Qual das seguintes afirmações pode preencher o espaço em branco para compilar o código com sucesso? (Escolha todas as opções aplicáveis.)

```
Set<? extends RuntimeException> mySet = new _________ ();
```

A. HashSet<? extends RuntimeException>
B. HashSet<Exception>
C. TreeSet<RuntimeException>
D. TreeSet<NullPointerException>
E. None of the above

ANÁLISE DA QUESTÃO

A questão é sobre tipos genéricos e como podemos criar um conjunto de exceções que se estendem de RuntimeException. Vamos analisar cada opção. Primeiro, a declaração original é:

```
Set<? extends RuntimeException> mySet = new _________ ();
```

Aqui, *Set<? extends RuntimeException>* significa que mySet pode ser qualquer tipo de conjunto cujos elementos sejam subtipos de RuntimeException.

Agora, vamos verificar as opções:

A. HashSet<? extends RuntimeException>

Isso não funciona porque, na criação de um *HashSet*, precisamos especificar um tipo concreto, não um wildcard (curinga). Então, *HashSet<? extends RuntimeException>* não é válido.

B. HashSet<Exception>

Isso também não funciona porque *Exception* não é um subclasse de *RuntimeException*. *Exception* é a **superclasse** de *RuntimeException*.

C. TreeSet<RuntimeException>

Sim, isso funciona. *TreeSet<RuntimeException>* é um *Set* onde os elementos são exatamente do tipo *RuntimeException* ou suas subclasses. Isso preenche o curinga *? extends RuntimeException* corretamente.

D. TreeSet<NullPointerException>

Sim, isso também funciona. *NullPointerException* é uma subclasse de *RuntimeException*, então *TreeSet<NullPointerException>* é um conjunto válido de subtipos de *RuntimeException*.

E. None of the above

Como já identificamos que C e D são opções corretas, esta opção está errada.

Portanto, as opções que preenchem corretamente o espaço em branco e fazem o código compilar são:

- **C. TreeSet<RuntimeException>**
- **D. TreeSet<NullPointerException>**

Vamos destrinchar o conceito de tipos genéricos com o '?' em Java, de um jeito bem simples e prático.

1. **Introdução aos Tipos Genéricos**

   Tipos genéricos em Java permitem que classes, interfaces e métodos **operem sobre tipos específicos de dados, mas sem especificar explicitamente esses tipos até que sejam usados.** Isso torna o código mais flexível e reutilizável.

2. **Curingas (Wildcards) em Java**

   Os curingas são usados em genéricos para indicar que qualquer tipo que se encaixe em uma certa restrição pode ser usado. Existem três principais tipos de curingas:

a) ***Curinga não delimitado (?):*** Representa qualquer tipo. É útil quando você quer aceitar qualquer tipo de parâmetro, mas não vai fazer nada que dependa do tipo específico.

```
List<?> list = new ArrayList<String>();
```

Aqui, *list* pode ser uma lista de qualquer tipo, no exemplo acima a lista criada é do tipo String, mas poderia ser de qualquer outro tipo.

Note que no caso da declaração abaixo, você está criando uma lista intencionalmente não especificada em termos de tipo de elemento (<?>). O wildcard (?) significa que o compilador não sabe ou especifica que tipo de objetos a lista pode conter. Isso é útil para cenários onde você deseja flexibilidade nos tipos de objetos armazenados na lista, mas também **significa que você não pode adicionar objetos arbitrários diretamente a ela.**

```
List<?> list = new ArrayList<>();
```

O compilador impedirá você de adicionar objetos a um List<?> porque não pode garantir a segurança de tipos. Como a lista pode conter qualquer tipo de objetos (ou uma mistura de tipos), permitir adições arbitrárias poderia levar a inconsistências de tipo em tempo de execução.

b) ***Curinga delimitado superior (? extends Type):*** Representa qualquer tipo que seja um subtipo de *Type*. Isso é útil quando você quer ler de uma estrutura de dados e não quer modificar (escrever) nela.

```
List<? extends Number> list = new ArrayList<Integer>();
```

Aqui, list pode ser uma lista de qualquer subtipo de *Number*, como *Integer*, *Double*, etc.

c) **Curiga delimitado inferior (? super Type):** Representa qualquer tipo que seja um supertipo de Type. Isso é útil quando você quer escrever em uma estrutura de dados.

```
List<? super Integer> list = new ArrayList<Number>();
```

Aqui, list pode ser uma lista de qualquer supertipo de *Integer*, como *Number* ou *Object*.