Qual dos seguintes tipos pode ser inserido no espaço em branco para permitir que o programa compilar com sucesso? (Escolha todas as opções aplicáveis.)

```
1: import java.util.*;
2: final class Amphibian {}
3: abstract class Tadpole extends Amphibian {}
4: public class FindAllTadpoles {
5: public static void main(String… args) {
6: var tadpoles = new ArrayList<Tadpole>();
7: for (var amphibian : tadpoles) {
8: ___________ tadpole = amphibian;
9: } } }
```

A. `List<Tadpole>`
B. `Boolean`
C. `Amphibian`
D. `Tadpole`
E. `Object`
F. Nenhuma das opções acima.

BOYARSKY, Jeanne; SELIKOFF, Scott. **OCP Oracle® Certified Professional Java SE 17 Developer Study Guide**. John Wiley & Sons, 2022.

ANÁLISE DA QUESTÃO

A. `List<Tadpole>`

Isso não faz sentido no contexto porque `Amphibian` é um objeto e não uma lista. Então, não compilaria.

B. `Boolean`

`Amphibian` é um objeto da classe Tadpole, não um Boolean. Então, isso não compilaria.

C. `Amphibian`

Isso funcionaria. MAS, CONTUDO, ENTRETANTO, TODAVIA só funcionaria se a classe `Amphibian` **não fosse** `final`. Uma classe `final` não pode ser estendida para criar uma subclasse. Todos os métodos em uma classe final são implicitamente finais. A classe `String` é um exemplo de classe `final`. Por exemplo, se tivermos `String a = "Bom";` e `String b = "Dia";` e fizermos `a = a + b;` um novo objeto será instanciado para atribuir a soma destas duas strings, você pode verificar por si mesmo, usando o método estático

`System.identityHashCode()` e perceberá que o hash code de `a` vai ser diferente.

D. `Tadpole`

Funcionaria se não fosse a clásula `final`.

E. `Object`

Funcionaria se não fosse a clásula `final`.

RESPOSTA: Alternativa **F**. O código, como está, vai dar o seguinte erro de compilação (no Eclipse):

The type Tadpole cannot subclass the final class Amphibian

Agora vamos supor que não existisse a cláusula `final` no código, as opções válidas seriam a C,D ou E devido ao polimorfismo. Vamos explicar melhor.

Código Modificado

```
1: import java.util.*;
2: class Amphibian {}
3: abstract class Tadpole extends Amphibian {}
4: public class FindAllTadpoles {
5: public static void main(String… args) {
6: var tadpoles = new ArrayList<Tadpole>();
7: for (var amphibian : tadpoles) {
8: ___________ tadpole = amphibian;
9: } } }
```

Opção C: Amphibian

```
Amphibian tadpole = amphibian;
```

Explicação:

- `Tadpole` estende `Amphibian`, o que significa que todo objeto `Tadpole` é um `Amphibian` (herança).
- `Polimorfismo` permite que um objeto da subclasse (`Tadpole`) seja tratado como um objeto da superclasse (`Amphibian`).

- Portanto, é válido atribuir `amphibian` (do tipo `Tadpole`) a uma variável do tipo `Amphibian`.

Opção D: Tadpole

```
Tadpole tadpole = amphibian;
```

Explicação:

- `amphibian` é do tipo `Tadpole`, e estamos atribuindo a uma variável do tipo `Tadpole`.
- Não há necessidade de casting aqui, porque a variável `amphibian` já é do tipo `Tadpole`.
- Portanto, essa atribuição é diretamente válida.

Opção E: Object

```
Object tadpole = amphibian;
```

Explicação:

- Em Java, `Object` é a superclasse de todas as classes.
- Todo objeto em Java é uma instância de Object, incluindo `Tadpole`.
- Polimorfismo permite que um objeto de qualquer classe seja tratado como um `Object`.
- Portanto, atribuir `amphibian` (do tipo `Tadpole`) a uma variável do tipo `Object` é válido.

O polimorfismo é um dos pilares da programação orientada a objetos e permite que um objeto de uma subclasse seja tratado como um objeto de qualquer uma de suas superclasses. Isso é útil para escrever código mais flexível e reutilizável.

