JAVA

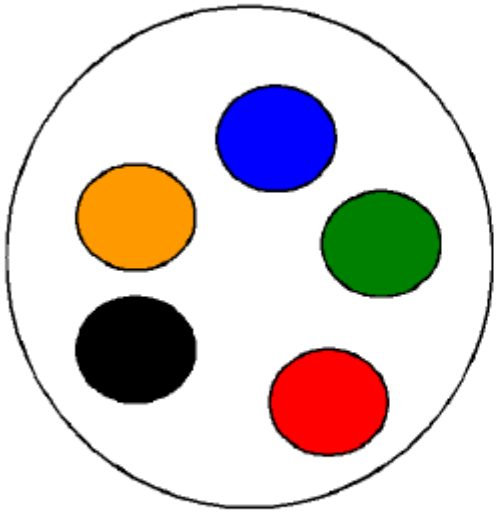
Tópicos Especiais
Informática Industrial – CEFET

Prof. Gustavo Guedes

Atualização: 16/03/2011

Conjunto

- Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.
- Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto. A interface não define como deve ser este comportamento. Tal ordem varia de cada implementação para implementação.



Possíveis ações em um conjunto:

- A camiseta Azul está no conjunto?
- Remova a camiseta Azul.
- Adicione a camiseta Vermelha.
- Limpe o conjunto.

- Não existem elementos duplicados!

- Ao percorrer um conjunto, sua ordem não é conhecida!

Conjuntos

```
Set conjunto = new HashSet();

conjunto.add("paulo");
conjunto.add("guilherme");
conjunto.add("thadeu");
conjunto.add("cosen");
conjunto.add("sergio");
conjunto.add("guilherme"); // repetido!

// imprime na tela todos os elementos
System.out.println(conjunto);
```

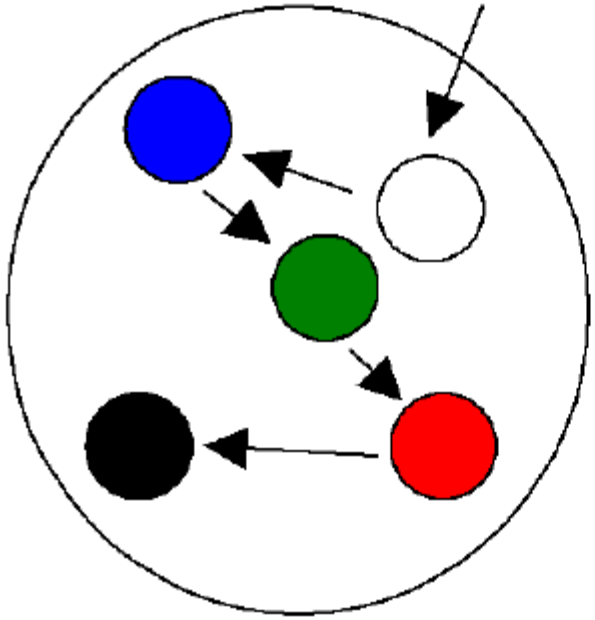
Conjuntos

- Mostrar método com retorno um HashSet
- E Se eu quiser mudar a implementação para treeset

Iterator

- Toda coleção fornece acesso a um Iterator, um objeto que implementa a interface Iterator, que conhece internamente a coleção e dá acesso a todos os seus elementos, como a figura abaixo mostra. A ideia do Iterator é retirar da coleção a responsabilidade de acessar e caminhar na estrutura e colocar a responsabilidade em um novo objeto separado chamado Iterator.

Iterator



Possíveis ações em um iterador:

- Existe um próximo elemento na coleção?.
- Pegue o próximo elemento.
- Remova o elemento atual da coleção.

```
ContaCorrente c =
  (ContaCorrente) iterator.next();
```

Iterator

- O Iterator nada mais é do que uma interface que permite percorrer os elementos de uma Collection. Abaixo podemos ver sua utilização:

```
Iterator elementos = lista.iterator();
while (elementos.hasNext()){
    System.out.println(elementos.next());
}
```

- Tente fazer com o "for".
- PS: Tente criar o Iterator e depois inserir os elementos no conjunto ou lista. Nada será exibido. Por quê?

Iterator

ava.util

Interface Set

All Superinterfaces:
Collection

All Known Subinterfaces:
SortedSet

All Known Implementing Classes:
AbstractSet, HashSet, LinkedHashSet, TreeSet

java.util

Interface List

All Superinterfaces:
Collection

docs.oracle.com/javase/1.4.2/docs/api/java/util/Collection.html

	Returns `true` if this collection contains no elements.
Iterator	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(Collection c) Removes all this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.

Iterator

- A interface Iterator possui dois métodos principais: hasNext() (com retorno booleano), indica se ainda existe um elemento a ser percorrido; next(), retorna o próximo objeto.

