

# Java Starter

[www.t2ti.com](http://www.t2ti.com)

# Java Starter

[www.t2ti.com](http://www.t2ti.com)

# Curso Java Starter

---

## **Apresentação**

O Curso Java Starter foi projetado com o objetivo de ajudar àquelas pessoas que têm uma base de lógica de programação e desejam entrar no mercado de trabalho sabendo Java,

A estrutura do curso é formada por módulos em PDF e por mini-cursos em vídeo. O aluno deve baixar esse material e estudá-lo. Deve realizar os exercícios propostos. Todas as dúvidas devem ser enviadas para a lista de discussão que está disponível para inscrição na página do Curso Java Starter no site [www.t2ti.com](http://www.t2ti.com). As dúvidas serão respondidas pelos instrutores Albert Eije, Cláudio de Barros e Miguel Kojiio, além dos demais participantes da lista.

Nosso objetivo é que após o estudo do Curso Java Starter o aluno não tenha dificuldades para acompanhar um curso avançado onde poderá aprender a desenvolver aplicativos para Web, utilizando tecnologias como Servlets e JSP e frameworks como Struts e JSF, além do desenvolvimento para dispositivos móveis.

Albert Eije trabalha com informática desde 1993. Durante esse período já trabalhou com várias linguagens de programação: Clipper, PHP, Delphi, C, Java, etc. Atualmente mantém o site [www.alberteije.com](http://www.alberteije.com).

Cláudio de Barros é Tecnólogo em Processamento de Dados.

Miguel Kojiio é bacharel em Sistemas de Informação, profissional certificado Java (SCJP 1.5).

O curso Java Starter surgiu da idéia dos três amigos que trabalham juntos em uma instituição financeira de grande porte.

# Classes Abstratas, Interfaces e Exceções

## Introdução

Estamos alcançando o fim da nossa trilha – que é apenas o início do maravilhoso mundo que Java nos oferece – e este momento é bastante adequado para introduzirmos os conceitos de classes Abstratas, Interfaces e Exceções. Classes Abstratas e Interfaces são estruturas muito importantes quando programamos Orientado a Objetos com Java e, visto que já conhecemos os demais fundamentos da linguagem (reuso, herança, polimorfismo, encapsulamento, métodos, etc.), está na hora de aplicarmos estes recursos existentes na linguagem.

Uma das diferenças mais evidentes entre um bom e um mal programa é o tratamento de exceções, isto é, a capacidade que o seu programa tem de responder a situações inesperadas. Quando o tratamento de exceções é feito adequadamente o seu programa fica mais robusto e ao mesmo tempo a usabilidade – usabilidade nada mais é do que a facilidade que um usuário tem em utilizar o programa – aumenta sensivelmente.

Porém antes de iniciarmos o novo conteúdo vamos conhecer uma importante característica do construtor com parâmetros levando em conta o uso de herança.

## Herança - continuação

Nós já vimos que uma classe herda de sua classe pai as suas características porém a subclasse não herda o **construtor**, isto é, o construtor é individual para cada classe.

Sempre que o construtor de uma classe é invocado ele invoca , quando não explicitado, implicitamente o construtor da superclasse. Desta forma o construtor da

## Curso Java Starter

---

classe Object sempre é invocado, pois todas as classes estendem dela. Observe o código abaixo referente a classe Veiculo:

```
public class Veiculo {  
  
    private String marca;  
    private Double capacidadeTanqueCombustivel;  
  
    public Veiculo(String marca) {  
        this.marca = marca;  
    }  
  
    //Demais membros...
```

Quando o compilador gerar o bytecode o que nós teremos é o seguinte:

```
public class Veiculo {  
  
    private String marca;  
    private Double capacidadeTanqueCombustivel;  
  
    public Veiculo(String marca) {  
        super();  
        this.marca = marca;  
    }  
  
    //Demais membros...
```

Observe que a declaração `super()` indica que o construtor default – construtor public e sem parâmetros – da superclasse está sendo invocado. Isto acontece de forma automática, nós não precisamos nos preocupar.

O construtor default é adicionado pelo compilador quando não há construtor declarado explicitamente, isto é, se o programador não declarar um construtor o compilador irá adicionar, implicitamente, um público e sem parâmetros.

No entanto se for adicionado explicitamente um construtor o compilador não irá adicionar implicitamente o construtor default, o que significa que se nós criarmos um construtor com parâmetros e estendermos a classe, obrigatoriamente teremos que, explicitamente, invocar o construtor da superclasse passando os parâmetros.

Para ficar mais claro vamos estender a classe Veiculo que possui um construtor com parâmetros:

```
public class Jipe extends Veiculo{  
  
    public Jipe() {  
    }  
  
    //Demais membros...
```



O código acima demonstrado não compila. Imagine o compilador tentando

## Curso Java Starter

adicionar a declaração `super()` porém sem sucesso. Isto acontece pois não existe construtor sem parâmetros na superclasse, existe apenas um construtor com o parâmetro `marca` do tipo `String`.

