

Aula 05

Robustez

Gestão de Falhas

Programação II, 2019-2020

v1.10, 08-03-2020

Mecanismo de Exceções

- Fundamentos
- Exceções em Java
- Especificar Exceções em Métodos
- Classificação de Exceções
- Discussão

Gestão de Falhas em Módulos

- Técnica da Avestruz
- Programação Defensiva (caso 1)
- Programação Defensiva (caso 2)
- Programação por Contrato
- Discussão

Gestão de Falhas em Programas

1 Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

2 Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

3 Gestão de Falhas em Programas

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

1 Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

2 Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

3 Gestão de Falhas em Programas

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: Motivação

- Durante a execução de um programa, por vezes ocorrem **eventos anómalos** ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de **exceções**.
- Esses eventos podem ser causados por **erros internos** do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice fora dos limites de um array, dividir por zero, etc.
- Ou podem ser devidos a **erros externos**, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante **interromper** imediatamente o fluxo de execução.
- Por outro lado, se for possível rectificar a situação, é importante poder **retomar a execução** normal do programa.
- O **mecanismo de exceções** serve precisamente estes dois propósitos.

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - Quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - Quando a exceção desce para algum *try* num *bloco vigiado*, onde o tratamento está tipicamente disponível.

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - Quando interrompe o código *main*, passando a ser tratado pelo programa com uma mensagem de erro; ou
 - Quando a exceção desce para um *bloco vigiado* (ver abaixo) e encontra o tipo de exceção

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Exceções: como funcionam (1)

O mecanismo de exceções funciona assim:

- Quando a exceção ocorre, a instrução que estava a ser executada não termina e a **execução é interrompida**. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a exceção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Se a instrução interrompida não estiver num *bloco vigiado* (ver abaixo), então o método que a contém é interrompido e encaminha o **objeto-exceção** para o local de onde foi invocado.
- Nesse local, o processo repete-se: a instrução de invocação é interrompida, o método onde está também, a exceção é reencaminhada e assim sucessivamente.
- Esta propagação da exceção só termina:
 - quando interrompe o método `main`, causando a terminação do programa com uma mensagem de erro; ou
 - quando a instrução causadora estiver num bloco vigiado que intercepte esse tipo de exceção.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Exceções: como funcionam (2)

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

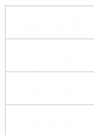
Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ...; ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ...; ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ...; ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ...; ← não chega a ser executado (1)  
    }  
}
```



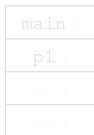
Exceções: como funcionam (2)

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```



Exceções: como funcionam (2)

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```



Exceções: como funcionam (2)

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ...; ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```

main	⚡
p1	⚡
p2	⚡
p3	⚡

Exceções: como funcionam (2)

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```

main	⚡
p1	⚡
p2	⚡
p3	⚡

Exceções: como funcionam (2)

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```

main	⚡
p1	⚡
p2	⚡
p3	⚡

Exceções: como funcionam (2)

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```

main	⚡
p1	⚡
p2	⚡
p3	⚡

Exceções: como funcionam (2)

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (...) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (...) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (...) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (...) ← não chega a ser executado (1)  
    }  
}
```

main ⚡

p1 ⚡

p2 ⚡

p3 ⚡

Exceções: como funcionam (2)

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

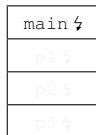
Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Example {  
    public static void main(String[] args) {  
        ...; p1(); ⚡ (4) ← não chega a ser executado (4)  
    }  
  
    static void p1() {  
        ...; p2(); ⚡ (3) ← não chega a ser executado (3)  
    }  
  
    static void p2() {  
        ...; p3(); ⚡ (2) ← não chega a ser executado (2)  
    }  
  