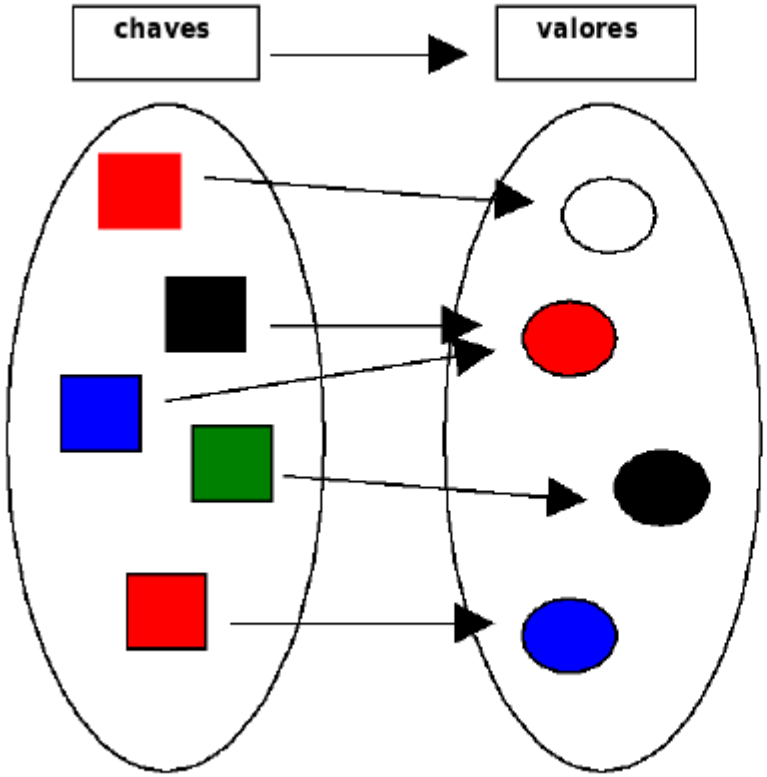
- List -> procurando objetos
- Set -> procurando objetos

- HashSet

Map

- Muitas vezes queremos buscar rapidamente um objeto dada alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa.
- Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. É equivalente ao conceito de dicionário, usado em várias linguagens. Algumas linguagens, como Perl ou PHP, possuem um suporte mais direto a mapas, onde são conhecidos como matrizes/arrays associativas.
- java.util.Map é um mapa, pois é possível usá-lo para mapear uma chave a um valor, por exemplo: mapeie à chave “cpf” o valor de um objeto da classe pessoa. Semelhante a associações de palavras que podemos fazer em um dicionário.

Map



Possíveis ações em um mapa:

Mapeie uma chave a um valor
O que está mapeado na chave X?
Remapeie uma certa chave
Quero o conjunto de chaves.
Quero o conjunto de valores.
Desmapeie a chave X.

Map

- O método put(Object, Object) da interface Map recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado “objeto-chave”, passa-se esse “objeto-chave” no método get(Object).
- Sem dúvida essas são as duas operações principais e mais frequentes realizadas sobre um mapa.
- Observe o exemplo: criamos duas contas correntes e as colocamos em um mapa associando-as aos seus donos.

Map

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);

// cria o mapa
Map mapaDeContas = new HashMap();

// adiciona duas chaves e seus respectivos valores
mapaDeContas.put("diretor", c1);
mapaDeContas.put("gerente", c2);

// qual a conta do diretor?
ContaCorrente contaDoDiretor = (casting!) mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

Map

- Um mapa é muito usado para “indexar” objetos de acordo com determinado critério para podermos buscar esse objetos rapidamente.
- Ele, assim como as coleções, trabalha diretamente com Object’s (tanto na chave quanto no valor), o que tornaria necessário o casting no momento que recuperar elementos.

- Suas principais implementações são o HashMap, o TreeMap e o Hashtable.

Map

- Criar a classe Pessoa com nome, idade, endereco e cpf.
- Criar 3 pessoas.
- Inserir no mapa (chave = cpf).
- Criar um novo método que recebe como argumento um mapa e a chave que quer buscar. O método retorna Object.
- No main, chame o método acima. Pegue o objeto (ainda no main) e imprima o nome, idade, endereco e cpf desse objeto (no main).

Map

- Para percorrer um mapa:
  - mapa.keySet().iterator() -> itera pelas chaves
  - Mapa.values().iterator() ->itera pelos valores

Exceções

- Observe a classe abaixo. A solução mais simples utilizada antigamente para apresentar um erro era marcar o retorno de um método como boolean e retornar true, se tudo ocorreu como deveria, ou false, caso contrário:

```
public class Conta {
    private double limite;
    private double saldo;
    public boolean saca (double quantidade) {
        if (quantidade > this.saldo + this.limite) { //posso sacar até saldo+limite
                System.out.println("Não posso sacar fora do limite!");
                return false;
            } else {
                this.saldo = this.saldo - quantidade;
                return true;
            }
    }
}
```

- Dessa maneira, teríamos o método main da seguinte forma:

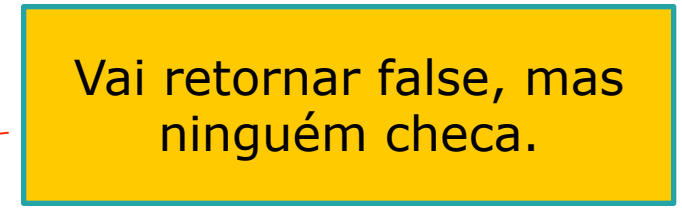
```
public class TesteConta {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        if (!minhaConta.saca(1000)) {
            System.out.println("Não saquei");
        }
    }
}
```

Exceções

- Repare que tivemos de lembrar de testar o retorno do método, mas não somos obrigados a fazer isso.

- Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de autoatendimento poderia vir a liberar a quantia desejada de dinheiro, mesmo que o sistema não tivesse conseguido efetuar o método saca com sucesso, como no exemplo a seguir. Imagine que o saldo da conta seja 0:

```
public class TesteConta {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        int valor = 1000;
        minhaConta.saca(valor);
        caixaEletronico.emiteValor(valor);

    }
}
```



Exceções

- Por esses e outros motivos, utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não iria acontecer. O exemplo do argumento do saque inválido ou do id inválido de um cliente é uma exceção à regra.