Podemos solucionar o problema apresentado de diversas maneiras, a primeira seria invocando o construtor da superclasse e encaminhando um parâmetro padrão para todas as instâncias da classe `Jipe`:

```
public class Jipe extends Veiculo{  
    public Jipe() {  
        super("Marca padrão de jipe");  
    }  
    //Demais membros...
```

**Construtor sem parâmetro**

**Invocação do construtor da superclasse**

Perceba que no construtor da classe `Jipe` estamos invocando o construtor da classe `Veiculo` e passando uma `String` ("`Marca padrão de jipe`"), ou seja, todas as instâncias desta classe `Jipe` possuirão a mesma marca.

Outra solução seria a implementação de um construtor na classe `Jipe` que receba o mesmo parâmetro repassando-o para a superclasse:

```
public class Jipe extends Veiculo{  
    public Jipe(String marca) {  
        super(marca);  
    }  
    //Demais membros...
```

**Construtor com parâmetro**

**Invocação do construtor da superclasse com repasse do parâmetro recebido**

Desta forma é mantida a lógica implementada pela superclasse. E por fim, uma terceira abordagem seria a mais de um construtor, observe que neste caso os dois construtores desenvolvidos anteriormente foram mantidos:

```
public class Jipe extends Veiculo{  
    private String modelo;  
    public Jipe(){  
        super("Marca padrão de jipe");  
    }  
    public Jipe(String marca) {  
        super(marca);  
    }  
    public Jipe(String marca, String modelo) {  
        super(marca);  
        this.modelo = modelo;  
    }  
    //Demais membros...
```

**Construtor novo com dois parâmetros**

Observe que, neste último caso, temos um construtor sem parâmetros, um

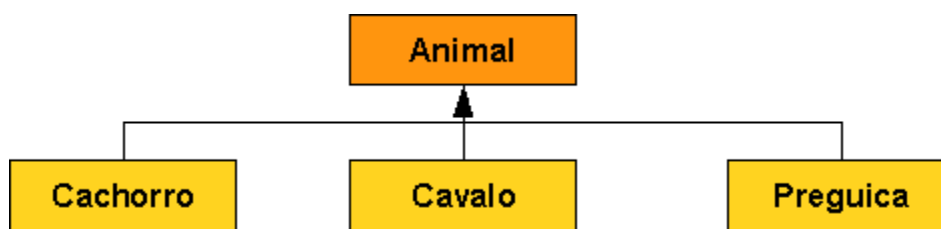
construtor que recebe o mesmo parâmetro da superclasse e outro que recebe **dois** parâmetros (marca e modelo), ou seja, poderíamos ter N-construtores, um para cada situação.

### Classes Abstratas

Até este momento todas as classes que havíamos desenvolvido eram classes **concretas**, isto é todas as nossas classes podem originar objetos. Classes concretas são estruturas definidas e prontas para instanciarem objetos.


Uma classe **abstrata** se comporta de forma diferente de uma classe **concreta**. Uma classe abstrata **nunca** pode ser instanciada de forma direta, seu maior propósito, a razão da sua existência, é ser estendida.

Para que serve esta classe então? Imagine a seguinte situação, nós temos uma hierarquia de classe conforme abaixo:



Todos os animais (cachorro, cavalo e preguiça) estendem da classe Animal, no entanto, no nosso programa a classe Animal por si só não representa nenhum tipo de animal. Isto é, esta classe representa apenas um animal genérico e ela existe apenas para ser estendida dando origem a um animal de fato.

Nesta situação é mais adequado que seja feita a mudança na classe Animal a fim de torná-la abstrata e inviabilizar a sua instanciação por qualquer parte do programa. Isto é realizado adicionando-se o modificador **abstract** conforme abaixo:

```
  
public abstract class Animal {  
    //corpo da classe  
}
```

Observe que agora a classe Animal não pode ser instanciada indevidamente em nenhum trecho do nosso programa, o exemplo abaixo irá produzir um erro de

compilação:

```
Animal animal = new Animal();
```



Desta forma nós temos certeza que durante a execução do programa nunca haverá um objeto da classe `Animal` e sim, sempre, de uma de suas subclasses.

Uma classe abstrata pode possuir métodos **concretos** – um método concreto é todo aquele possui corpo – veja o exemplo abaixo:

```
public abstract class Animal {  
    private String nome;  
    //getters e setters...  
    public void emitirSom()  
    {  
        System.out.println("Animal emitindo som");  
    }  
}
```

Método concreto

Perceba que o método `emitirSom()` na verdade deverá ser sobrescrito por cada uma das subclasses da classe `Animal` a fim de que forneçam um comportamento adequado pois, cada tipo de animal emite um som diferente dos demais.

Porém, da forma como foi feito, esta condição não é garantida pois, podemos ter uma subclasse da classe `Animal` que não sobrescreve o método `emitirSom`, no entanto existem formas de forçarmos esta condição. Ao modificarmos o método com **abstract** impedimos que alguma das subclasses não implementem o método. Segue exemplo abaixo:

```
public abstract class Animal {  
    private String nome;  
    //getters e setters...  
    public abstract void emitirSom();  
}
```

Método abstrato



Agora todas as subclasses da classe `Animal` irão obrigatoriamente ter que fornecer uma implementação para o método `emitirSom()`. Além da palavra reservada **abstract** devemos encerrar o método sem as chaves para que seja um método abstrato válido.