    static void p3() {  
        ...; throw ⚡ (1) ← não chega a ser executado (1)  
    }  
}
```



Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Em Java, o mecanismo é implementado através de:

- Uma instrução – `throw` – que permite gerar exceções.
- Uma cláusula – `throws` – que serve para declarar a lista de exceções que um método pode gerar ou propagar.
- Uma instrução composta – `try/catch/finally` – que permite definir um bloco de código vigiado e interceptar exceções que aí ocorram.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Em Java, o mecanismo é implementado através de:

- Uma instrução – `throw` – que permite gerar exceções.
- Uma cláusula – `throws` – que serve para declarar a lista de exceções que um método pode gerar ou propagar.
- Uma instrução composta – `try/catch/finally` – que permite definir um bloco de código vigiado e interceptar exceções que aí ocorram.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Em Java, o mecanismo é implementado através de:

- Uma instrução – `throw` – que permite gerar exceções.
- Uma cláusula – `throws` – que serve para declarar a lista de exceções que um método pode gerar ou propagar.
- Uma instrução composta – `try/catch/finally` – que permite definir um bloco de código vigiado e interceptar exceções que aí ocorram.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Em Java, o mecanismo é implementado através de:

- Uma instrução – `throw` – que permite gerar exceções.
- Uma cláusula – `throws` – que serve para declarar a lista de exceções que um método pode gerar ou propagar.
- Uma instrução composta – `try/catch/finally` – que permite definir um bloco de código vigiado e interceptar exceções que aí ocorram.

Exceções em Java (2)

- Gerar (ou lançar) exceção:

```
if (t == null)
    throw new NullPointerException();
// throw new NullPointerException("t null");
```

- Declarar lista de exceções potenciais:

```
public
void func() throws NullPointerException, IOException
{ ... }
```

- Interceptar (ou apanhar) exceções:

```
try {
    /* Bloco de código normal a vigiar */
}
catch (Errortype a) {
    /* Código para retificar a causa da exceção */
}
```

Exceções em Java (2)

- Gerar (ou lançar) exceção:

```
if (t == null)
    throw new NullPointerException();
// throw new NullPointerException("t null");
```

- Declarar lista de exceções potenciais:

```
public
void func() throws NullPointerException, IOException
{ ... }
```

- Interceptar (ou apanhar) exceções:

```
try {
    /* Bloco de código normal a vigiar */
}
catch (Errortype a) {
    /* Código para retificar a causa da exceção */
}
```

Exceções em Java (2)

- Gerar (ou lançar) exceção:

```
if (t == null)
    throw new NullPointerException();
// throw new NullPointerException("t null");
```

- Declarar lista de exceções potenciais:

```
public
void func() throws NullPointerException, IOException
{ ... }
```

- Interceptar (ou apanhar) exceções:

```
try {
    /* Bloco de código normal a vigiar */
}
catch (Errortype a) {
    /* Código para retificar a causa da exceção */
}
```

- Gerar (ou lançar) exceção:

```
if (t == null)
    throw new NullPointerException();
// throw new NullPointerException("t null");
```

- Declarar lista de exceções potenciais:

```
public
void func() throws NullPointerException, IOException
{ ... }
```

- Interceptar (ou apanhar) exceções:

```
try {
    /* Bloco de código normal a vigiar */
}
catch (Errortype a) {
    /* Código para retificar a causa da exceção */
}
```

A manipulação de excepções é feita através da instrução `try/catch/finally`:

```
try {  
    // Código que pode gerar excepções do tipo Type1,  
    // Type2 ou Type3  
} catch(Type1 id1) {  
    // Gerir excepção do tipo Type1  
} catch(Type2 id2) {  
    // Gerir excepção do tipo Type2  
} catch(Type3 id3) {  
    // Gerir excepção do tipo Type3  
} finally {  
    // Bloco executado independentemente de haver  
    // ou não uma excepção  
}
```

A manipulação de excepções é feita através da instrução
`try/catch/finally`:

```
try {  
    // Código que pode gerar excepções do tipo Type1,  
    // Type2 ou Type3  
} catch(Type1 id1) {  
    // Gerir excepção do tipo Type1  
} catch(Type2 id2) {  
    // Gerir excepção do tipo Type2  
} catch(Type3 id3) {  
    // Gerir excepção do tipo Type3  
} finally {  
    // Bloco executado independentemente de haver  
    // ou não uma excepção  
}
```

A manipulação de exceções é feita através da instrução
`try/catch/finally`:

