Suponha que o arquivo birds.dat exista, seja acessível e contenha dados para um objeto Bird. Qual é o resultado da execução do seguinte código? (Escolha todas as opções corretas.)

```
1:  import java.io.*;
2:  public class Bird {
3:  private String name;
4:  private transient Integer age;
5:
6:  // Getters/setters omitidos
7:
8:  public static void main(String[] args) {
9:      try(var is = new ObjectInputStream(
10:             new BufferedInputStream(
11:             new FileInputStream("birds.dat")))) {
12:                     Bird b = is.readObject();
13:                     System.out.println(b.age);
14: } } }
```

A. Compila e imprime 0 em tempo de execução.
B. Compila e imprime null em tempo de execução.
C. Compila e imprime um número em tempo de execução.
D. O código não compila por causa das linhas 9-11.
E. O código não compila por causa da linha 12.
F. Compila, mas lança uma exceção em tempo de execução.

ANÁLISE DA QUESTÃO

Primeiro vamos entender passo a passo o que o código está fazendo:

- Linhas 1-2: O código importa as classes necessárias para a manipulação de arquivos e define a classe *Bird*.
- Linha 3: Declara o campo name da classe *Bird*, que é uma String.
- Linha 4: Declara o campo age da classe *Bird*, que é um *Integer* e é marcado como *transient*. A palavra-chave *transient* indica que este campo não deve ser serializado.
- Linha 6: Comentário indicando que os getters e setters foram omitidos.
- Linhas 8-14: O método *main* é o ponto de entrada do programa.
- Linhas 9-11: Um *ObjectInputStream* é criado para ler objetos a partir do arquivo "birds.dat". Ele está encadeado com um *BufferedInputStream* e um *FileInputStream*.
- Linha 12: Tenta ler um objeto do arquivo usando o método *readObject* e o atribui a uma variável *Bird* chamada *b*.
- Linha 13: Imprime o valor do campo *age* do objeto *Bird*.

Agora vamos analisar o que estaria errado:

1. **Exceções não Tratadas:**

   - *FileNotFoundException*: Gerada pelo *FileInputStream* na linha 11.
   - *IOException*: Pode ser gerada por qualquer operação de entrada/saída nas linhas 10 e 12.
   - *ClassNotFoundException*: Pode ser gerada pela chamada a *readObject()* na linha 12.

Todas as exeções acima são *checked exceptions* (exceções verificadas) e são exceções que são **verificadas pelo compilador em tempo de compilação**. Isso significa **que o compilador exige que o código lide com essas exceções**, garantindo que o programa possa responder a erros de maneira controlada.

Por que precisam ser tratadas ou declaradas?

A necessidade de tratar ou declarar checked exceptions tem a ver com a robustez e a confiabilidade do código. Aqui estão as razões principais:

- **Prevenção de Falhas:** Checked exceptions geralmente representam condições que o programa deve esperar e lidar de alguma maneira. Por exemplo, ao tentar abrir um arquivo que pode não existir, é razoável esperar que essa operação possa falhar e o código deve estar preparado para isso.
- **Claridade de Código:** Declarar exceções na assinatura do método (throws) torna explícito quais exceções um método pode lançar, facilitando a compreensão e o uso correto dos métodos.
- **Manutenção:** Quando um método declara exceções que pode lançar, qualquer código que chama esse método é forçado a lidar com essas exceções. Isso ajuda a garantir que todas as possíveis falhas sejam consideradas e tratadas adequadamente, facilitando a manutenção do código a longo prazo.

Como Tratar ou Declarar Checked Exceptions?

Existem duas maneiras de lidar com checked exceptions em Java:

- **Tratamento com bloco try-catch:** Envolva o código que pode lançar uma exceção com um bloco *try* e capture a exceção com *catch*.

```
try {
    // Código que pode lançar uma exceção
} catch (IOException e) {
    // Código para tratar a exceção
    e.printStackTrace();
}
```

- **Declaração com throws:** Declare que o método pode lançar uma exceção específica na assinatura do método.

```
public void meuMetodo() throws IOException {
    // Código que pode lançar IOException
}
```

Aqui está um exemplo completo para ilustrar:

```
import java.io.*;

public class ExemploCheckedException {
    public static void main(String[] args) {
        try {
            lerArquivo("arquivo_inexistente.txt");
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao tentar ler o arquivo: " + e.getMessage());
        }
    }

    public static void lerArquivo(String nomeArquivo) throws IOException {
        BufferedReader leitor = new BufferedReader(new FileReader(nomeArquivo));
        String linha;
        while ((linha = leitor.readLine()) != null) {
            System.out.println(linha);
        }
        leitor.close();
    }
}
```

Neste exemplo:

- método *lerArquivo* declara que pode lançar uma *IOException*.
- método *main* usa um bloco *try-catch* para lidar com essa exceção, garantindo que o programa possa tratar a situação em que o arquivo não existe.

2. Casting Necessário::

- método *readObject()* retorna um *Object*, então precisamos fazer o **casting explícito** para Bird.

