Universidade Federal de Minas Gerais
Ciência da Computação

Linguagens de Programação - Haniel Barbosa

Lista de Exercícios 7

1. Considere a classe *Staff* abaixo, implementada em Python:

```
class Staff:
  payroll = {}
  def getSalary(self, name):
    if self.payroll.has_key(name):
      return self.payroll[name]
    else:
      return 0.0
  def addEmp(self, name, salary):
    self.payroll[name] = salary
  def raiseSalary(self, name, salary):
    self.payroll[name] = self.payroll[name] + salary
```

   (a) O método *getSalary* utiliza um valor especial, 0.0, como o salário de um empregado inexistente. Qual a desvantagem desta forma de tratamento de erros?

   Solução: Não é possível diferenciar se um funcionário tem salário 0.0 ou não é um funcionário.

   (b) Crie uma classe de exceções *NonExistentEmployee* para tratar a situação excepcional de uma busca sobre um empregado inexistente.

   (c) Modifique o método *getSalary* para disparar uma exceção do tipo *NonExistentEmployee* caso o nome solicitado não possua uma entrada no banco de dados.

   (d) Agora crie uma outra classe de exceção, chamada *NegativeSalary*, para lidar com tentativas de criar empregados com salarário negativo. Instâncias desta classe devem possuir dois atributos, *name* e *salary*, que representam, respectivamente, o nome do empregado e o salário que deram origem à exceção.

   (e) Modifique o método *addEmp* para disparar uma exceção do tipo *NegativeSalary*, caso seja feita a tentativa de adicionar um empregado cujo salário seja negativo.

   (f) Modifique o método *raiseSalary* para tratar tanto erros de empregado não existente quanto erros de salário negativo.

(g) Considere o método *readEmployees* abaixo. Reescreva este método para que ele implemente o tratamento de erro. Cuide para que sua nova implementação não termine o programa logo que uma exceção for disparada. Isto é, tendo inserido um nome inválido, o usuário deveria ter uma nova chance de informar um nome. O mesmo vale para o salário negativo.

```
def readEmployees(s):
  name = raw_input("Please, enter a name (Press RETURN to
     finish) ")
  while name != '':
    salary = float(raw_input("Please, enter the salary: "))
    s.raiseSalary(name, salary)
    name = raw_input("Please, enter a name (Press RETURN to
       finish) ")
```

2. Cada uma das questões a seguir diz respeito ao programa abaixo:

```
class Wrapper
{
 private:
  int* d_o;

 public:
  Wrapper() { d_o = nullptr; }
  Wrapper(int o) { d_o = new int(o); }
  int get() { return *d_o; }
  ~Wrapper() { delete d_o; }
};

int main()
{
  std::unique_ptr<Wrapper> w(new Wrapper());
  std::cout << w->get() << "\n";
}
```

(a) Que tipo de erro será causado pelo programa acima?

(b) Defina uma classe de exceção para representar este erro.

(c) Modifique o método *get* para disparar a exceção criada anteriormente.

(d) Modifique o método *main* para tratar esta exceção.

3. Java é uma (cada vez menos...) popular linguagem de programação com forte foco em tratamento de exceções. Em particular, ela provê funcionalidades ausentes em outras linguagens com tratamento de exceções, como Python e C++. Um exemplo são exceções verificáveis estaticamente. Estas exceções precisam ser tratadas explicitamente pelo desenvolvedor. Por exemplo, o código Java abaixo não compila a não ser que cláusulas *try/catch* apropriadas sejam utilizadas:

```
import java.io.*;

public class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

   (a) Quais exceções são verificadas estaticamente? Isto é, como podemos declarar exceções assim?

   (b) Cite uma vantagem desta abordagem adotada por Java.

   (c) Existem exceções, como por exemplo *ClassCastException*, que não são verificadas estaticamente. Por que Java provê exceções assim?

   (d) Java é a única linguagem muito (porém cada vez menos...) popular que adota exceções verificáveis estaticamente. Obviamente existem desvantagens nesta abordagem, pois linguagens posteriores a Java não a seguiram. Cite uma desvantagem das exceções verificadas estaticamente.

4. A passagem de parâmetros por expansão de macros é um mecanismo bastante utilizado em C. Um exemplo é dado abaixo:

```c
#define SUM(X, Y) (X) + (Y)

int main(int argc, char** argv)
{
  printf("sum = %d\n", SUM(argc, argv[0][0]));
}
```

   (a) Em C, macros são expandidas por um componente do compilador chamado pré-processador. Escreva o código do programa acima, após o pré-processamento.

   (b) Um dos problemas com expansão de macros é a chamada captura de variáveis. Ilustre este problema com um exemplo.

   (c) Um outro problema é a múltipla avaliação de parâmetros, isto é, o fato de parâmetros serem avaliados múltiplas vezes pode dar ao programa um significado diferente da intenção do programador. Escreva um exemplo em C que demonstre que parâmetros de macros são avaliados múltiplas vezes.

5. Consider a classe abaixo, implementada em Java:

```
class MyInt {
  int i;
  MyInt(int k) {
    i = k;
  }
  void swap1(MyInt j) {
    MyInt tmp = j;
    j = new MyInt(i);
    i = tmp.i;
  }
  void swap2(MyInt j) {
    MyInt tmp = j;
    j.i = i;
    i = tmp.i;
  }
  void swap3(int j) {
    int tmp = j;
    j = i;
    i = tmp;
  }
}
```

Cada uma das próximas questões é completamente independente uma das outras. Estas questões devem ser respondidas com base nas definições abaixo:

```
MyInt m1 = new MyInt(3);
MyInt m2 = new MyInt(4);
```

(a) Qual é o valor de m1.i e m2.i depois da chamada m1.swap1(m2)?

(b) Qual é o valor de m1.i e m2.i depois da chamada m1.swap2(m2)?

(c) Qual é o valor de m1.i e m2.i depois da chamada m1.swap3(m2)?

(d) Qual é o tipo de passagem da parâmetros que Java adota para tipos primitivos (*int*, *float*, *char*, etc)?

(e) Qual é o tipo de passagem da parâmetros que Java adota para objetos?

6. Escreva duas funções ML, f e g, que demonstrem que ML não faz passagem de parâmetros por expansão de macros. A função f deve chamar apenas a função g, e a função g não deve chamar nenhuma outra função. Explique os resultados que a chamada por expansão de macros produziria, e mostre os resultados que ML efetivamente produz.

7. Um dos mecanismos de passagem de parâmetros chama-se passagem por nome. Na passagem por nome os parâmetros não são avaliados imediatamente. A avaliação acontece quando estes parâmetros são utilizados. Um detalhe importante, contudo, é que a avaliação acontece em que a função foi chamada, não no contexto em que ela é executada. Isto faz com que não ocorra o problema da captura de variáveis, como ocorre em expansão de macros. A ideia de passagem por nomes foi lançada em Algol, e viu uso também em Simula, a linguagem precursora das linguagens orientadas por objeto. Por outro lado, este mecanismo acabou abandonado, por ser difícil de ser implementado e complicar o código dos programas. Porém é um mecanismo que permite soluções engenhosas para certos problemas. Considere, por exemplo, a função abaixo, escrito em Simula, e responda a pergunta a seguir:

```
Real Procedure Sigma (k, m, n, u);
Name k, u;
Integer k, m, n; Real u;
Begin
Real s:=0;
k:= m;
While k <= n Do Begin s:= s + u; k:= k + 1; End;
Sigma:= s;
End;
```

Qual o valor de Z na chamada abaixo? Por quê?

```
Z:= Sigma (i, 1, 4, i ** 2);
```