Uma classe abstrata pode conter apenas métodos abstratos, concretos ou ambos. No entanto a presença de um método abstrato torna, obrigatoriamente, a classe abstrata também.




## Curso Java Starter

---

A extensão de uma classe Abstrata é feita da mesma forma que a extensão realizada por classes concretas:

```
  
public class Cachorro extends Animal {  
  
    public void emitirSom() {  
        System.out.println("au! au!");  
    }  
  
}
```

Em Java não existe o conceito de herança múltipla, isto é, cada classe pode estender (**extends**) apenas **uma** superclasse de cada vez. Por exemplo, a seguinte situação não é permitida:

```
public class Cachorro extends Animal, Mamifero { 
```

No entanto podemos ter hierarquia de classes de qualquer tamanho o que viabilizaria a extensão de diversas classes nesta estrutura.

## Interfaces

Pense em uma interface como sendo uma classe totalmente abstrata, isto é que não possui métodos concretos. Desta forma, quando uma classe implementa uma interface ela obrigatoriamente deve implementar o conjunto de métodos definidos pela interface.

Do site da SUN, "implementar uma interface permite a uma classe tornar-se mais formal em relação ao comportamento que ela promete fornecer. Interfaces formam um contrato entre a classe e o mundo exterior, e este contrato é garantido em tempo de compilação pelo compilador. Se a sua classe deseja implementar uma interface, todos os métodos definidos pela interface devem aparecer no corpo antes da sua classe ser compilada".

Em síntese, quando uma classe concreta implementa uma interface ela está garantindo que possui os métodos especificados bem como a implementação destes, mesmo que o corpo do método seja vazio. Aqui cabe uma observação: O uso de interfaces garante a existência do comportamento porém não garante a sua correta implementação.

Da mesma forma que classes, a definição de uma interface cria um novo

## Curso Java Starter

tipo e nós devemos utilizar interfaces quando existem determinados comportamentos que sejam similares entre hierarquias de classes diferentes.

Veja o método `emitirSom()` da classe abstrata `Animal`, agora suponha que no mesmo programa eu tenha uma classe `Bola`, que por sinal também emite som. Em um determinado momento do programa eu desejo invocar o método `emitirSom` para alguns objetos conforme a situação abaixo:

```
public void limpar(Object obj)
{
    //Código para limpar uma bola ou um animal...

    if(obj instanceof Bola)
    {
        ((Bola)obj).emitirSom();
    } else if(obj instanceof Animal)
    {
        ((Animal)obj).emitirSom();
    }
}
```

Neste programa nós temos um método que efetua a limpeza de qualquer objeto, seja ele um animal ou uma bola, porém caso o objeto seja de um dos dois tipos, `Bola` ou `Animal`, estes deverão emitir seus sons característicos.

Esta situação não pode ser resolvida através do uso de herança já que a classe `Bola` e a classe `Animal` não estão na mesma hierarquia de classes – na verdade até poderia ser resolvida com herança e sobrescrita, mas havemos de convir que colocar a classe `Bola` e a classe `Animal` na mesma hierarquia é um erro de projeto – logo, a solução seria a criação da interface `EmiteSom` conforme abaixo:

```
public abstract interface EmiteSom {
    public abstract void emitirSom();
}
```

Observe que o uso do `abstract` é opcional tanto na classe quanto no método pois toda interface é obrigatoriamente abstrata e seus métodos também o são.

A classe `Animal` e a classe `Bola` ficariam da seguinte forma:

```
public class Bola implements EmiteSom{
    public void emitirSom()
    {
        System.out.println("Boing, boing...");
    }
}
```

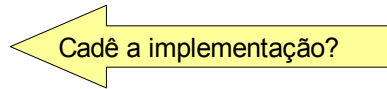


Implementação do método abstrato

## Curso Java Starter

---

```
public abstract class Animal implements EmiteSom{  
    private String nome;  
    //getters e setters...  
}
```

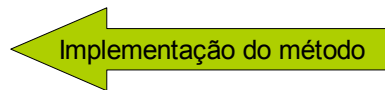


Perceba o uso da palavra reservada **implements** que informa ao compilador que esta classe está implementando uma interface. A classe `Animal` não fornece uma implementação para o método `emitirSom()`, ao contrário da classe `Bola`, isto acontece porque a classe `Animal` é do tipo `abstract`.

Quando uma classe abstrata implementa uma interface ela pode ou não fornecer a implementação dos métodos oriundos da interface. Caso a classe abstrata não forneça a implementação destes métodos (situação existente na classe `Animal`), obrigatoriamente cada subclasse concreta da classe abstrata irá ter que fornecer a implementação.