O *casting* em Java é o processo de converter um tipo de dado em outro. Isso é necessário quando você precisa usar um tipo de dado específico, mas atualmente possui um tipo diferente. O *casting* pode ser implícito ou explícito. Vamos entender cada um desses conceitos.

Casting Implícito (Implicit Casting)

O *casting* implícito, também conhecido como conversão automática, acontece quando o compilador Java automaticamente converte um tipo de dado menor para um tipo de dado maior sem que o programador precise fazer nada explicitamente. Isso é seguro e não há perda de dados.

Exemplos de Casting Implícito:

1. Conversão de *integer* para *double*:

```
int numInt = 10;
double numDouble = numInt; // Conversão implícita de int para double
```

2. Conversão de *char* para *integer*:

```
char letra = 'A';
int codigoAscii = letra; // Conversão implícita de char para int
```

Aqui, o *char letra* é convertido implicitamente para o código ASCII correspondente em *int*.

Casting Explícito (Explicit casting)

O *casting* explícito, também conhecido como conversão manual, **é quando o programador precisa especificar a conversão de um tipo de dado maior para um tipo de dado menor**, ou de um **tipo de objeto para um tipo mais específico**. Isso é necessário porque pode haver perda de dados ou a necessidade de uma verificação de tipo.

Exemplos de Casting Explícito:

1. Conversão de duble para int:

```
double numDouble = 10.5;
int numInt = (int) numDouble; // Conversão explícita de double para int
```

Nesse caso, o *double numDouble* é convertido explicitamente para *int numInt*. Note que a parte decimal (.5) será perdida na conversão.

1. Conversão de superclasse para subclasse::

```
Object obj = "Hello, World!";
String str = (String) obj; // Conversão explícita de Object para String
```

Aqui, um objeto *Object* que realmente contém uma *String* é convertido explicitamente para *String*. O compilador precisa do *casting* explícito para saber que o programador está ciente do tipo específico do objeto.

Por Que o Casting Explícito é Necessário?

O *casting* explícito é necessário porque **nem todas as conversões são seguras e automáticas**. Por exemplo, converter de um tipo mais específico para um mais genérico é seguro (como de int para double), mas converter de um tipo genérico para um mais específico pode ser perigoso (como de Object para String). **O *casting* explícito garante que o programador reconheça e aceite o risco potencial envolvido na conversão.**

Resumindo a análise desta questão, dois problemas distintos impedem a compilação do código:

D. O código não compila por causa das linhas 9-11.

Correto. A linha 10 pode lançar *IOException*, a linha 11 pode lançar *FileNotFoundException*, e ambas são exceções verificadas que precisam ser tratadas ou declaradas.

E. O código não compila por causa da linha 12.

Correto. *readObject()* retorna um *Object* e precisa ser feito um *casting* explícito para *Bird*. Além disso, *readObject()* pode lançar *IOException* e *ClassNotFoundException*, que também precisam ser tratadas ou declaradas.

Portanto, as opções corretas são: **D e E**

Uma versão possível do código corrigido seria:

```
1:  import java.io.*;
2:  public class Bird implements Serializable { // Supondo que Bird agora implementa Serializable
3:  private String name;
4:  private transient Integer age;
5:
6:  // Getters/setters omitted
7:
8:  public static void main(String[] args) throws IOException, ClassNotFoundException {
9:      try(var is = new ObjectInputStream(
10:             new BufferedInputStream(
11:             new FileInputStream("birds.dat")))) {
12:                     Bird b = (Bird) is.readObject(); // Cast explícito para Bird
13:                     System.out.println(b.age);
14: } } }
```

A classe Bird agora implementa *Serializable*, o que permite que objetos *Bird* sejam serializados e desserializados. Apesar de que se não for extendida não gera erro de compilação.

Uma observação, como o campo *age* é marcado como transient,significa que ele não será serializado. Durante a desserialização, o campo transient não terá seu valor restaurado do arquivo; ele será configurado para seu valor padrão que no caso será *null*.

Note também que a solução proposta está utilizando a abordagem de propagar as exceções. O método main está declarado com *throws IOException, ClassNotFoundException*, o que significa que ele propaga essas exceções para o chamador (neste caso, a JVM), em vez de tratá-las localmente com um bloco catch.

Outra abordagem é capturar as exceções localmente, você pode adicionar um bloco *catch* para tratar as exceções dentro do método *main*. Aqui está uma versão modificada do código que inclui blocos catch:

```
1:  import java.io.*;
2:  public class Bird implements Serializable {
3:  private String name;
4:  private transient Integer age;
5:
6:  // Getters/setters omitidos
7:
8:  public static void main(String[] args) {
9:      try(var is = new ObjectInputStream(
10:             new BufferedInputStream(
11:             new FileInputStream("birds.dat")))) {
12:                     Bird b = (Bird) is.readObject();
13:                     System.out.println(b.age);
14: }   catch (IOException | ClassNotFoundException e) {
15:             e.printStackTrace();
16: } } }
```