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

Exceções

- Antes de entrarmos nas exceções, vamos ver o que ocorre quando alguma situação inesperada ocorre:

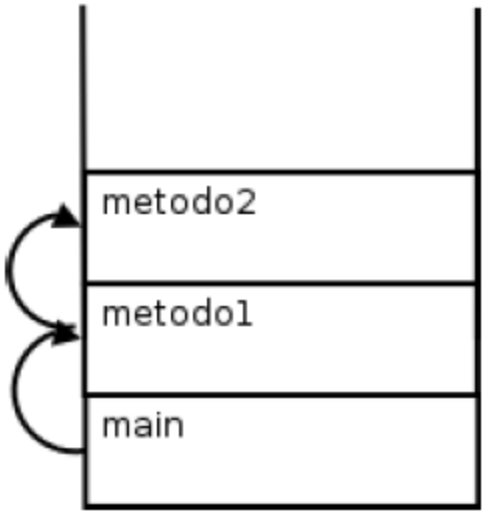
Exceções

```
public class TesteErro {

    public static void main(String[] args) {
        System.out.println("inicio do main");
        metodo1();
        System.out.println("fim do main");
    }

    public static void metodo1() {
        System.out.println("inicio do metodo1");
        metodo2();
        System.out.println("fim do metodo1");
    }

    public static void metodo2() {
        System.out.println("inicio do metodo2");
        int[] array = new int[10];
        for (int i = 0; i <= 15; i++) {
            array[i] = i;
            System.out.println(i);
        }
        System.out.println("fim do metodo2");
    }
}
```

Exceções

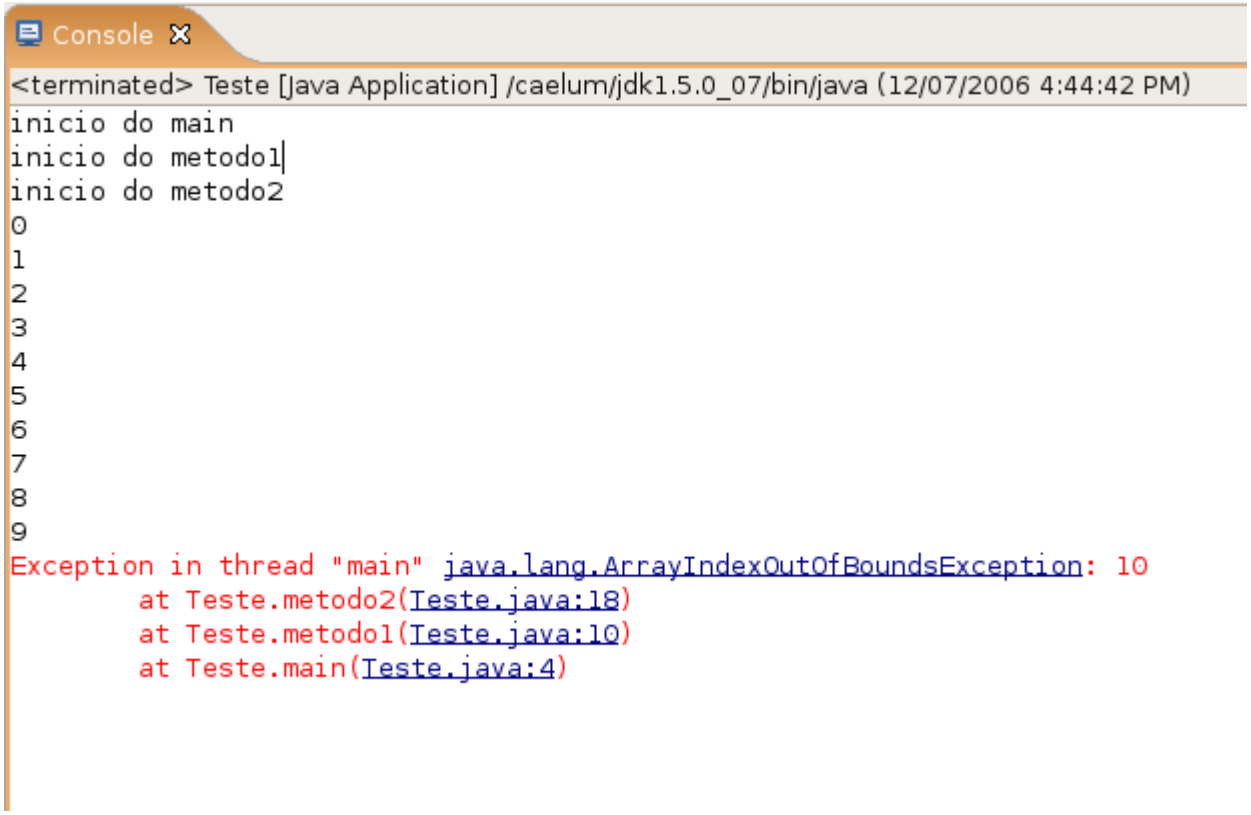
- Repare o método main chamando metodo1 e esse, por sua vez, chamando o metodo2. Cada um desses métodos pode ter suas próprias variáveis locais, sendo que, por exemplo, o metodo1 não enxerga as variáveis declaradas dentro do metodo2.
- Como o Java (e muitas das outras linguagens) faz isso? Toda invocação de método é empilhada... em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da pilha de execução (stack). Basta jogar fora um gomo da pilha (stackframe):

Exceções



- Porém, o nosso metodo2 propositadamente possui um enorme problema: está acessando um índice de array
- indevido para esse caso; o índice estará fora dos limites da array quando chegar em 10!
- Rode o código. Qual é a saída? O que isso representa? O que ela indica?

