Revisão POO

- **Collections Framework:**
  - Para representar um grupo de objetos de um mesmo tipo temos opção de utilizar um array, porém:
    - não podemos redimensionar um array em Java;
    - é impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
    - não conseguimos saber quantas posições do array já foram populadas sem criar, para isso, métodos auxiliares e percorrer o array inteiro.
  - Além dessas dificuldades que os arrays apresentam, faltava um conjunto robusto de classes para suprir a necessidade de estruturas de dados básicas, como listas ligadas e tabelas de hash, árvores, etc.
  - A API do Collections é robusta e possui diversas classes que representam estruturas de dados avançadas;

Revisão POO

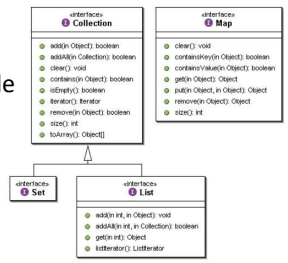
- **Collections Framework:**
  - **java.util.List:**
    - Uma interface implementada por algumas classes concretas;
    - Tem como característica permitir armazenar objetos duplicados e manter uma ordem entre estes objetos armazenados;
    - Implementações:
      - ArrayList: implementada internamente com um array. É mais rápida e econômica computacionalmente porém torna-se ineficiente se a lista for sofrer constantes alterações;
      - LinkedList: usa mais memória que a anterior e é mais lenta para ser percorrida, porém é sempre rápida para inserções e remoções.

Revisão POO

- **Conjuntos java.util.Set:**
  - Semelhante à definição matemática de um conjunto;
  - Contém objetos, sendo que cada objeto só pode aparecer uma vez no conjunto;
  - 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 de□ne como deve ser este comportamento. Tal ordem varia de implementação para implementação.
  - Implementações:
    - HashSet: usa uma tabela de hash. Não mantém ordem nenhuma;
    - TreeSet: usa uma árvore. Coloca os objetos em ordem crescente, para tanto os objetos devem implementar Comparable ou o set deve receber um Comparator para a ordenação;
    - LinkedHashSet: usa uma hash e uma lista ligada para implementação. Mantém a ordem usada na inserção para o posterior percorrimento.

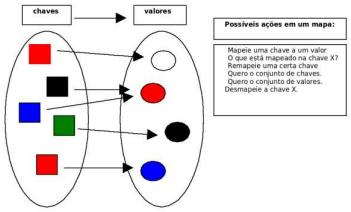
Revisão POO

- **java.util.Collection:**
  - Implementações que não garantem ordem e não se importam com duplicidade implementam apenas Collection;
  - Representa apenas uma coleção de objetos;
  - É a interface pai de Set e List;



Revisão POO

- **java.util.Map:**
  - Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele.
  - Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance.
  - Aqui entra o mapa: um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor.



Revisão POO

- **java.util.Map:**
  - Cada inserção, busca ou remoção de valores no Map deverá fornecer o objeto chave associado àquele valor;

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);
// cria o mapa
Map<String, ContaCorrente> mapaDeContas = new HashMap<>();

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

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

Revisão POO

- **java.util.Map:**
  - Implementações:
    - HashMap: utiliza um hash, não guardando ordem;
    - TreeMap: utiliza árvore, mantendo a mesma ordenada;
    - LinkedHashMap: utiliza lista ligada e tabela de hash. Mantém a ordem de inserção.

Revisão POO

- **equals() e hashCode():**
  - São métodos da classe Object, da qual todas as classes herdam;
  - equals() retorna true na implementação de Object sempre que duas referências forem iguais (mesmo que usar ==);
  - Normalmente pode ser interessante sobrescrever este método. P ex. equals() de Pessoa pode comparar o CPF, etc;
  - Ocorre que muitas das estruturar citadas anteriormente utilizam tabelas de hash para implementação e a tabela de hash utiliza o hashCode() de cada objeto para localizar de forma direta o seu lugar na tabela;
  - Para que as tabelas de hash funcionem **sempre** que equals for sobrescrito hashCode() também precisa ser, para que:
    - **Sempre que dois objetos forem iguais, estes devem possuir o mesmo hashCode();**
  - Dentro de um mesmo hashCode na tabela de hash, vários objetos podem existir porém em uma lista. Isto é indesejável pois deixa as buscas mais demoradas;
    - Dois objetos diferentes podem ter o mesmo hashCode, mas isto deve ser evitado. Um bom algoritmo de geração de hashCode distribui bem a geração de códigos dentro da faixa de precisão dos números inteiros.

Revisão POO

- Exercício 3:
  - Altere o exercício 2 e utilize Map para representar o conjunto de contas existente no banco associado a cada identificador de conta.
  - Desenvolva uma classe que crie 10.000 contas, inserindo uma a uma no banco e depois percorra estas contas;
  - Busque cada uma destas contas com o método get(), enquanto percorre novamente o mapa;
  - Contabilize o tempo gasto para inserir, percorrer o Map e buscar todos os objetos em cada tipo de implementação diferente do Map.

