catch**. Essa é a forma de dizer ao compilador que você sabe que alguma coisa ruim poderá acontecer no método que você está chamando e que você está preparando para o que der e vier!

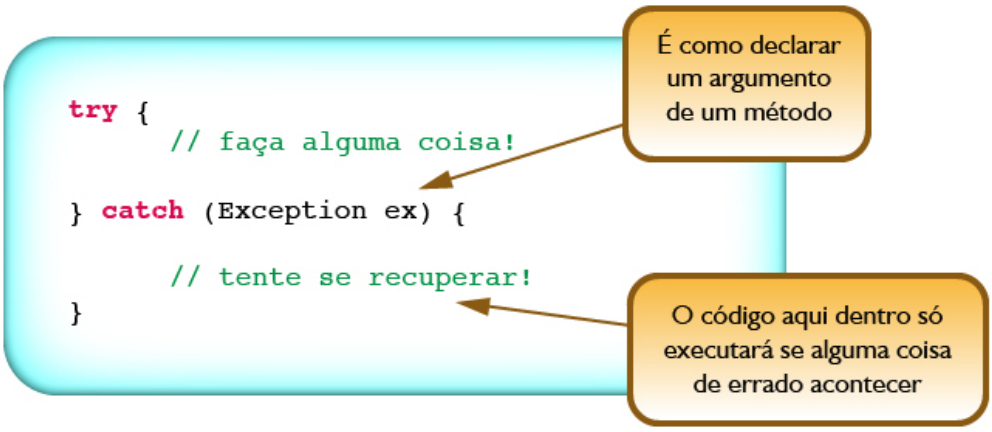
Lembre-se: o compilador não quer saber como você irá tratar o problema, ele se importa apenas em saber se você está tomando as devidas precauções.

Anote a Dica

Uma **exceção** é um objeto... do tipo **Exception**

Lembra da nossa árvore "genealógica", mostrada anteriormente? Pois então, toda exceção que criarmos é filha de **Exception**, que por sua vez herda de **Throwable**, sendo assim, como uma exceção é um objeto, tudo que capturamos na cláusula **'catch'** é também um objeto, representado pelo parâmetro **'ex'**, conforme o exemplo a seguir.

Figura 07



Agora que aprendemos um pouco mais sobre o mundo das exceções em Java, vamos por em prática os nossos conhecimentos.

Vídeo 03 - Hierarquia

Atividade 01

1. Escreva uma classe que tenha um método que lance uma exceção utilizando a cláusula **throws**. Use a sua imaginação e utilize uma exceção