```
try {  
    // Código que pode gerar exceções do tipo Type1,  
    // Type2 ou Type3  
} catch(Type1 id1) {  
    // Gerir exceção do tipo Type1  
} catch(Type2 id2) {  
    // Gerir exceção do tipo Type2  
} catch(Type3 id3) {  
    // Gerir exceção do tipo Type3  
} finally {  
    // Bloco executado independentemente de haver  
    // ou não uma exceção  
}
```

Classe Throwable

- Podemos usar a exceção `java.lang.Throwable` para interceptar exceções de qualquer tipo:

```
catch(Throwable e) { // Apanha todas as exceções
    err.println("Caught exception: " + e.getMessage());
    exit(1);
}
```

- Podemos gerar nova exceção de forma a ser tratada num nível superior:

```
catch(Throwable e) {
    ... (faz qualquer coisa)
    throw e; // A exceção vai ser relançada
}
```


- Podemos usar a exceção `java.lang.Throwable` para interceptar exceções de qualquer tipo:

```
catch(Throwable e) { // Apanha todas as exceções
    err.println("Caught exception: " + e.getMessage());
    exit(1);
}
```

- Podemos gerar nova exceção de forma a ser tratada num nível superior:

```
catch(Throwable e) {
    ...(faz qualquer coisa)
    throw e; // A exceção vai ser relançada
}
```

- Podemos usar a exceção `java.lang.Throwable` para interceptar exceções de qualquer tipo:

```
catch(Throwable e) { // Apanha todas as exceções
    err.println("Caught exception: " + e.getMessage());
    exit(1);
}
```

- Podemos gerar nova exceção de forma a ser tratada num nível superior:

```
catch(Throwable e) {
    ...(faz qualquer coisa)
    throw e; // A exceção vai ser relançada
}
```

Especificar Exceções em Métodos

- A linguagem Java permite que se associe à assinatura dos métodos uma lista de exceções que os mesmos podem lançar:

```
void m() throws TooBigException,  
               TooSmallException,  
               DivByZeroException {  
    //...  
}
```

- Desta forma, o (eventual) lançamento destas exceções passa a fazer parte da informação sintáctica sobre o método.

- A linguagem Java permite que se associe à assinatura dos métodos uma lista de exceções que os mesmos podem lançar:

```
void m() throws TooBigException,  
               TooSmallException,  
               DivByZeroException {  
    //...  
}
```

- Desta forma, o (eventual) lançamento destas exceções passa a fazer parte da informação sintáctica sobre o método.

- A linguagem Java permite que se associe à assinatura dos métodos uma lista de exceções que os mesmos podem lançar:

