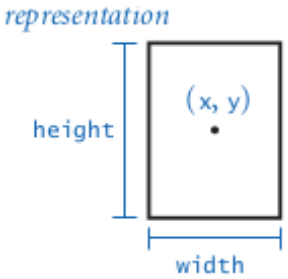
Questões

1. **Retângulos**. A classe `Rectangle` abstrai retângulos alinhados aos eixos cartesianos. Cada retângulo é representado pelo seu ponto central (x, y), sua altura (*height*) e sua largura (*width*), conforme ilustrado na figura. Implemente os métodos listados.



   - `public double area()`

     Retorna o valor da área do retângulo.

   - `public double perimeter()`

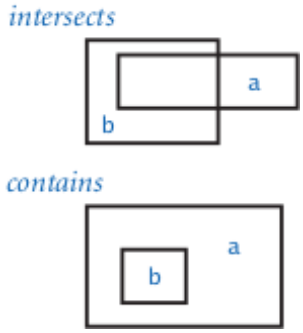
     Retorna o valor do perímetro do retângulo.

   - `public boolean intersects(Rectangle)`

     Determina se o retângulo interseciona com o retângulo passado como argumento.

   - `public boolean contains(Rectangle)`

     Determina se o retângulo contém o retângulo passado como argumento.



   Observações:

   - Considere que dois retângulos sofrem intersecção quando compartilham um ou mais pontos. Com isso, `a.intersects(a)` e `a.contains(a)` são sempre `true`.
   - Na pasta `src/test/resources/rectangles` você pode visualizar as imagens dos retângulos que foram usados nos testes. O nome do arquivo coincide com o nome do caso (`caseX`).

2. **Inteiros grandes**. Os tipos primitivos inteiros em Java têm precisão pré-definida, sendo 32 bits para `int` e 64 bits para `long`, o que permite representar, no máximo, os inteiros $2^{31}-1$ e $2^{63}-1$, respectivamente. O mesmo vale para as classes encapsuladoras `Integer` e `Long`. Se quisermos trabalhar com números maiores, precisamos definir tipos de dados adicionais.

   A classe `LargeInteger` representa inteiros *positivos* (i.e. números naturais) como um array de caracteres, sendo que cada caractere representa um dígito. Por exemplo, o inteiro `123456` é representado pelo array de caracteres `{'1','2','3','4','5','6'}`. Ao utilizarmos arrays para armazenar dígitos, podemos representar inteiros arbitrariamente grandes, com tamanho limitado apenas pela quantidade de memória do ambiente de execução[1]. Defina os seguintes métodos na classe `LargeInteger`:

   - `public LargeInteger()`

     Cria uma instância com valor zero.

   - `public LargeInteger(long)`

     Cria uma instância com o mesmo valor do parâmetro (tipo `long`).

   - `public LargeInteger(LargeInteger)`

     Cria uma instância com o mesmo valor de outra instância.

- `public LargeInteger(String)`
  Cria uma instância com os mesmos dígitos da `String` passada como argumento.
  Dispara uma `IllegalArgumentException` se o argumento possui caracteres que não são dígitos.
- `public int size()`
  Informa quantos digitos o inteiro possui.
- `public compareTo(LargeInteger)`
  Compara duas instâncias, retornando: um número positivo, caso a instância atual seja maior que o parâmetro; um número negativo, caso a instância atual seja menor que o parâmetro; zero, caso ambas instâncias tenham o mesmo valor.
- `public LargeInteger plus(LargeInteger)`
  Adiciona duas instâncias (a atual e o parâmetro), retornando uma terceira instância como resultado da soma.
- `public LargeInteger minus(LargeInteger)`
  Subtrai duas instâncias (a atual e o parâmetro), retornando uma terceira instância como resultado da subtração. Como `LargeInteger` só define números naturais (i.e. não há negativos), considere como resultado a magnitude (valor absoluto) da diferença (por exemplo, $|10 - 4| = |4 - 10| = 6$).
  Dica: há vários algoritmos para subtração, um método simples de implementar computacionalmente é o método dos complementos[2].
- `public LargeInteger ninesComplement(LargeInteger)`
  Gera um `LargeInteger` correspondente ao complemento de nove do parâmetro. O complemento de nove consiste em subtrair cada dígito de 9. Por exemplo, o complemento de nove do número 1234 é o número 8765.
- `public String toString()`
  Gera a representação da instância como `String` (sem zeros à esquerda, caso haja).

Importante. Não é permitido usar `java.math.BigInteger` ou similar.

3. **Modelagem de software orientado a objetos**. Uma empresa contratou você e sua equipe para desenvolver um sistema que permita monitorar os pontos de acesso (*access points*) da rede. Você ficou responsável por construir o modelo de dados desse sistema utilizando orientação a objetos. Você precisa desenvolver:

   - Um conjunto de classes que modele adequadamente os requisitos.
     Armazenar todas as classes que você definir dentro do pacote `accesspoints`.
   - Um código-cliente (classe com método `main`) que teste adequadamente as APIs das classes.

   Requisitos do modelo:

   A empresa está distribuída em múltiplos prédios. Cada prédio tem uma quantidade de andares, sendo que cada andar tem uma quantidade específica de salas (i.e. alguns andares podem ter mais salas que outros). Cada sala pode ter uma

quantidade de access points. Em relação a cada access point, importa saber qual é o seu estado (ligado ou desligado) e, caso esteja ligado, em que mês e ano foi ligado pela última vez.

Requisitos da API das classes:

Um código-cliente que utilize as classes deve ser capaz de efetuar as seguintes consultas: instanciar objetos de todas as classes, com construtores adequados; determinar quantos andares há em um prédio; determinar quantas salas há em um andar; determinar quantos pontos de acesso há em uma sala; determinar se um ponto de acesso está ligado ou desligado; determinar em que mês e ano um access point foi ligado pela última vez. Além disso, o código cliente pode ligar ou desligar um access point, sendo que a data (mês e ano) dessa operação deve ser registrada automaticamente.

Observações sobre a avaliação:

- A questão será avaliada manualmente.
- Atenção aos critérios de avaliação estabelecidos no plano de ensino.
- Projetar as APIs das classes com atenção ao princípio de encapsulamento (ocultamento da informação), permitindo o acesso externo às informações somente quando estritamente necessário.
- Faça uma leitura sobre o *Single Responsibility Principle*[3] (Princípios de Responsabilidade Única), garantindo que está sendo aplicado em sua solução.

[1]Esta abordagem claramente não é a mais eficiente do ponto de vista do consumo de memória, mas adotamos esta simplificação para fins didáticos.

[2]https://en.wikipedia.org/wiki/Method_of_complements

[3]https://en.wikipedia.org/wiki/Single-responsibility_principle