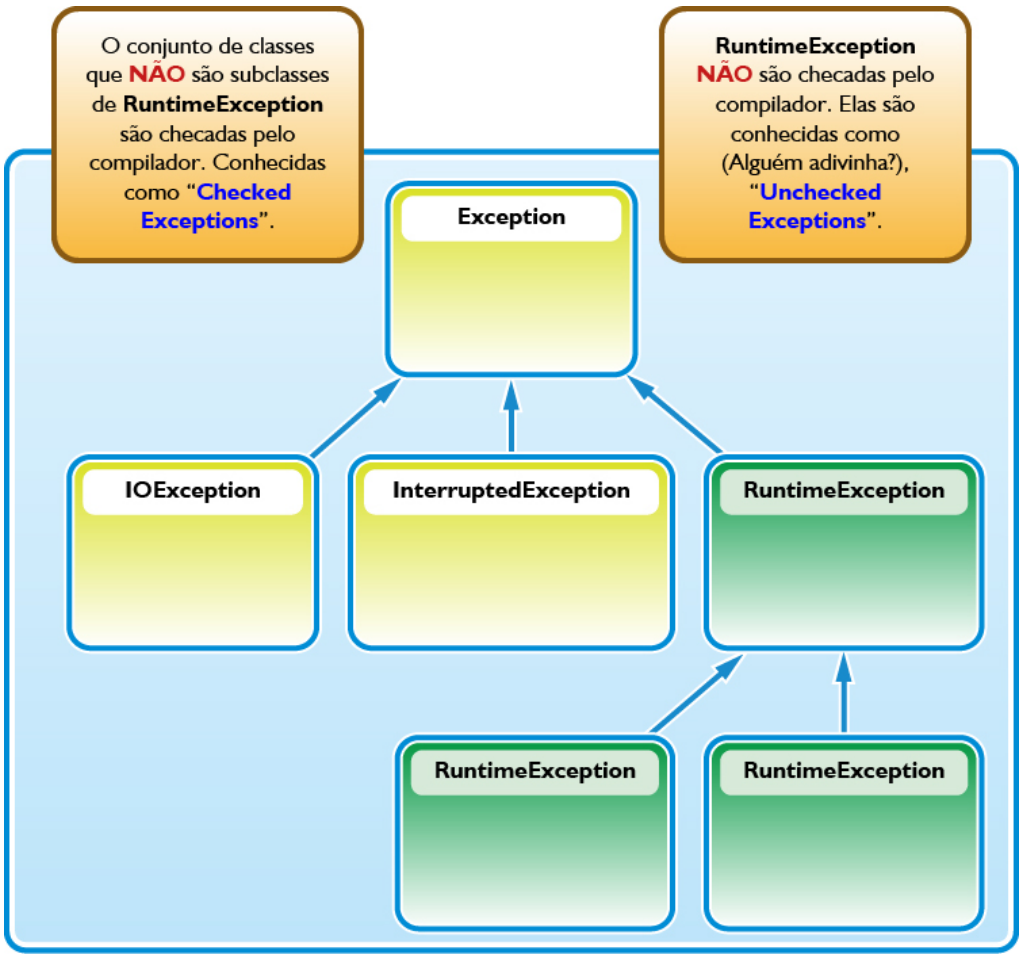
comum da API que não seja filha de **RuntimeException**.

2. Crie um programa Java que utilize o método criar acima e trate a exceção dentro de um *bloco try/catch*.

Tipos Comuns de Exceções

Eu sei que já vimos a hierarquia de exceções no início da aula, mas vamos dar uma olhada mais detalhada nessa hierarquia para entender melhor os tipos mais comuns de exceções.

Figura 08



RuntimeException? Como assim, eu sou diferente das demais? Mas eu não sou, ao final de contas, filha de Exception também!

Visão do Compilador

Ele se preocupa com todas as exceções que sejam subclasses de Exception, a menos que elas sejam de um tipo especial (**RuntimeException**). Toda classe de exceção que herde de *RuntimeException* tem passe livre, ou seja, para o compilador tanto faz se você declará-la na cláusula **throws** ou não, também não faz diferença se você a trata dentro de um bloco **try/catch**, para ele é como se as *RuntimeException* sequer existissem.

Figura 09

Espera aí! Mas por que tanto desprezo com as *RuntimeException*, o que elas têm de diferente das outras, afinal de contas, somos todas da mesma família! Então, vamos às explicações.

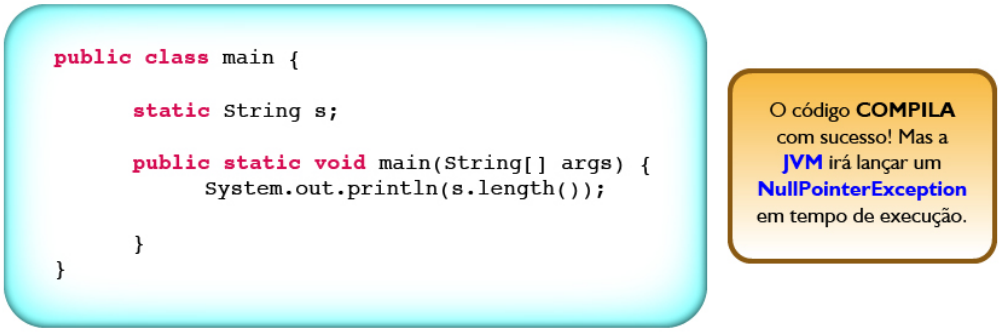
A maioria das *RuntimeException* são derivadas de problemas no seu código, ao invés de um problema que acontece em tempo de execução que você não poderia prever. Você não pode garantir que ao abrir um arquivo ele esteja lá, ou que o servidor está funcionando, mas você pode garantir que o seu código não faça acesso a um índice inexistente em um Array (é para isso que serve o atributo *'length'*).

