# Computação Orientada a Objetos

Arquivos – continuação

***Profa. Thienne Johnson***

***EACH/USP***

# Conteúdo

- Java, como programar, 6ª edição
  - Deitel & Deitel
  - Capítulo 14

# Serialização

# E/S de alto nível

- Serialização de objetos
  - Lê e escreve objetos inteiros em arquivo;
  - Arquivo em formato binário.

# Arquivos de objetos

- Classe de dados deve implementar a interface **`Serializable`**
- A leitura (**`ObjectInputStream`**) e escrita (**`ObjectOutputStream`**) serial de objetos são filtros acoplados a um fluxo principal
- Permitem a leitura e escrita de objetos inteiros, incluindo suas referências a outros objetos
- Inclui suporte a tipos **`Collection`**

# Arquivos de objetos

- Serialização de objetos: mecanismo para ler ou gravar um objeto inteiro a partir de um arquivo
  - realizada com fluxos de bytes
- Objeto serializado: representado como uma sequência de bytes que inclui:
  - os dados do objeto
  - as informações sobre o tipo do objeto
  - os tipos dos dados armazenados no objeto

# Abertura do fluxo

- Associando um arquivo

```
File arquivo =   new File("meusobjetos.bin");
```

- Para leitura

```
ObjectInputStream entrada =
  new ObjectInputStream(
  new FileInputStream(arquivo));
```

- Para escrita

```
ObjectOutputStream saida =
  new ObjectOutputStream(
  new FileOutputStream(arquivo));
```

# Leitura, escrita e fechamento

- Leitura de dados (pode gerar *EOFException*)

```
objeto = ( Tipo ) entrada.readObject();
```

- Escrita de dados

```
saida.writeObject(objeto);
```

- Fechamento do arquivo

```
entrada.close();
saida.close();
```

# Exemplo

- Considere uma classe representando registros de itens em um estoque (nome do produto, quantidade e valor);

```
import java.io.Serializable;
public class Produto implements Serializable {
        private String nome;
        private int unidades;  // estoque em unidades
        private float custo;   // custo unitário
        public Produto(){
            this(" ", 0 , 0.0);
        }  ...}
```

# Exemplo

- Escrevendo em um fluxo de objetos

```
...
Produto item = new Produto(“livro java”, 10, 148.50);
try {
  FileOutputStream arq = new FileOutputStream(“item.dat");
  ObjectOutputStream objarq = new ObjectOutputStream(arq);
  objarq.writeObject(item);
  objarq.close();
}
catch(IOException e) {
  System.out.println(e.getMessage());
  e.printStackTrace();
}
...
```

# Exemplo

- Lendo a partir de um fluxo de objetos

```
...
Produto item1 = new Produto();
try {
  FileInputStream arq = new FileInputStream("item.dat");
  ObjectInputStream objarq = new ObjectInputStream(arq);
  item1 = (Produto) objarq.readObject();
  objarq.close();
}
catch(IOException e) {
  System.out.println(e.getMessage());
  e.printStackTrace();
}
...
```

# Arquivos de coleções

- Estruturas de dados do tipo *Collection* podem ser lidas ou escritas na sua totalidade sem necessidade de iteração

```
Set <Integer> s = new HashSet <Integer> ();

...
ObjectOutputStream saida =
  new ObjectOutputStream(
  new FileOutputStream(arquivo));

saida.writeObject(s);
```

# Arquivos de coleções

- Leitura

```
Set <Integer> s = new HashSet <Integer> ();

...
ObjectInputStream entrada =
  new ObjectInputStream(
  new FileInputStream(arquivo));

s = (Set)entrada.readObject();
```

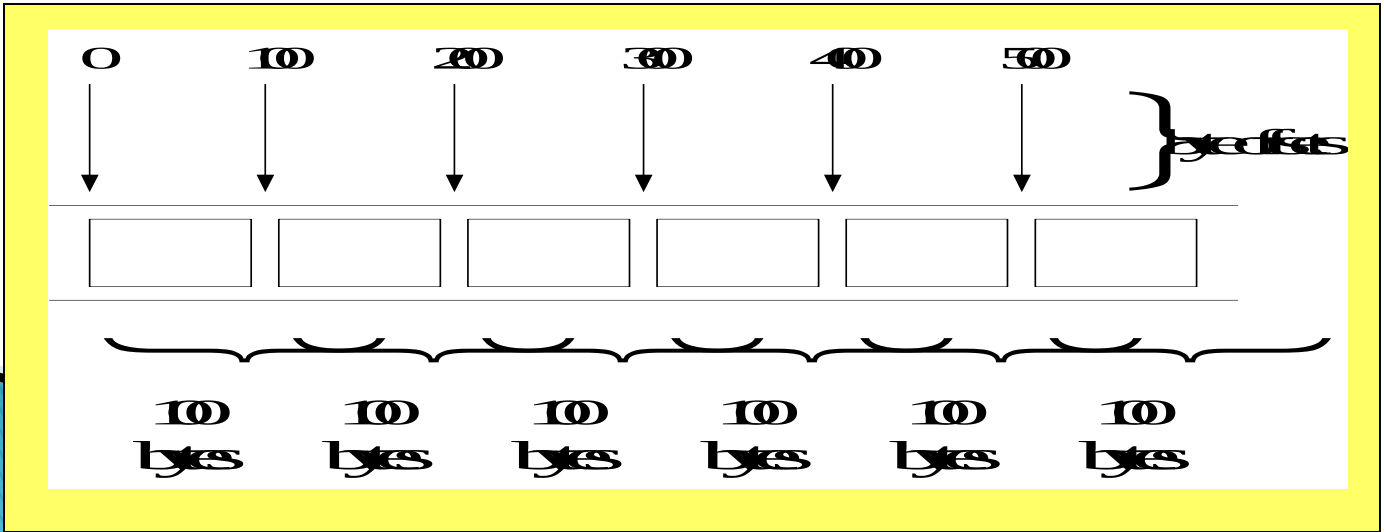
# Arquivos de Acesso Aleatório

# Arquivos de Acesso Aleatório

- Permitem ler ou escrever a partir de qualquer posição no arquivo
  - acesso rápido
- Podem ser criados utilizando a classe **`RandomAccessFile`**
  - permite que se trabalhe nos modos “leitura” (**`r`**), “gravação” (**`w`**) ou “leitura e gravação” (**`rw`**)