Exceções



Esse é o conhecido rastro da pilha (stacktrace). É uma saída importantíssima para o programador.

Exceções

- Em qualquer forum ou lista de discussão, é comum enviarem junto com a descrição do problema essa stacktrace. Por que isso aconteceu? O sistema de exceções do Java funciona da seguinte maneira:
- Quando uma exceção é lançada (throws) a JVM entra em um estado de alerta e vai ver se o método atual toma alguma precaução ao tentar executar esse trecho de código. Como podemos ver, o metodo2 não toma nenhuma medida diferente do que vimos até agora.
- Como o metodo2 não está tratando esse problema, a JVM pára a execução dele anormalmente, sem esperar
- ele terminar, e volta um stackframe pra baixo, onde será feita nova verificação: o metodo1 está se precavendo
- de um problema chamado ArrayIndexOutOfBoundsException? Não... volta para o main, onde também não há proteção, então a JVM morre (na verdade, quem morre é apenas a Thread corrente, veremos mais para frente).

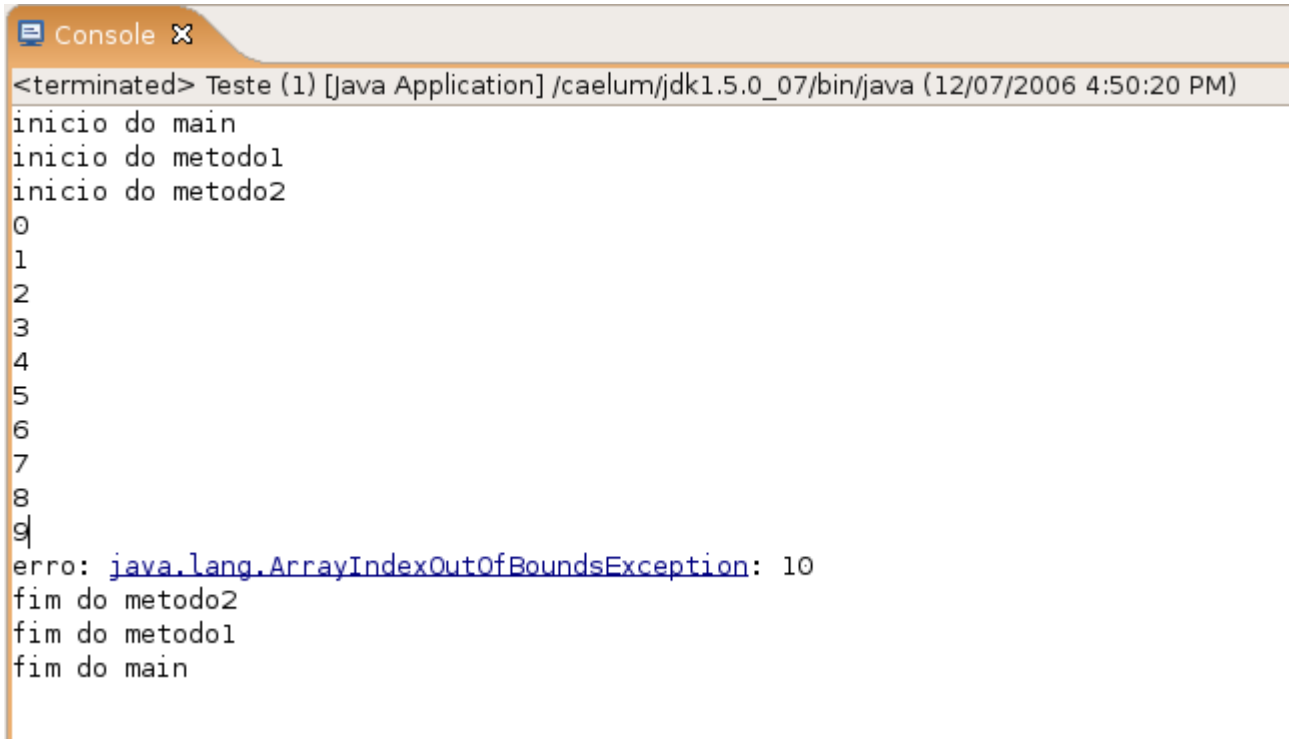
Exceções

- Obviamente, aqui estamos forçando esse caso, e não faria sentido tomarmos cuidado com ele. É fácil arrumar um problema desses: basta percorrermos a array no máximo até o seu length.
- Porém, apenas para entender o controle de fluxo de uma Exception, vamos colocar o código que vai **tentar** (try) executar o bloco perigoso e, caso o problema seja do tipo ArrayIndexOutOfBoundsException, ele será **pego** (catched). Repare que é interessante que cada exceção no Java tenha um tipo... ela pode ter atributos e métodos.