```
void m() throws TooBigException,  
               TooSmallException,  
               DivByZeroException {  
    //...  
}
```

- Desta forma, o (eventual) lançamento destas exceções passa a fazer parte da informação sintáctica sobre o método.

Classificação das Exceções

A linguagem Java agrupa as exceções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *exceções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma exceção *checked* tem de estar:
 - dentro de um bloco `try` com um `catch` que apanhe esse tipo de exceção, ou então
 - dentro de um método que especifique que pode propagar esse tipo de exceção, através de uma cláusula `throws` na declaração do método.
- As *exceções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as exceções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - Podem ser apanhadas (com `catch`).
 - Se não forem apanhadas, não propagam automaticamente.

Classificação das Excepções

A linguagem Java agrupa as excepções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *excepções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma excepção *checked* tem de estar:
 - dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de excepção, ou então
 - dentro de um método que *especifique* que pode propagar esse tipo de excepção, através de uma cláusula `throws` na declaração do método.
- As *excepções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as excepções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - Podem ser apanhadas (com `catch`).
 - Se não forem apanhadas, são propagadas automaticamente.

Classificação das Excepções

A linguagem Java agrupa as excepções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *excepções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma excepção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de excepção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de excepção, através de uma cláusula `throws` na declaração do método.
- As *excepções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as excepções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Excepções

A linguagem Java agrupa as excepções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *excepções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma excepção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de excepção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de excepção, através de uma cláusula `throws` na declaração do método.
- As *excepções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as excepções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Exceções

A linguagem `Java` agrupa as exceções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *exceções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma exceção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de exceção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de exceção, através de uma cláusula `throws` na declaração do método.
- As *exceções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as exceções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Excepções

A linguagem `Java` agrupa as excepções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *excepções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma excepção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de excepção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de excepção, através de uma cláusula `throws` na declaração do método.
- As *excepções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as excepções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Excepções

A linguagem `Java` agrupa as excepções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *excepções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma excepção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de excepção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de excepção, através de uma cláusula `throws` na declaração do método.
- As *excepções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as excepções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Exceções