Aqui está um exemplo:

```
// classe base
class B {
    void metodoB() {
       System.out.println("Método em B");
    }
}

// subclasse de classe B
class A extends B {
    void metodoA() {
       System.out.println("Método em A");
    }
}
```

```
public class ExemploPolimorfismo {

    public static void main(String[] args) {
       B var_b = new A(); // Isso é válido
       var_b.metodoB();   // Isso chama o método da classe B

       // Mas var_b não pode acessar metodoA() diretamente
       //var_b.metodoA(); // Isso daria erro de compilação
    }

}
```

Aqui, `var_b` é uma variável do tipo `B`, mas pode armazenar uma instância de `A` devido ao polimorfismo, **mas perceba que `var_b` não pode chamar um método específico da classe `A`** pois a classe `B` não possui o método desta subclasse.

A seguir estão alguns exemplos contrários, onde podem haver erros de compilação quando a declaração de uma variável é de um tipo e a instanciação de um objeto de outro tipo, e o porque do erro.

1. **Incompatibilidade Direta:**

```
class Animal {
    void fazerBarulho() {
        System.out.println("Barulho genérico de animal");
    }
}

class Cachorro extends Animal {
    void fazerBarulho() {
        System.out.println("Au au!");
    }
}

class Gato extends Animal {
    void fazerBarulho() {
        System.out.println("Miau!");
    }
}

public class Teste {
    public static void main(String[] args) {
        Cachorro c = new Animal(); // Erro de compilação: Animal não é um tipo
válido para Cachorro
        c.fazerBarulho();
    }
}
```

Neste exemplo, `Cachorro` e `Gato` são subclasses de `Animal`, mas `Animal` não é uma subclasse de `Cachorro`. Portanto, você não pode atribuir um objeto `Animal` diretamente a uma variável do tipo

`Cachorro` sem fazer casting explícito. Lembra que em nossa questão era o contrário, no contexto desse código, devido ao polimorfismo, poderíamos perfeitamente fazer o contrário:

```
Animal c = new Cachorro();
```

Lembrando, novamente, que se houver a chamada de algum método específico da classe `Cachorro` pela variável de instância `c`, não vai compilar.

2. **Hierarquia de Classes Diferentes:**

```
class Fruta {
    void mostrarTipo() {
        System.out.println("Sou uma fruta.");
    }
}

class Maca extends Fruta {
    void mostrarTipo() {
        System.out.println("Sou uma maçã.");
    }
}

class Laranja extends Fruta {
    void mostrarTipo() {
        System.out.println("Sou uma laranja.");
    }
}

public class Teste {
    public static void main(String[] args) {
        Maca m = new Laranja(); // Erro de compilação: Laranja não é um tipo
válido para Maca
        m.mostrarTipo();
    }
}
```

Neste caso, `Maca` e `Laranja` são subclasses de `Fruta`, mas `Laranja` não é uma subclasse de `Maca`. Portanto, não é possível atribuir um objeto `Laranja` a uma variável do tipo `Maca`.

3. **Utilização de Interfaces:**

```
interface Movel {
    void mover();
}

class Carro implements Movel {
    public void mover() {
        System.out.println("Carro se movendo.");
```

```
    }

    void acelerar() {
        System.out.println("Carro acelerando.");
    }
}

class Aviao implements Movel {
    public void mover() {
        System.out.println("Avião voando.");
    }

    void decolar() {
        System.out.println("Avião decolando.");
    }
}

public class Teste {
    public static void main(String[] args) {
        Movel movel = new Carro(); // Correto, Carro implementa Movel
        movel.mover();

        // movel.acelerar(); // Erro de compilação: Movel não possui o método
acelerar
    }
}
```

Neste exemplo, `Carro` e `Aviao` implementam a interface `Movel`. A variável `movel` é do tipo `Movel`, que permite que seja referenciado um objeto de `Carro` ou `Aviao`. No entanto, como `movel` é do tipo `Movel`, você só pode chamar métodos definidos na interface `Movel`. Se você tentar chamar métodos específicos de `Carro` ou `Aviao` que não estão na interface `Movel`, você terá um erro de compilação.