- Adicione um try/catch em volta do for, pegando ArrayIndexOutOfBoundsException. O que o código imprime agora?

```
try {

    for (int i = 0; i <= 15; i++) {
        array[i] = i;
        System.out.println(i);
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
```

Exceções


Console
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:50:20 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo2
fim do metodo1
fim do main

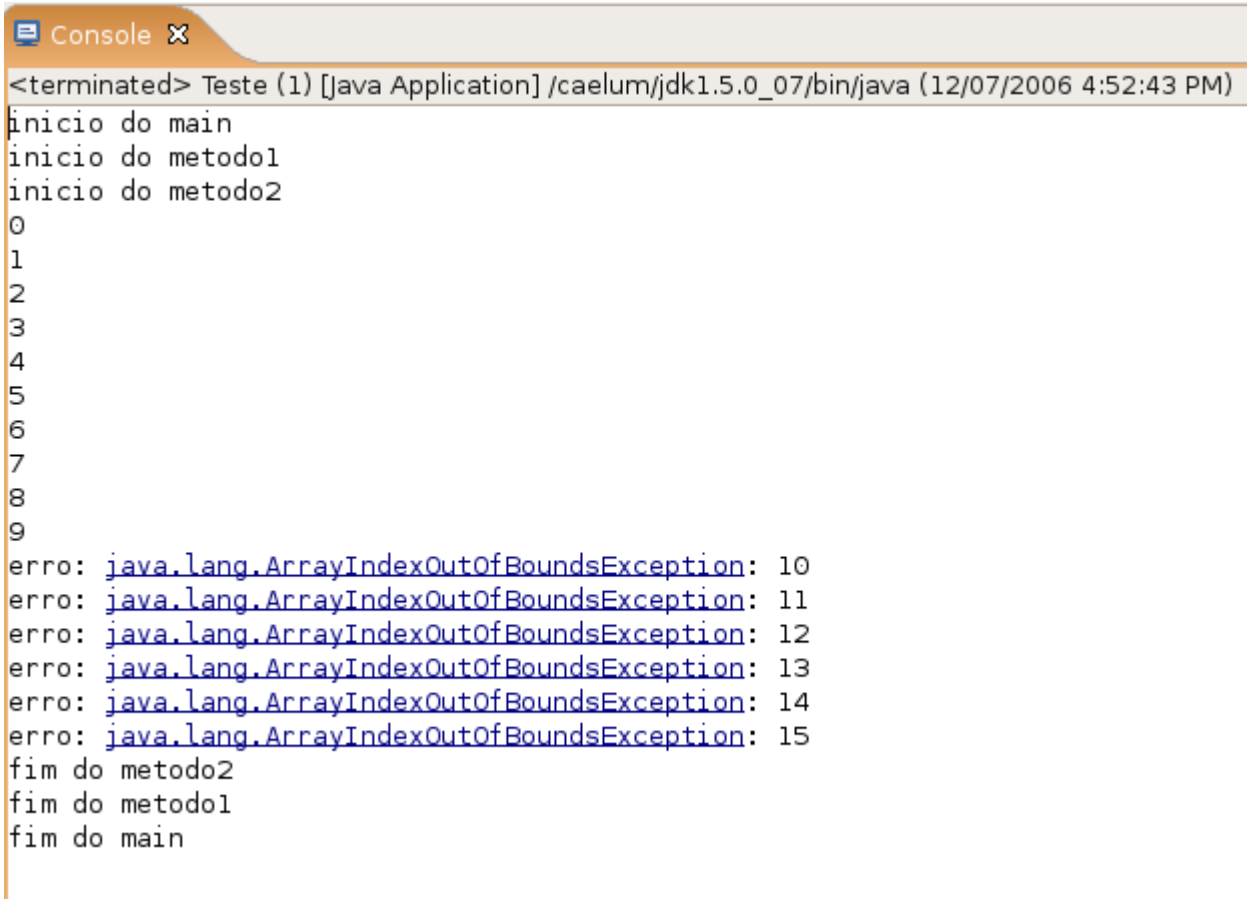
Exceções

- Em vez de fazer o try em torno do for inteiro, tente apenas com o bloco de dentro do for:

```
for (int i = 0; i <= 15; i++) {

    try {
        array[i] = i;
        System.out.println(i);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("erro: " + e);
    }
}
```

- Qual a diferença?