A linguagem `Java` agrupa as exceções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *exceções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma exceção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de exceção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de exceção, através de uma cláusula `throws` na declaração do método.
- As *exceções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as exceções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Exceções

A linguagem `Java` agrupa as exceções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *exceções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma exceção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de exceção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de exceção, através de uma cláusula `throws` na declaração do método.
- As *exceções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as exceções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

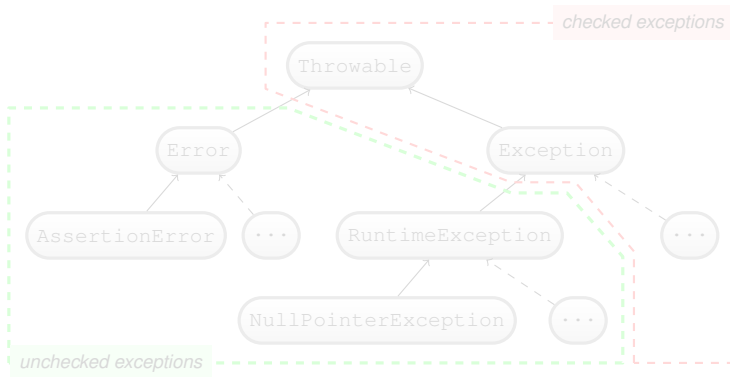
Classificação das Exceções

A linguagem `Java` agrupa as exceções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *exceções checked* obrigam o programador a apanhá-las ou a especificar que as propaga.
- Assim, qualquer excerto de código que possa lançar uma exceção *checked* tem de estar:
 - 1 dentro de um bloco `try` com um `catch` que *apanhe* esse tipo de exceção, ou então
 - 2 dentro de um método que *especifique* que pode propagar esse tipo de exceção, através de uma cláusula `throws` na declaração do método.
- As *exceções unchecked* diferem das anteriores apenas pelo facto de não imporem essa obrigação.
- Ou seja, as exceções *unchecked* não são ignoradas. Funcionam da mesma forma que as outras:
 - 1 Podem ser apanhadas (com `catch`).
 - 2 Se não forem apanhadas, são propagadas automaticamente.

Classificação das Exceções (2)

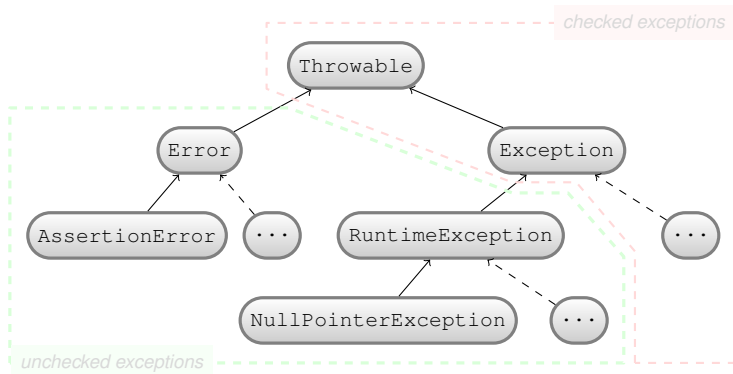
- As exceções organizam-se numa hierarquia de tipos e subtipos.



- As *unchecked exceptions* são todas aquelas que derivam das classes **RuntimeException** ou **Error**.
- Todas as outras são *checked exceptions*.

Classificação das Exceções (2)

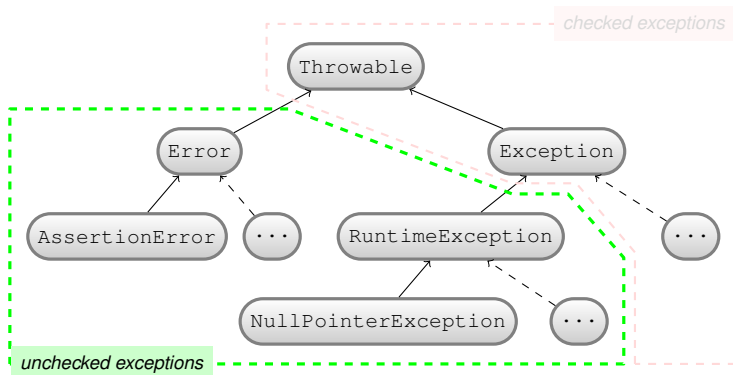
- As exceções organizam-se numa hierarquia de tipos e subtipos.



- As *unchecked exceptions* são todas aquelas que derivam das classes **RuntimeException** ou **Error**.
- Todas as outras são *checked exceptions*.

Classificação das Exceções (2)

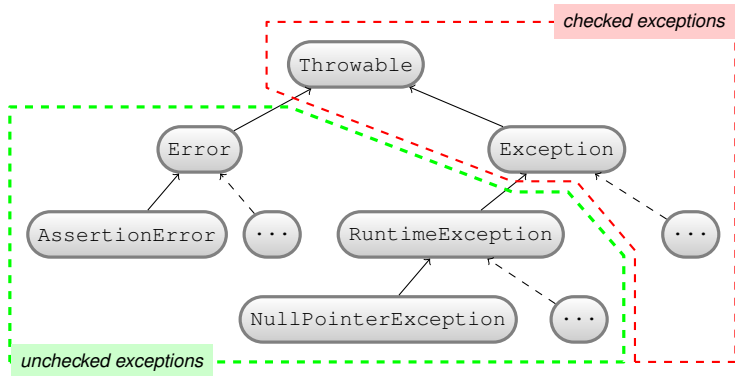
- As exceções organizam-se numa hierarquia de tipos e subtipos.



- As *unchecked exceptions* são todas aquelas que derivam das classes `RuntimeException` ou `Error`.
- Todas as outras são *checked exceptions*.

Classificação das Exceções (2)

- As exceções organizam-se numa hierarquia de tipos e subtipos.



- As *unchecked exceptions* são todas aquelas que derivam das classes `RuntimeException` ou `Error`.
- Todas as outras são *checked exceptions*.

Algumas vantagens das exceções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Algumas vantagens das exceções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Algumas vantagens das exceções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Algumas vantagens das exceções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

Algumas vantagens das exceções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

Algumas vantagens das exceções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

Ignorar as falhas e continuar a trabalhar.



- **Programação Defensiva:**

Tratar todas as situações, ter código específico para lidar com erros.

- **Programação por Contrato:**

Assumir responsabilidades no módulo.

O módulo só tem de cumprir a sua parte do contrato.

Assumir responsabilidades por contrato para o módulo de testes, em tempo de execução (as APIs não têm de cumprir o contrato).

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Existem basicamente três possibilidades:

- Técnica da avestruz:



- Programação Defensiva:

- Programação por Contrato:

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- Não aconselhável!



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- Não aconselhável!



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- **Não aconselhável!**



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