Por exemplo, a classe `Cachorro` fica da seguinte forma:

```
public class Cachorro extends Animal {  
    public void emitirSom() {  
        System.out.println("au! au!");  
    }  
}
```



Então, podemos interpretar a situação da seguinte forma: "A classe `Animal` implementa a interface `EmiteSom`, mas deixou a responsabilidade pela implementação do método para a classe concreta `Cachorro`".

E agora o nosso mesmo método `limpar()` utilizando a interface:

```
public void limpar(Object obj)  
{  
    //Código para limpar uma bola ou um animal...  
    if(obj instanceof EmiteSom)  
    {  
        ((EmiteSom)obj).emitirSom();  
    }  
}
```

Perceba que independente do tipo de objeto que vier como parâmetro, desde que ele seja do tipo `EmiteSom`, eu sei que irá possuir a implementação do método `emitirSom()` e que portanto o código será executado sem problemas, isto também nos permite acabar com aquelas cadeias indesejáveis de `if's` deixando o código mais legível.

### Exceções

Até agora nós criamos pequenos programas sem nos preocuparmos em tratar possíveis erros. No entanto todos aqueles que programam há algum tempo sabem que o tratamento de erros (exceções) é parte fundamental da programação.

Vamos começar os nossos estudos conhecendo a forma como uma exceção é apresentada e o seu significado, observe as duas classes abaixo:

```
public class modulo10Main {  
  
    public static void main(String[] args)  
    {  
        new RecebeArray(args);  
        System.out.println("Término do programa!");  
    }  
}  
  
public class RecebeArray {  
  
    public RecebeArray(String[] array)  
    {  
        imprimirPosicao0(array);  
    }  
    public void imprimirPosicao0(String[] array)  
    {  
        System.out.println(array[0]);  
    }  
}
```

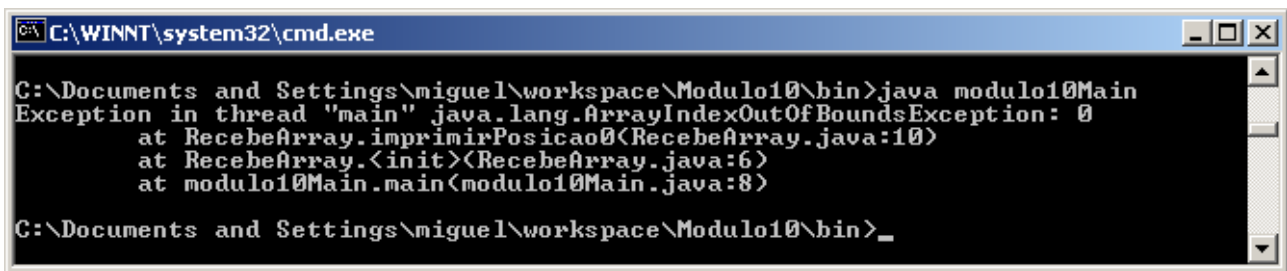
As setas indicam o fluxo de execução até o ponto onde irá acontecer a exceção (estrela) o que ocasiona a quebra no fluxo normal da aplicação. Esta execução pode ser entendida da seguinte forma:

Ao invocar o método main, sem o envio de parâmetros, é instanciado um novo objeto RecebeArray e passado como parâmetro do construtor o array vazio, o construtor por sua vez invoca um método desta mesma classe (imprimirPosicao0()) repassando o parâmetro recebido, este método tenta imprimir o elemento da posição 0, primeira posição, e o resultado obtido é uma exceção, já que o array está vazio.

Quando esta exceção acontece o programa é encerrado de forma inesperada, perceba que a última linha a ser executada (linha grifada em amarelo) não é percorrida devido ao término inesperado.

Podemos observar a visualização da execução desta aplicação na figura a seguir:

## Curso Java Starter



```
C:\WINNT\system32\cmd.exe
C:\Documents and Settings\miguel\workspace\Modulo10\bin>java modulo10Main
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at RecebeArray.imprimirPosicao0(RecebeArray.java:10)
    at RecebeArray.<init>(RecebeArray.java:6)
    at modulo10Main.main(modulo10Main.java:8)
C:\Documents and Settings\miguel\workspace\Modulo10\bin>_
```

A exceção foi do tipo **java.Lang.ArrayIndexOutOfBoundsException** que significa a tentativa de acesso a uma posição fora dos limites do array, a seguir ele informa que posição foi, neste caso o índice 0 (zero).

Embaixo da definição da exceção nós temos a **stacktrace** que contém a seqüência da pilha de execução. Pela stacktrace é possível perceber quais caminhos a execução percorreu até alcançar a exceção.

A primeira linha da stacktrace contém a definição da exceção, a segunda linha contém o último trecho de código executado com o número da linha no código fonte, neste caso foi a invocação do método **imprimirPosicao0()** da classe **RecebeArray**, que no código fonte (arquivo RecebeArray.java) refere-se a linha 10.