Exceções

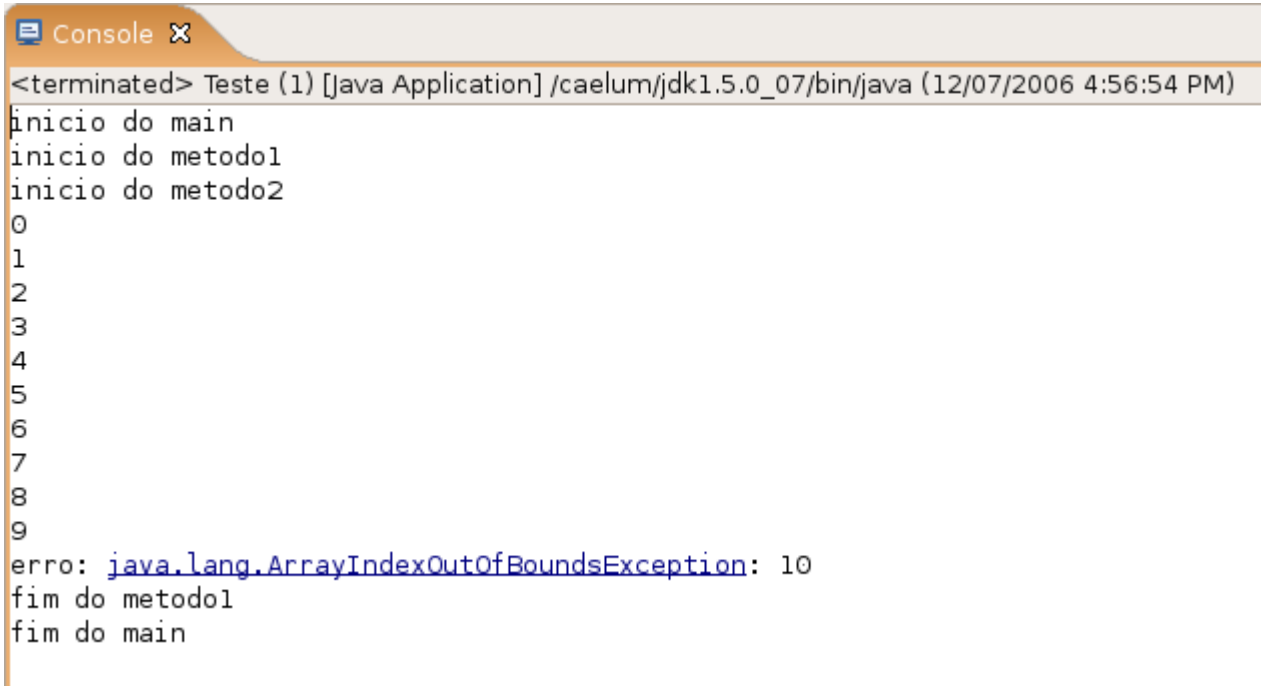

Console
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:52:43 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
erro: java.lang.ArrayIndexOutOfBoundsException: 11
erro: java.lang.ArrayIndexOutOfBoundsException: 12
erro: java.lang.ArrayIndexOutOfBoundsException: 13
erro: java.lang.ArrayIndexOutOfBoundsException: 14
erro: java.lang.ArrayIndexOutOfBoundsException: 15
fim do metodo2
fim do metodo1
fim do main

Exceções

- Agora retire o try/catch e coloque ele em volta da chamada do metodo2.

```
System.out.println("inicio do metodo1");

try {
    metodo2();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
System.out.println("fim do metodo1");
```

Exceções

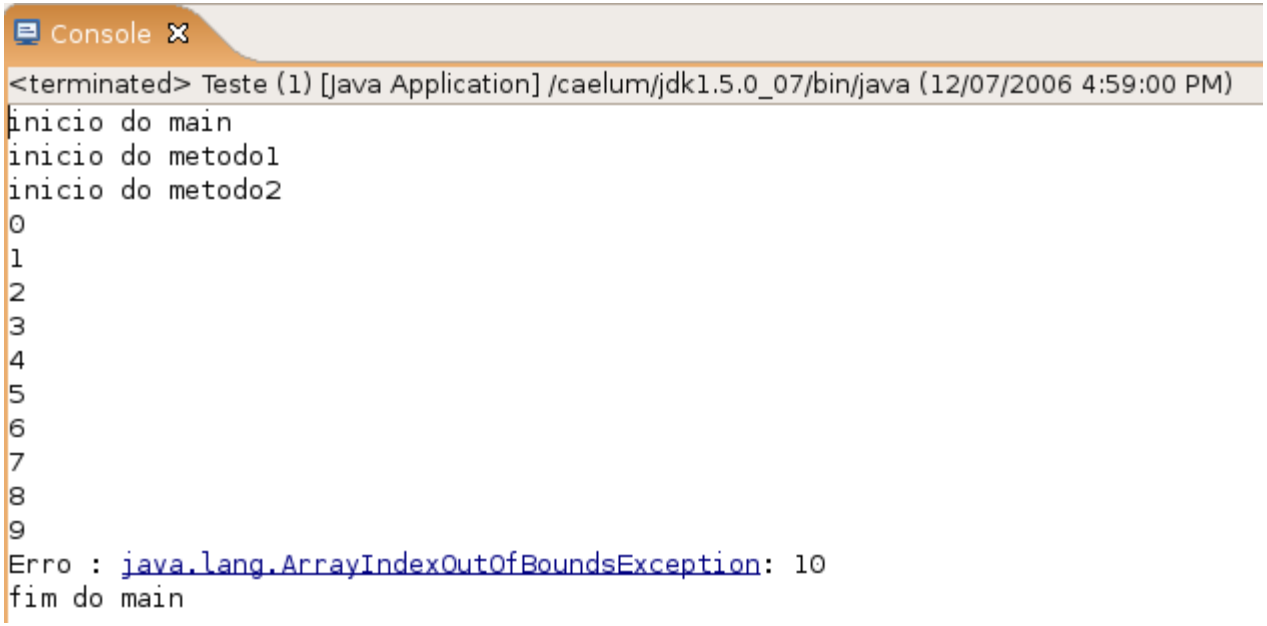

Console
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:56:54 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo1
fim do main

Exceções

- Faça o mesmo, retirando o try/catch novamente e colocando em volta da chamada do metodo1. Rode os
- códigos, o que acontece?

```
System.out.println("inicio do main");

try {
    metodo1();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Erro : "+e);
}
System.out.println("fim do main");
```

Exceções



- Repare que, a partir do momento que uma exception foi catched (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

Exceções

```
public class TestandoUmaReferenciaNula {

    public static void main(String[] args) {
        Conta c = null;
        System.out.println(c.getSaldo());

    }

}
```

- Tente executar esse código. O que ocorre?

Exceções

- Um ArrayIndexOutOfBoundsException ou um NullPointerException poderia ser facilmente evitado
- com o for corretamente escrito ou com ifs que checariam os limites da array.
- Outro caso em que também ocorre tal tipo de exceção é quando um cast errado é feito.
- Em todos esses casos, tais problemas provavelmente poderiam ser evitados pelo programador. É por esse
- motivo que o java não te obriga a dar o try/catch nessas exceptions e chamamos essas exceções de unchecked.
- Em outras palavras, o compilador não checa se você está tratando essas exceções.