```
public class Data {
```

```
    public Data(int dia,int mes,int ano) {  
        aDia = dia; aMes = mes; aAno = ano;  
    }
```



```
    public static int diasDoMes(int mes,int ano) {  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```

```
    public void main(String[] args) {  
        Data d = new Data(25,4,1974);  
        ...  
        if (Data.diasDoMes(mes,ano) != 31)  
            ...  
    }
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```



```
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```



```
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```

Programação Defensiva (caso 1)

Robustez

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia, int mes, int ano) {  
        if (!valida(dia, mes, ano)) {  
            aErro = true;        }  
        else {  
            aErro = false;        }  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
}  
  
    public static int diasDoMes(int mes, int ano) {  
        int result;  
  
        if (!mesValido(mes)) {  
            result = -1;        }  
        else {  
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
        }  
        return result;  
}  
  
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia, aMes, aAno;  
}
```

erro guardado num atributo

erro no resultado da função

função e atributo de erro

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    if (d.erro())  
        doSomethingWithError;  
    ...  
    int r = Data.diasDoMes(mes, ano);  
    if (r == -1)  
        doSomethingWithError;  
    if (r != 31)  
        ...  
}
```


Programação Defensiva (caso 1)

Robustez

Mecanismo de Exceções

- Fundamentos
- Exceções em Java
- Especificar Exceções em Métodos
- Classificação de Exceções
- Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        if (!valida(dia,mes,ano)) {  
            aErro = true;  
        } else {  
            aErro = false;  
            aDia = dia; aMes = mes; aAno = ano;  
        }  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        int result;  
  
        if (!mesValido(mes)) {  
            result = -1;  
        } else {  
            final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
        }  
        return result;  
    }  
  
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia,aMes,aAno;  
}
```

erro guardado num atributo

erro no resultado da função

função e atributo de erro

```
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    if (d.erro())  
        doSomethingWithError;  
    ...  
    int r = Data.diasDoMes(mes,ano);  
    if (r == -1)  
        doSomethingWithError;  
    if (r != 31)  
        ...  
}
```

Programação Defensiva (caso 1)

Robustez

```
public class Data {  
  
    public Data(int dia, int mes, int ano) {  
        if (!valida(dia, mes, ano)) {  
            aErro = true;   
        }   
        else {  
            aErro = false;  
            aDia = dia; aMes = mes; aAno = ano;  
        }  
    }  
  
    public static int diasDoMes(int mes, int ano) {  
        int result;  
  
        if (!mesValido(mes)) {  
            result = -1;  
        }   
        else {  
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
        }  
        return result;  
    }  
  
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia, aMes, aAno;  
}
```

erro guardado num atributo

erro no resultado da função

função e atributo de erro

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    if (d.erro())  
        doSomethingWithError;  
    ...  
    int r = Data.diasDoMes(mes, ano);  
    if (r == -1)  
        doSomethingWithError;  
    if (r != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Programação Defensiva (caso 1)

Robustez

```
public class Data {
```

```
    public Data(int dia, int mes, int ano) {  
        if (!valida(dia, mes, ano)) {  
            aErro = true;  
        }  
        else {  
            aErro = false;  
            aDia = dia; aMes = mes; aAno = ano;  
        }  
    }  
}
```

erro guardado num atributo

```
    public static int diasDoMes(int mes, int ano) {  
        int result;
```

```
        if (!mesValido(mes)) {  
            result = -1;  
        }  
        else {  
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
        }  
        return result;  
    }
```

erro no resultado da função

função e atributo de erro

```
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia, aMes, aAno;  
}
```

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    if (d.erro())  
        doSomethingWithError();  
    ...  
    int r = Data.diasDoMes(mes, ano);  
    if (r == -1)  
        doSomethingWithError();  
    if (r != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Programação Defensiva (caso 1)

Robustez

```
public class Data {

    public Data(int dia, int mes, int ano) {
        if (!valida(dia, mes, ano)) {
            aErro = true;
        } else {
            aErro = false;
            aDia = dia; aMes = mes; aAno = ano;
        }
    }

    public static int diasDoMes(int mes, int ano) {
        int result;

        if (!mesValido(mes)) {
            result = -1;
        } else {
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
            result = dias[mes-1];
            if (mes == 2 && anoBissexto(ano))
                result++;
        }
        return result;
    }

    public boolean erro() { return aErro; }
    private boolean aErro = false;
    private int aDia, aMes, aAno;
}
```

erro guardado num atributo

erro no resultado da função

função e atributo de erro

```
public void main(String[] args) {
    Data d = new Data(25, 4, 1974);
    if (d.erro())
        doSomethingWithError();
    ...
    int r = Data.diasDoMes(mes, ano);
    if (r == -1)
        doSomethingWithError();
    if (r != 31)
        ...
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

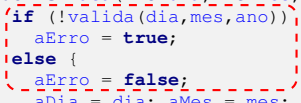
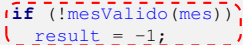
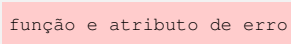
Programação por Contrato

Discussão

Gestão de Falhas em Programas

Programação Defensiva (caso 1)

Robustez