A fim de construirmos programas mais robustos, poderíamos tratar situações como esta. Para isto iremos utilizar o bloco **try/catch**:

```
try{
    //Código perigoso
}catch(Exception e)
{
    //tratamento da exceção
}
```

O bloco **try/catch** define dois trechos de código, um sujeito a erros (try) e outro que responsável pelo tratamento do erro caso aconteça (catch). Vamos modificar agora a nossa classe RecebeArray e adicionar este bloco ao método imprimirPosicao0():

```
public class RecebeArray {

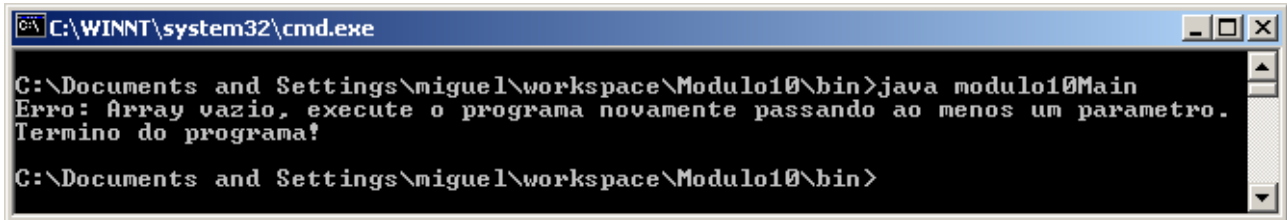
    public RecebeArray(String[] array)
    {
        imprimirPosicao0(array);
    }

    public void imprimirPosicao0(String[] array)
    {
        try{
            System.out.println(array[0]);
        }catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Erro: Array vazio, execute o programa novamente" +
                " passando ao menos um parâmetro.");
        }
    }
}
```

## Curso Java Starter

```
}  
}
```

E agora ao executarmos novamente este mesmo programa o usuário irá receber uma mensagem mais significativa, observe a execução abaixo:



```
C:\WINNT\system32\cmd.exe  
C:\Documents and Settings\miguel\workspace\Modulo10\bin>java modulo10Main  
Erro: Array vazio, execute o programa novamente passando ao menos um parametro.  
Termino do programa!  
C:\Documents and Settings\miguel\workspace\Modulo10\bin>
```

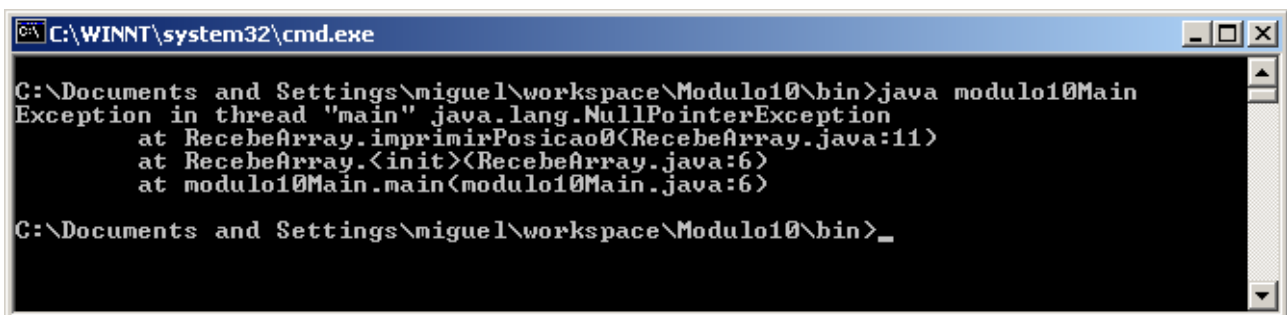
Pronto! Agora ao executar o programa sem parâmetros o usuário saberá o que fazer. Porém o mais interessante é o seguinte, uma vez que a exceção foi tratada, o programa continua o fluxo de execução e termina normalmente.

Devemos observar o seguinte, esta não é a única exceção que pode acontecer neste método, vamos modificar o método main() de forma que ele encaminhe como parâmetro um valor nulo (`null`).

```
public class modulo10Main {  
    public static void main(String[] args)  
    {  
        new RecebeArray(null);  
        System.out.println("Termino do programa!");  
    }  
}
```

Repare que agora a nova instância da classe RecebeArray está recebendo como parâmetro uma referência nula, isto ocasionará uma exceção do tipo `java.lang.NullPointerException`.

Vejamos o resultado da execução deste código:



```
C:\WINNT\system32\cmd.exe  
C:\Documents and Settings\miguel\workspace\Modulo10\bin>java modulo10Main  
Exception in thread "main" java.lang.NullPointerException  
at RecebeArray.inprimirPosicao0(RecebeArray.java:11)  
at RecebeArray.<init>(RecebeArray.java:6)  
at modulo10Main.main(modulo10Main.java:6)  
C:\Documents and Settings\miguel\workspace\Modulo10\bin>_
```

Ok! O programa encerrou de forma inesperada – a mensagem "Termino do

## Curso Java Starter

programa!” não foi apresentada – por uma exceção pois o nosso bloco try/catch não foi capaz de capturá-la, isto aconteceu pois ele foi construído de forma que apenas exceções do tipo **java.lang.ArrayIndexOutOfBoundsException** sejam tratadas.