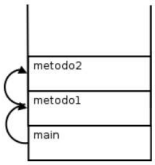
Exceções

- **Exceções:**
  - Em Java, os métodos dizem qual o contrato que eles devem seguir.
  - Se, ao chamar o método, ele não consegue fazer o que deveria, ele precisa avisar ao usuário que a operação não foi feita.
  - Normalmente, e graças ao encapsulamento, quem sabe tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural do que a classe apenas sinalizar que um erro ocorreu.
  - Uma forma de resolver isto (utilizado na programação procedural) é utilizar um código de retorno no método: ***magic numbers!***
    - ***Problema:*** quem chama o método não é obrigado a tratar o magic number, ou seja, se nada for tratado a minha ação equivale a presumir que tudo deu certo, mesmo que tudo tenha dado errado!
  - Um cliente que tenta realizar um saque numa conta sem saldo e sem limite, está infringindo uma regra de negócio, ou seja, está realizando **uma exceção à regra**!

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

Exceções

- **Exceções:**
  - Toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada método. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da pilha de execução (stack): basta remover o marcador que está no topo da pilha:



Exceções

- **Exceções:**
  - Quando uma exceção ocorre em um método e não é tratada pelo método que o chamou (e assim sucessivamente...) a exceção é impressa na saída padrão de erros (System.err)

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at Teste.metodo2(Teste.java:18)
        at Teste.metodo1(Teste.java:10)
        at Teste.main(Teste.java:4)
```

  - Esta visão da exceção, na verdade é o rastro da pilha, ou stacktrace;
    - Pode ser impresso manualmente (exception.printStacktrace());
  - Na primeira linha aparece o arquivo:linha onde a exceção foi gerada, e nas próximas, o arqvuivo:linha onde o método que gerou a exceção foi chamado.
  - Algumas exceções deixam uma mensagem para esclarecer o erro e facilitar o diagnóstico (boa prática).

Exceções

- **Exceções:**
  - Como tratar uma exceção?

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

  - Como gerar uma exceção?
    - Exceções são objetos como qualquer um, porém somente estas podem ser

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


Exceções

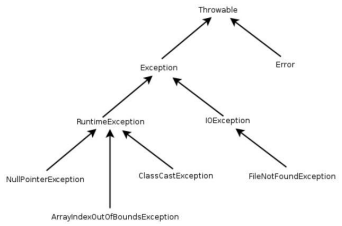
- **Exceções:**
  - Tipos de exceções:
    - **RuntimeException**:
      - São exceções de tempo de execução como acesso fora dos limites de um vetor, ou acesso à uma referência nula, por exemplo;
      - Podem ser tratadas com try-catch, mas o tratamento não é obrigatório;
    - **CheckedException**:
      - São o tipo mais comum de exceções.
      - Sempre devem ser tratadas ou então declaradas na assinatura do método para

```
try {
    objeto.metodoQuePodeLancarIOeSQLException();
} catch (IOException e) {
    // ..
} catch (SQLException e) {
    // ..
}

public void abre(String arquivo) throws IOException, SQLException {
    // ..
}
```


Exceções

- **Exceções:**
  - Tipos de exceções:
    - **Error**:
      - Normalmente são erros internos da JVM;
      - Quando ocorrem não há muito o que fazer...



Exceções

- **Exceções:**
  - Mais sobre exceções:
    - **Bloco try-catch-finally**:

```
try {
    // bloco try
} catch (IOException ex) {
    // bloco catch 1
} catch (SQLException sqlex) {
    // bloco catch 2
} finally {
    // bloco que será sempre executado, independente
    // se houve ou não exception e se ela foi tratada ou não
}
```


Exceções

- **Exceções:**
  - Mais sobre exceções:
    - Mas se sempre podemos tratar a exceção ou atirar para cima (declarar no método), qual opção é a melhor?

      Regra de ouro: se dentro do escopo do objeto eu souber o que fazer no caso daquela exceção devo tratar. Se não souber a responsabilidade deve ser passada para quem me chamou.

Exceções

- **Exercício 4:**
  - Altere as validações das rotinas de saque e depósito do Exercício 2 para a utilização de exceções. Faça também o tratamento adequado destas.

Concorrência

- **Programação Concorrente:**
  - Aspectos sobre concorrência, condições de corrida, semáforos, monitores, etc. já discutidos na disciplina de Sistemas Operacionais.
  - A classe Thread é a representação de uma thread da JVM que será escalonada pela JVM ou pelo SO.

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

        GeraPDF gerapdf = new GeraPDF();
        Thread threadDoPdf = new Thread(gerapdf);
        threadDoPdf.start();

        BarraDeProgresso barraDeProgresso = new BarraDeProgresso();
        Thread threadDaBarra = new Thread(barraDeProgresso);
        threadDaBarra.start();

    }
}
```

```
public class GeraPDF implements Runnable {
    public void run () {
        // lógica para gerar o pdf...
    }
}

public class BarraDeProgresso implements Runnable {
    public void run () {
        // mostra barra de progresso e vai atualizando ela...
    }
}
```


Concorrência

- **Exercício 5:**
  - Construa um algoritmo concorrente para fazer testes nas rotinas desenvolvidas no Exercício 2. Verifique se a concorrência pode causar problemas, identifique e proponha soluções.