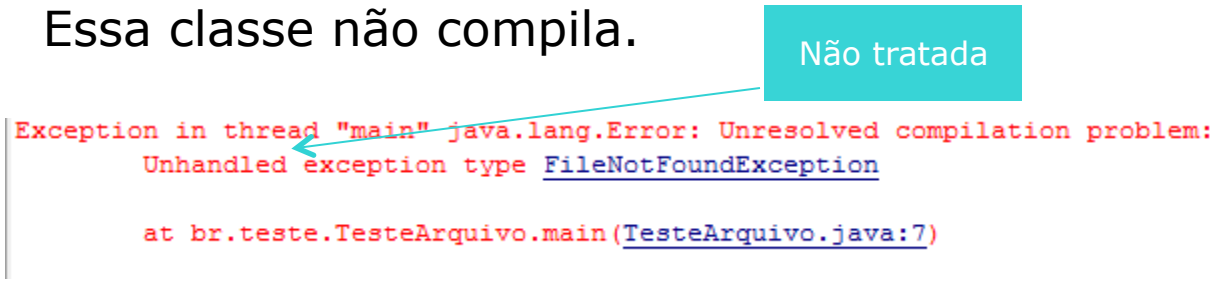
Exceções

- Todos os exemplos que fizemos compilaram e rodaram sem o try/catch (mesmo que tenham gerado um erro), certo? Existe um outro tipo de exceção que obriga a quem chama o método ou contrutor a tratar essa exceção. Chamamos esse tipo de exceção de checked, pois o compilador checará se ela está sendo devidamente tratada, diferente das anteriores, conhecidas como unchecked. Um exemplo que podemos mostrar é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir.

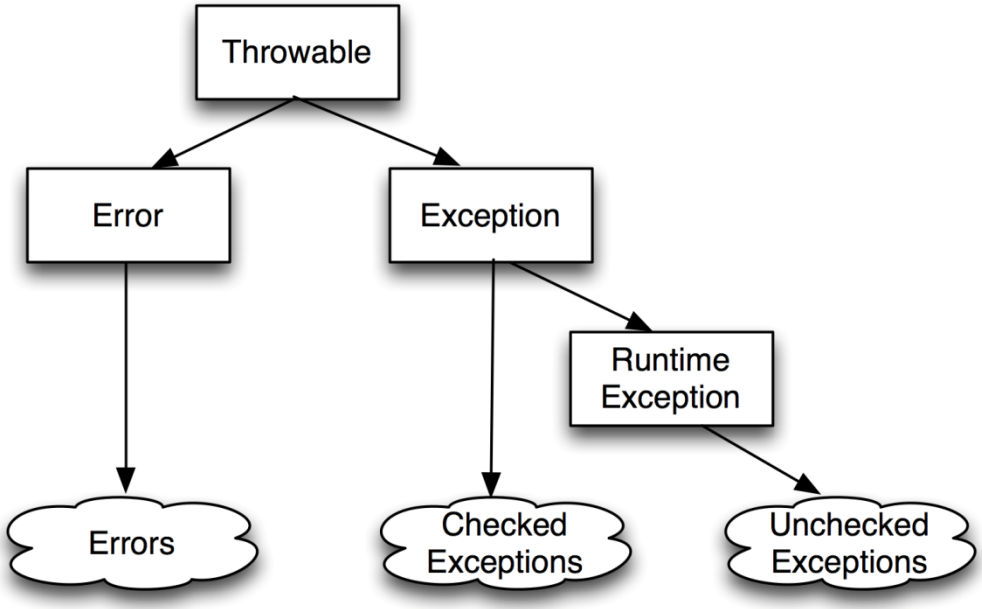
```
import java.io.FileInputStream;

public class TesteArquivo {
    public static void main(String[] args) {
        new FileInputStream("arquivo.txt");

    }

}
```

Exceções

- Essa classe não compila.