Você desejará que as RuntimeException aconteçam durante o desenvolvimento. Você não desejaria ter o trabalho de tratar tal tipo de erro dentro de um bloco **try/catch**, por exemplo, e ter todo overhead de lidar com isso, para capturar alguma coisa na cláusula catch, que na realidade não deveria ter acontecido antes de tudo!

O bloco **try/catch** é para capturar condições excepcionais e não fluxo de código. Use a cláusula *catch* para tentar se recuperar de situações que você não pode garantir que irá acontecer com sucesso, ou, em último caso, exibir uma mensagem para o usuário e imprimir a pilha de erro para que alguém possa descobrir o que está errado!

Em exemplo clássico de uma *RuntimeException* é o famoso **NullPointerException**. Alguém já viu uma dessa antes? Aposto que sim! Esse tipo de exceção, como todos nos sabemos, acontece quando tentamos acessar uma referência de um objeto **nulo**!

Figura 10



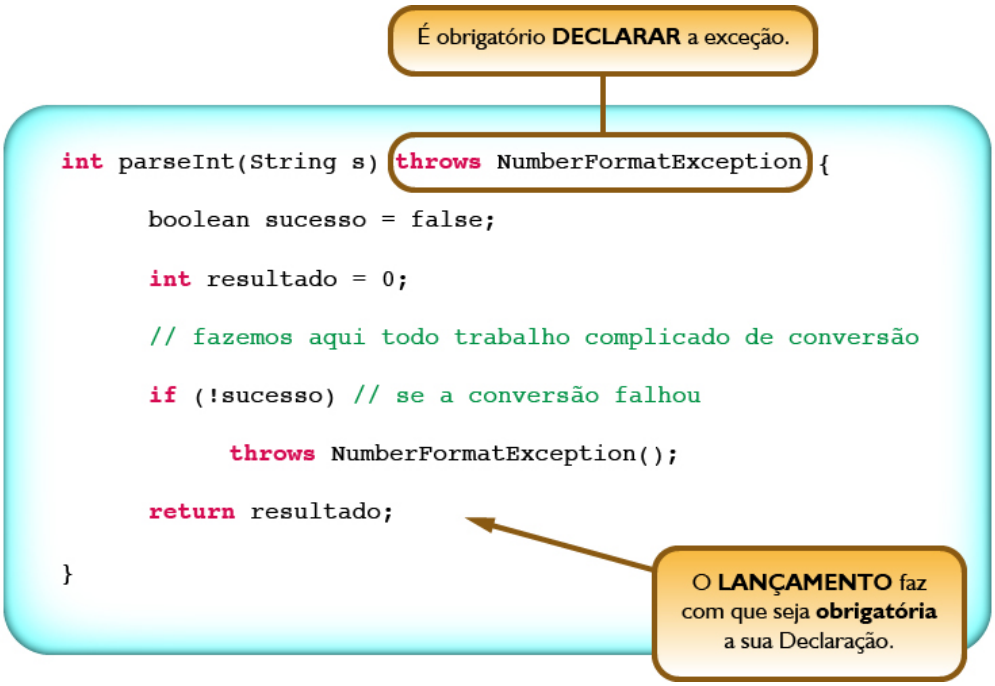
Exceções Programáticas

Agora chegou a hora de conhecermos um pouco das exceções que o compilador pode detectar, as quais nos forçam a usar todas as maravilhas disponíveis na API Java para tratamento de exceções. Vamos dar uma olhada no seguinte caso.