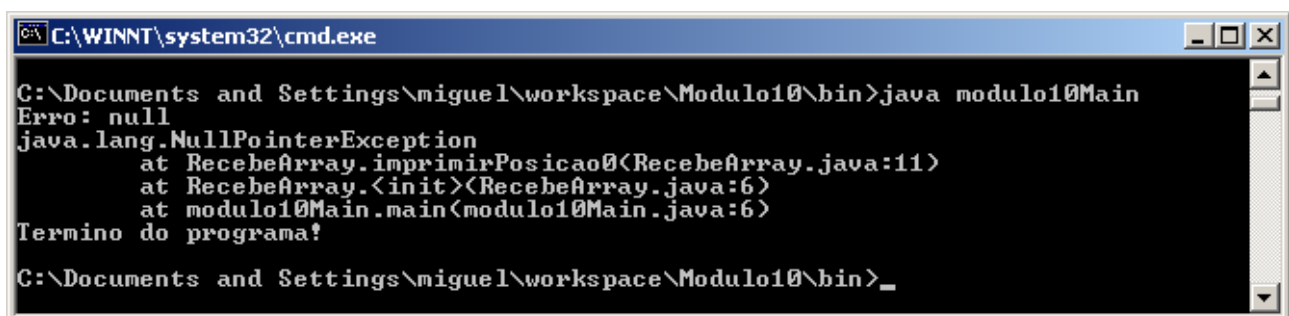
Para corrigir este problema podemos encadear mais um bloco catch que capture todos os tipos de exceções, vejamos:

```
public class RecebeArray {  
    public RecebeArray(String[] array)  
    {  
        imprimirPosicao0(array);  
    }  
    public void imprimirPosicao0(String[] array)  
    {  
        try{  
            System.out.println(array[0]);  
        } catch (ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("Erro: Array vazio, execute o programa novamente" +  
                " passando ao menos um parametro.");  
        } catch (Throwable t)  
        {  
            System.out.println("Mensagem do erro: "+t.getMessage());  
            t.printStackTrace();  
        }  
    }  
}
```

Perceba agora que, no código destacado, adicionamos mais um bloco de catch que captura todos os tipos de exceções – veremos mais a frente como está estruturada a hierarquia de classes de exceções – desta forma caso seja uma exceção do tipo `java.lang.ArrayIndexOutOfBoundsException` será tratado no primeiro bloco, todos os demais tipos serão tratados no segundo bloco.

O código destacado irá imprimir a mensagem (`getMessage()`) e na seqüência irá imprimir a stacktrace, é importante perceber novamente que após o tratamento da exceção pelo segundo bloco o programa é encerrado normalmente.

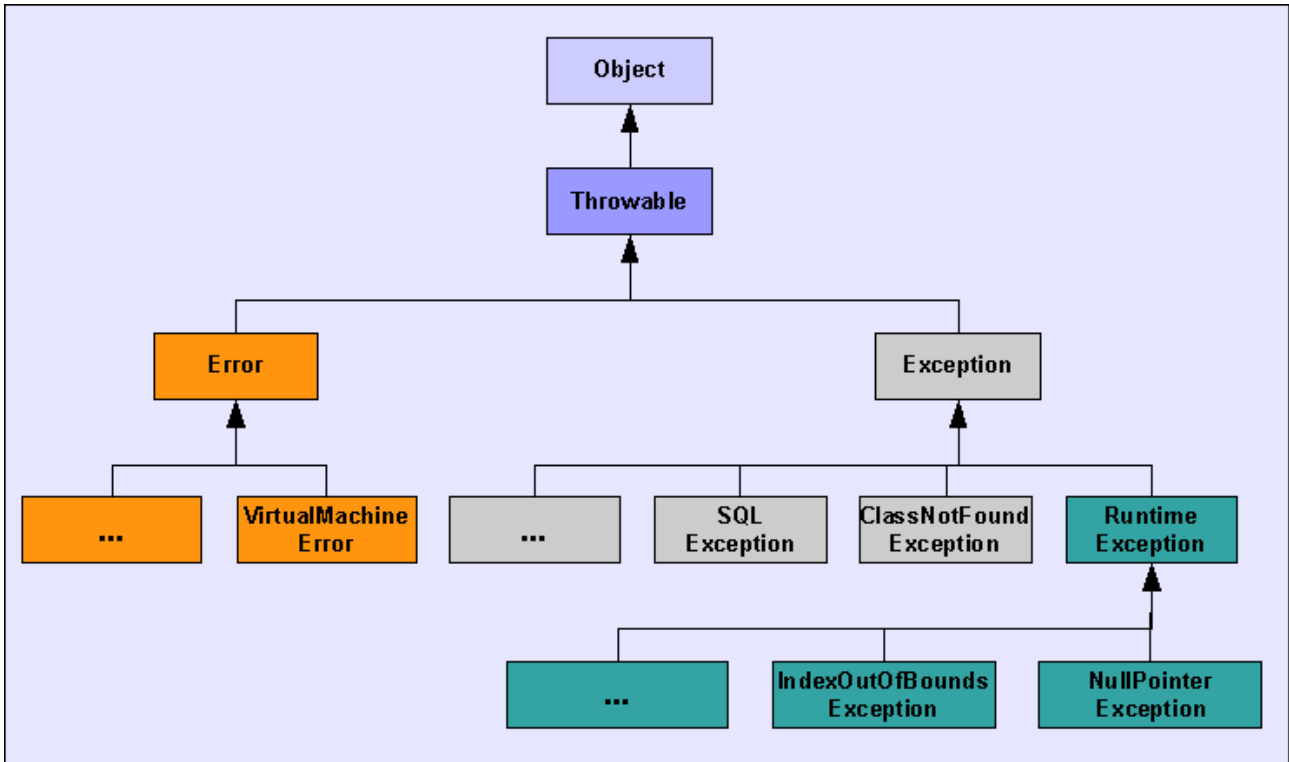
A seguir temos a execução do novo programa:



```
C:\WINNT\system32\cmd.exe  
C:\Documents and Settings\miguel\workspace\Modulo10\bin>java modulo10Main  
Erro: null  
java.lang.NullPointerException  
    at RecebeArray.imprimirPosicao0(RecebeArray.java:11)  
    at RecebeArray.<init>(RecebeArray.java:6)  
    at modulo10Main.main(modulo10Main.java:6)  
Termino do programa!  
C:\Documents and Settings\miguel\workspace\Modulo10\bin>_
```