```
public class Data {  
  
    public Data(int dia, int mes, int ano) {  
        if (!valida(dia, mes, ano)) {  
            aErro = true;  erro guardado num atributo  
        } else {  
            aErro = false;  
            aDia = dia; aMes = mes; aAno = ano;  
        }  
    }  
  
    public static int diasDoMes(int mes, int ano) {  
        int result;  
  
        if (!mesValido(mes)) {  
            result = -1;  erro no resultado da função  
        } else {  
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
             função e atributo de erro  
        }  
        return result;  
    }  
  
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia, aMes, aAno;  
}
```

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    if (d.erro())  
        doSomethingWithError;  
    ...  
    int r = Data.diasDoMes(mes, ano);  
    if (r == -1)  
        doSomethingWithError;  
    if (r != 31)  
        ...  
}
```

Mecanismo de Exceções

- Fundamentos
- Exceções em Java
- Especificar Exceções em Métodos
- Classificação de Exceções
- Discussão

Gestão de Falhas em Módulos

- Técnica da Avestruz

Programação Defensiva (caso 1)

- Programação Defensiva (caso 2)
- Programação por Contrato
- Discussão

Gestão de Falhas em Programas

Programação Defensiva (caso 1)

Robustez

```
public class Data {  
  
    public Data(int dia, int mes, int ano) {  
        if (!valida(dia, mes, ano)) {  
            aErro = true;   
        } else {  
            aErro = false;  
            aDia = dia; aMes = mes; aAno = ano;  
        }  
    }  
  
    public static int diasDoMes(int mes, int ano) {  
        int result;  
  
        if (!mesValido(mes)) {  
            result = -1;  
        } else {  
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
        }  
        return result;  
    }  
  
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia, aMes, aAno;  
}
```

erro guardado num atributo

erro no resultado da função

função e atributo de erro

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    if (d.erro())  
        doSomethingWithError;  
    ...  
    int r = Data.diasDoMes(mes, ano);  
    if (r == -1)  
        doSomethingWithError;  
    if (r != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

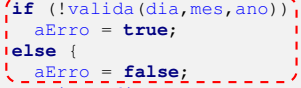
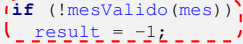
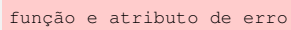
Programação por Contrato

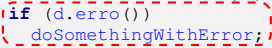
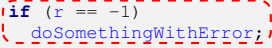
Discussão

Gestão de Falhas em Programas

Programação Defensiva (caso 1)

Robustez

```
public class Data {  
  
    public Data(int dia, int mes, int ano) {  
        if (!valida(dia, mes, ano)) {  
            aErro = true;  erro guardado num atributo  
        } else {  
            aErro = false;  
            aDia = dia; aMes = mes; aAno = ano;  
        }  
    }  
  
    public static int diasDoMes(int mes, int ano) {  
        int result;  
  
        if (!mesValido(mes)) {  erro no resultado da função  
            result = -1;  
        } else {  
            final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
            result = dias[mes-1];  
            if (mes == 2 && anoBissexto(ano))  
                result++;  
             função e atributo de erro  
        }  
        return result;  
    }  
  
    public boolean erro() { return aErro; }  
    private boolean aErro = false;  
    private int aDia, aMes, aAno;  
}
```

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    if (d.erro())  doSomethingWithError;  
    ...  
    int r = Data.diasDoMes(mes, ano);  
    if (r == -1)  doSomethingWithError;  
    if (r != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) (throws IllegalArgumentException) {  
        if (!valida(dia,mes,ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25,4,1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes,ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```



```
public class Data {  
  
    public Data(int dia, int mes, int ano) throws IllegalArgumentException {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) throws IllegalArgumentException {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código `catch` deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código `try` (inserindo todo o bloco `try/catch` num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    try {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
    try {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    try {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
    try {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    try {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
    try {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    try {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
    try {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    catch (IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```