Algumas classes da API Java têm métodos que recebem uma String como argumento, e converte essas Strings em tipos primitivos numéricos. Um bom exemplo dessas classes são as conhecidas **"Wrapper Classes"**.

Vamos tomar como exemplo de classe *Wrapper* a famosa **java.lang.Integer**. Vamos agora imaginar como seria a implementação do método parseInt(), que tem como objetivo receber uma String e retornar um número inteiro. Sendo assim, o programador sabiamente decide que se nesse método for passado uma String que não possa ser convertida em um tipo numérico, então, o método deve lançar uma exceção do tipo **NumberFormatException**. Vejamos o exemplo a seguir.

Figura 11



Vídeo 04 - Exemplo Prático

Atividade 02

1. Escreva uma classe que tenha um método que lance uma exceção utilizando a cláusula **throws**. Dessa vez, devemos utilizar uma exceção da API que seja filha de **RuntimeException**. Note o comportamento do compilador e diga o que foi notado.

2. Criar um programa Java que utilize o método criar acima e trate a exceção dentro de um bloco *try/catch*. Faça o mesmo com esse exercício, note o comportamento do compilador.

Conclusão

Existe um monte de exceções, passaríamos o dia falando delas e não acabariam. A tabela a seguir traz um resumo das exceções mais comuns.

Exception(Mais comuns)	Descrição	Tipicamente Lançadas
ArrayIndexOutOfBoundsException	Lançada quando tentamos acessar um Array com um valor de índice inválido (Negativo ou maior do que o tamanho do Array).	Lançada pela JVM
ClassCastException	Lançada quando tentamos fazer um 'cast' para uma referência e ocorre uma falha	Lançada pela JVM
IllegalArgumentException	Lançada quando um método recebe um argumento diferente do tipo esperado.	Programaticamente
NullPointerException	Lançada quando tentamos acessar uma referência de um objeto null.	Lançada pela JVM

Exception(Mais comuns)	Descrição	Tipicamente Lançadas
NumberFormatExceptoin	Lançada quando um método que converte uma String em um número recebe uma String que não é convertida.	Programaticamente

Resumo

Bom, chegamos ao final de mais uma aula, agora estamos mais familiarizado com a API Java para tratamento de exceções. Vimos toda sua Hierarquia, sabemos agora que Erros em Java são diferentes de Exceções, e que essas últimas podem ser vista em dois grupos (Lançadas pela JVM), as quais são todas aquelas do tipo *RuntimeException* e suas descendentes, e Progamaticamente, que são todas as demais, as quais o compilador faz questão que a tratemos, pois, diferentemente das *RuntimeException*, podemos evitar a sua ocorrência e até nos recuperarmos da forma adequada a cada situação. Vimos também a sintaxe básica para lançamento de exceções *(throws)*, assim como o bloco *try/catch*, o qual utilizamos para capturar e tratá-las. Por fim, fizemos um resumo dos tipos mais comuns de exceções e vimos alguns casos em que elas podem ocorrer.

Autoavaliação

1. Crie uma classe contendo um método que lance uma exceção do tipo NumberFormatExceptoin.
2. Crie um programa Java que utilize o método criar acima e trate a exceção dentro de um bloco try/catch.
3. Crie um programa Java que manipule os valores de um Array, porém, devemos fazer com que ele acesse um índice inválido, de forma que o programa irá compilar corretamente, porém, será lançada uma exceção em tempo de exceção do tipo ArrayIndexOutOfBoundsException.
4. Adapte o programa criado na questão 2 desta autoavaliação para imprimir a pilha de erros utilizando o printStackTrace().

Referências

KATHY SIERRA, Bert Bates. **Head First Java**. 2nd edition. 2005.

_____. **SCJP Sun Certified Programmer for Java 6 Study Guide**. 2008.