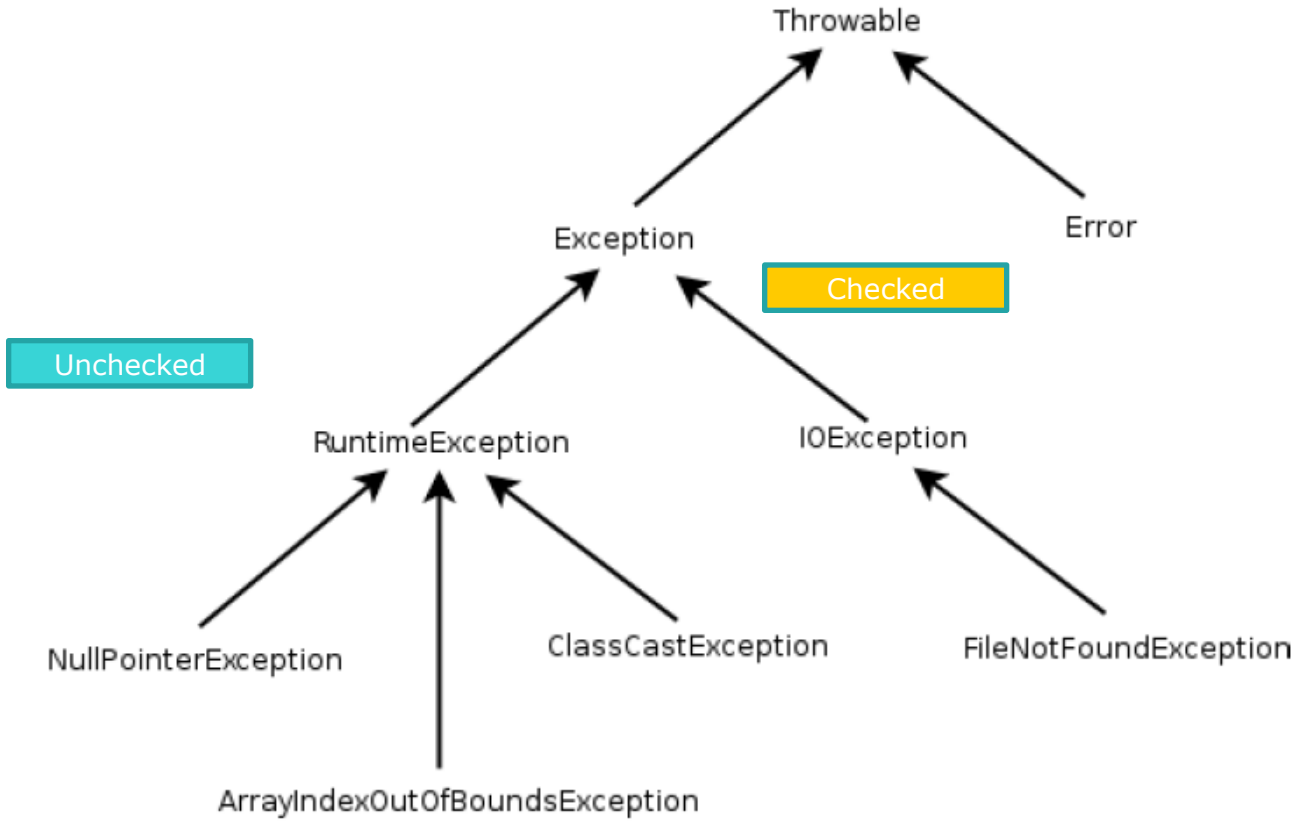


Exceções


Throwable
Error
Exception
Runtime Exception
Errors
Checked Exceptions
Unchecked Exceptions

Exceções

Uma pequena parte da Família Throwable:



Exceções

- Podemos, também, lançar uma Exception, o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um if no retorno de um método.

- A palavra chave **throw**, que está no imperativo, lança uma Exception. Isto é bem diferente de throws, que está no presente do indicativo, e que apenas avisa da possibilidade daquele método lançá-la, obrigando o outro método que vá utilizar deste de se preocupar com essa exceção em questão.

Exceções

- Lembre-se do método saca da nossa classe Conta. Ele devolve um boolean caso consiga ou não sacar:

```
boolean saca(double valor){
    if (this.saldo < valor){
        return false;
    } else {
        this.saldo-=valor;
        return true;
    }
}
```

- Agora, ele irá lançar uma exceção do typo unchecked:

```
void saca(double valor){
    if (this.saldo < valor){
        throw new RuntimeException();
    } else {
        this.saldo-=valor;
    }
}
```

Exceções

- RuntimeException é a exception mãe de todas as exceptions unchecked. A desvantagem, aqui, é que ela é muito genérica; quem receber esse erro não sabe dizer exatamente qual foi o problema. Podemos então usar uma Exception mais específica:

```
void saca(double valor){
    if (this.saldo < valor){
        throw new IllegalArgumentException();
    } else {

        this.saldo-=valor;
    }
}
```

Exceções

- IllegalArgumentException diz um pouco mais: algo foi passado como argumento e seu método não gostou.Ela é uma Exception unchecked pois estende de RuntimeException e já faz parte da biblioteca do java.
- (IllegalArgumentException é a melhor escolha quando um argumento sempre é inválido como, por exemplo, números negativos, referências nulas, etc).
- E agora, para pegar esse erro, não usaremos um if/else e sim um try/catch, porque faz mais sentido já que a falta de saldo é uma exceção:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println("Saldo Insuficiente");
}
```

Exceções

- Podíamos melhorar ainda mais e passar para o construtor da IllegalArgumentException o motivo da exceção:

```
void saca(double valor){
    if (this.saldo < valor){
        throw new IllegalArgumentException("Saldo insuficiente");
    } else {
        this.saldo-=valor;
    }
}
```

- O método getMessage() definido na classe Throwable (mãe de todos os tipos de erros e exceptions) vai retonar a mensagem que passamos ao construtor da IllegalArgumentException.

Exceções

```
try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

Exceções

- A declaração finally define um bloco de código que sempre é executado, mesmo que uma exceção seja identificada. O exemplo de código a seguir descreve esse procedimento:

```
1     try {
2       startFaucet();
3       waterLawn();
4     } catch (BrokenPipeException e) {
5       logProblem(e);
6     } finally {
7       stopFaucet();
8     }
```

Exceções

- Criando sua própria exceção:

```
public class MinhaException extends Exception { //checked exception
    public MinhaException () {

    }
    public MinhaException (String msgg) {
        super(msgg); //manda a mensagem para a classe superior. Essa mensagem pode ser recuperada posteriormente.
    }
}
```

```
package br.cefet.sort;

public class PrincipalException {
    public static void main(String[] args) throws Exception{
        throw new MinhaException("excecao inicial de exemplo...codigo de erro:15");
    }

}
```

Markers | Properties | Servers | Data Source Explorer | Snippets | Console | Package Explorer

```
<terminated> PrincipalException [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (06/11/2014 19:32:55)
Exception in thread "main" br.cefet.sort.MinhaException: excecao inicial de exemplo...codigo de erro:15
        at br.cefet.sort.PrincipalException.main(PrincipalException.java:9)
```

Garbage Collector

- A linguagem de programação Java tira de você a responsabilidade de desalocar memória. Ela fornece uma thread que controla cada alocação de memória. Durante os ciclos ociosos da JVM, a thread de garbage collector verifica e libera qualquer memória que pode ser liberada. Caso um objeto não possua referência em memória, ele se torna elegível para ser removido pelo Garbage Collector.