## Curso Java Starter

Todas as exceções em Java derivam da classe Throwable conforme hierarquia a seguir:



Observe que, conforme dito em módulos anteriores, todas as classes em Java estendem de alguma forma – direta ou indiretamente – da classe Object.

Nesta imagem estão representadas as três modalidades de exceções existentes na linguagem Java: **Unchecked Exception**, **Checked Exception** e **Error**.

- **Error**: Hierarquia em laranja na imagem, representam situações incomuns, que não são causadas pelo programa, indicam situações que não acontecem usualmente durante a execução de um programa (Ex: Estouro da pilha de execução – StackOverflowError);
- **Checked Exception**: Hierarquia em cinza, representam situações que, geralmente, não são erros de programação e sim indisponibilidade de recursos ou condição necessária para a correta execução inexistente (Ex: Em aplicações distribuídas existe dependência externa de rede de



comunicação – NoRouteToHostException).

- **Unchecked Exception (RuntimeException):** Hierarquia em turquesa (azul escuro), representam situações que, geralmente, identificam erros de programação (programa não é robusto) ou mesmo situações incomuns/difíceis de tratar (Ex: Acessar índice inválido em um array – `ArrayIndexOutOfBoundsException`);

As checked exceptions são tratadas obrigatoriamente, isto é, o compilador só compila a classe se houver um tratamento (bloco try/catch) para aquele tipo de exceção.

Segue quadro com relação de algumas das mais comuns exceções:

Exceção	Quando acontece
<code>ArrayIndexOutOfBoundsException</code>	Tentativa de acesso a posição inexistente no array.
<code>ClassCastException</code>	Tentativa de efetuar um cast em uma referência que não é classe ou subclasse do tipo desejado.
<code>IllegalArgumentException</code>	Argumento formatado de forma diferente do esperado pelo método.
<code>IllegalStateException</code>	O estado do ambiente não permite a execução da operação desejada.
<code>NullPointerException</code>	Acesso a objeto que é referenciado por uma variável cujo valor é null.
<code>NumberFormatException</code>	Tentativa de converter uma String inválida em número.
<code>StackOverflowError</code>	Normalmente acontece quando existe uma chamada recursiva muito profunda.
<code>NoClassDefFoundError</code>	Indica que a JVM não conseguiu localizar uma classe necessária a execução.

### Jogando (throw) exceções

Possivelmente em algum trecho do nosso programa, se uma determinada condição acontecer, nós queremos lançar uma exceção – por incrível que possa parecer – para informar que a situação esperada não aconteceu.

## Curso Java Starter

Vejamos um exemplo, suponha o método `abastecer()` pertencente a uma classe que representa um veículo. Este método recebe como parâmetro um `Double` que representa o valor que está sendo abastecido, logo não pode haver situações onde o valor seja negativo. Caso esta situação aconteça nós iremos lançar uma `IllegalArgumentException`.

```
public void abastecer(Double litros)
{
    if(litros < 0)
    {
        throw new IllegalArgumentException("Era esperado um valor maior que 0:
"+litros);
    } else
    {
        tanque += litros;
    }
}
```

A seta indica o uso da palavra reservada `throw` que é responsável por “jogar” a exceção, logo a seguir nós instanciamos um objeto do tipo `IllegalArgumentException` e passamos como parâmetro do método construtor o texto que deverá aparecer quando a stacktrace for impressa.