# Arquivos de Acesso Aleatório

- Podem ser organizados de forma simples usando registros de tamanho fixo
  - Facilita o cálculo da localização exata de qualquer registro em relação ao início do arquivo.



# Arquivos de Acesso Aleatório

- Os registros podem ser acessados diretamente, por meio de um ponteiro (ponteiro de arquivo)
- Inserção de dados não destrói outros registros
- Dados previamente armazenados podem ser atualizados sem sobreescrever outros

# Arquivos de Acesso Aleatório

- Ao associar um RandomAccessFile a um arquivo
  - Dados são lidos/escritos na posição do ponteiro de arquivo
  - Todos os dados são tratados como tipos primitivos(i.e., formato binário)
    - int: 4 bytes;
    - double: 8 bytes;
    - etc

# O ponteiro de arquivo

- Leitura e escrita ocorrem na posição do *ponteiro de arquivo* (iniciada em zero)

- Posição atual do ponteiro de arquivo:

```
long posicao = arq.getFilePointer();
```

- Deslocamento para posição específica:

```
arq.seek(posicao);
```

- Avanço de *n* posições:

```
arq.skipBytes(n);
```

# Arquivos de Acesso Aleatório

- Abertura do arquivo
  - Para leitura

```
RandomAccessFile entrada = new
  RandomAccessFile("dados.bin","r");
```

  - Para escrita

```
RandomAccessFile entrada = new
  RandomAccessFile("dados.bin","w");
```

  - Para leitura e escrita

```
RandomAccessFile saida = new
  RandomAccessFile("dados.bin","rw");
```

# Arquivos de Acesso Aleatório

- Leitura (pode gerar *EOFException*)

```
char c = entrada.readChar();
int i = entrada.readInt();
double d = entrada.readDouble();
```

- Escrita

```
saida.writeChar(c);
saida.writeChars(s);
saida.writeInt(i);
saida.writeDouble(d);
```

- Fechamento

```
entrada.close();
saida.close();
```

Strings devem ser construídas com **`readChar`**

# Exemplo: registros bancários

- Considere uma classe representando registros bancários (nro.de conta corrente, nome e saldo)
- Deseja-se armazenar 100 registros em um arquivo de acesso aleatório:
  - contas numeradas de 00 a 99
  - inclusão e atualização direta

# Exemplo: registros bancários

- A classe Registro

```
public class Registro {
    int nroconta;
    String nome;
    double saldo;
}
```

# Exemplo: registros bancários

- Escrita da classe Registro

Array de objetos Registro

```
RandomAccessFile saida = new RandomAccessFile("dados.bin","rw");
for (int i = 0; i < clientes.length; i ++) {
    saida.writeInt(i);
    saida.writeChars(clientes.nome[i]);
    saida.writeDouble(clientes.saldo[i]);
}
saida.close();
```

Uma alternativa melhor: definir um método de escrita para a classe Registro

# Exemplo: registros bancários

- Busca de um registro

```
// assume o arquivo saida já aberto
// cc é o número da conta a localizar

saida.seek( (cc) * Registro.tamanho());

// a classe Registro implementa o método tamanho, que
// retorna uma constante inteira representando o número
// de bytes que ocupa
```

# O tamanho de um registro

- Soma dos tamanhos de seus campos:
  - Tipo *Char* possui tamanho 2
  - Tipo *String* possui tamanho pré-definido pelo programador (nro. caracteres * 2)
  - Tipos *Integer*, *Double* etc têm seu tamanho (em bits) dado por **`SIZE`**:

```
TamanhoInteiro = Integer.SIZE / 8;
```

# Exemplo: registros bancários

- Método **`tamanho()`** da classe Registro:

```
public int tamanho () {
    return (
        // string de 15 chars.
        (Integer.SIZE/8) + 15 * 2 + (Double.SIZE/8));
        // ou simplesmente return(4+30+8);
}
```

# Exemplo: registros bancários

- Uma vez localizado o registro desejado, este pode ser reescrito com os métodos ***`writeInt`***, ***`writeChar`*** , etc.

- Prática comum: criar na própria classe Registro um método **`escrever(arquivo)`** para escrever um registro na posição atual do arquivo especificado

# Exemplo: registros bancários

- Escrita de um registro:

```
public void escrever(RandomAccessFile f)
    throws IOException {
      f.writeInt(nroconta);
      StringBuffer b = new StringBuffer(nome);
      b.setLength( 15 );
      f.writeChars( b.toString() );
      f.writeDouble(saldo);
}
```

Cria e manipula strings modificáveis

# Exemplo: registros bancários

- Leitura de um registro:

```
public void ler(RandomAccessFile f) throws IOException {
    nroconta = f.readInt();
    char letras[] = new char [15];
    for(int i=0;i<15;i++)
       letras[i] = f.readChar();
    nome = new String( letras ).replace( '\0', ' ' );
    saldo = f.readDouble();
}
```