Veja o resultado da execução deste método passando o valor `-1` (um negativo):

```
Exception in thread "main" java.lang.IllegalArgumentException: Valor indevido
era esperado um valor maior que 0: -1.0
    at Veiculo.abastecer(Veiculo.java:10)
    at modulo10Main.main(modulo10Main.java:6)
```

### Criando sua própria exceção

Suponha que ao invés de utilizar a `IllegalArgumentException` no método `abastecer()`, caso o valor recebido seja negativo, nós desejemos “jogar” uma exceção própria chamada “`QuantidadeLitrosException`”, neste caso basta que utilizemos os nossos conhecimentos em herança e façamos uma extensão da classe `RuntimeException` conforme abaixo:

```
public class QuantidadeLitrosException extends RuntimeException
{
    QuantidadeLitrosException(String mensagem)
    {
        super(mensagem);
    }
}
```

## Curso Java Starter

A linha em destaque significa que estou invocando o construtor da superclasse (Exception) e enviando o parâmetro mensagem para ela. O novo método abastecer fica da seguinte forma:

```
public void abastecer(Double litros)
{
    if(litros < 0)
    {
        throw new QuantidadeLitrosException("Valor indevido. Era esperado um valor
maior que 0: "+litros);
    }else
    {
        tanque += litros;
    }
}
```

A execução deste método com um valor negativo resulta em:

```
Exception in thread "main" QuantidadeLitrosException: Valor indevido. Era esperado um valor maior
que 0: -1.0
    at Veiculo.abastecer(Veiculo.java:10)
    at modulo10Main.main(modulo10Main.java:6)
```

Na prática, são poucos os motivos que nos levam a criar as nossas próprias exceções, normalmente as exceções existentes nas bibliotecas Java são suficientes para a maioria das ocorrências.

## Exercícios

Aprenda com quem também está aprendendo, veja e compartilhe as suas respostas no nosso [Fórum](#):

[Exercícios – Módulo 08 – Classes Abstratas, Interfaces e Exceções](#)

1. Crie uma classe abstrata que represente um quadrilátero e receba como parâmetros do construtor os quatro valores referentes a cada lado.
2. Estenda a classe criada no exercício 1, a subclasse deve representar um **quadrado** e portanto receber como parâmetro um único valor referente aos seus lados.
3. Estenda novamente a classe criada no exercício 1, a subclasse deve representar um **retângulo** e portanto deve receber como parâmetros dois valores diferentes.
4. Crie uma interface denominada FiguraGeometrica, adicione os métodos calcular área e calcular perímetro.

## Curso Java Starter

---

5. Modifique a classe abstrata criada no exercício 1 de forma que ela implemente a interface `FiguraGeométrica`.
6. Implemente os métodos definidos na interface `FiguraGeometrica` nas classes `Quadrado` e `Retangulo`.
7. Modifique o construtor da classe `Quadrado` de forma que caso seja recebido um valor igual a zero ou negativo seja "jogada" uma exceção do tipo **`IllegalArgumentException`** com o seguinte texto: "Valor inválido, o valor esperado é maior que 0 (zero)".
8. Modifique o construtor da classe `Retangulo` de forma que caso seja recebido como parâmetro um valor igual a zero ou negativo ou ambos os valores positivos idênticos porém idênticos seja "jogada" duas exceções do tipo **`IllegalArgumentException`** com os seguintes textos respectivamente: "Valor inválido, os valores esperados são maior que 0 (zero)" e "Valor inválido, modifique um dos valores a fim de torná-los diferentes".
9. Crie um programa que solicite a entrada de 2 parâmetros, crie um `Quadrado` e imprima a área e o perímetro.
10. Modifique o programa desenvolvido no exercício 9. Adicione o tratamento de exceções e caso aconteça uma exceção imprima a stacktrace. Execute o programa forçando uma exceção e observe a stacktrace.
11. Modifique novamente o programa desenvolvido no exercício 9 de forma que caso sejam passados valores inválidos ele trate a exceção e exiba a mensagem em um diálogo (`JOptionPane`).
12. Crie uma classe abstrata denominada conta corrente. Adicione os métodos `sacar`, `depositar` e `obter saldo`, adicione o atributo `saldo total`.
13. Estenda a classe desenvolvida no exercício anterior. Crie a classe `conta corrente especial` que contenha um atributo que represente um limite extra da conta corrente convencional de forma que:
  1. O saque debita primeiro o `saldo total`, na seqüência deve-se debitar o limite do cheque especial;
  2. O depósito primeiro deve creditar o limite do cheque especial, até cobrir, e somente após o `saldo total`.
14. Crie uma exceção que represente o estouro da conta corrente, modifique a classe desenvolvida no exercício 13 de forma que caso seja extrapolado o limite do extra esta exceção seja lançada.
15. Crie um programa que, utilizando as classes desenvolvidas nos exercícios

## Curso Java Starter

---

anteriores (12, 13 e 14), efetue alguns saques e depósitos, tanto sobre o limite extra quanto sobre saldo total, e ainda faça um saque que extrapole todo o valor disponível (limite extra + saldo total).