```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```

```
public class Data {  
  
    public Data(int dia, int mes, int ano) (throws IllegalArgumentException) {  
        if (!valida(dia, mes, ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) (throws IllegalArgumentException) {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

erro lançado como exceção

NOTA MUITO IMPORTANTE

No código catch deve-se: **terminar o programa**, ou **propagar a exceção**, ou **voltar a tentar** o código try (inserindo todo o bloco try/catch num ciclo). Qualquer outra ação pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    (try) {  
        d = new Data(25, 4, 1974);  
        ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
    (try) {  
        if (Data.diasDoMes(mes, ano) != 31)  
            ...  
    }  
    (catch (IllegalArgumentException e)) {  
        doSomethingWithError;  
    }  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        assert valida(dia,mes,ano); ← asserções  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        assert mesValido(mes); ←  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```

```
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        assert valida(dia,mes,ano);  
  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        assert mesValido(mes);  
  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```

asserções

```
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia, int mes, int ano) {  
        assert valida(dia, mes, ano); ← asserções  
  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes, int ano) {  
        assert mesValido(mes); ← asserções  
  
        final int[] dias = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia, aMes, aAno;  
}
```

```
public void main(String[] args) {  
    Data d = new Data(25, 4, 1974);  
    ...  
    if (Data.diasDoMes(mes, ano) != 31)  
        ...  
}
```

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        assert valida(dia,mes,ano);;  
  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        assert mesValido(mes);;  
  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```

asserções

```
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```

Gestão de falhas em módulos

- Técnica da Avestruz:

- Código simples, mas não seguro

- Programação Defensiva:

- Código mais complexo, mas com garantia de que o código detectará situações de falha (erro de programação)

- No caso 2 (exceções obrigatórias), o código não garante que o erro não ocorra, desde que não seja detectado pelo compilador, mas garante a propagação.

- Código mais complexo

- Programação por Contrato:

- Código simples, porém com informações adicionais

- No caso de se pretender se evitar a exceção, basta definir o contrato de programação de forma que seja impossível violá-lo, ou então garantir que o contrato sempre será cumprido.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**

- Código **simples**, mas **não robusto**.

- **Programação Defensiva:**

- Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
- No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
- Código mais **complexo**.

- **Programação por Contrato:**

- Código **simples**, **interna e externamente robusto**;
- No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **interna e externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples, interna e externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **interna e externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **internamente** e **externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **internamente** e **externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **interna e externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **interna e externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

- **Técnica da Avestruz:**
 - Código **simples**, mas **não robusto**.
- **Programação Defensiva:**
 - Código **internamente robusto**, mas sem garantir que os clientes detectam situações de erro (**externamente não robusto**).
 - No caso 2 (exceções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
 - Código mais **complexo**.
- **Programação por Contrato:**
 - Código **simples**, **interna e externamente robusto**;
 - No caso de se pretender apanhar a exceção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Erros internos: Erros >10% da responsabilidade do programador. A metodologia de programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Erros externos: Erros que não têm em completamente da responsabilidade do programador. Estes erros têm origem em factores externos ao programa. Para estes erros a melhor metodologia disponível é a da programação defensiva.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva.
- Temos assim dois tipos de erros num programa:

Internos: Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

Externos: Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quando são e eventuais problemas de conversão para números, strings vazias, etc.
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, strings vazias, etc.
- Leitura de ficheiros:
 - Lidar com a excepção `FileNotFoundException` (falha ao abrir o ficheiro)
 - Lidar com a excepção `IOException` na criação do scanner.
 - Nas excepções de `IOException` lidar com as excepções `IOException` e `FileNotFoundException`.
- Escrita de ficheiros:
 - Lidar com a excepção `FileNotFoundException` (falha ao abrir o ficheiro)
 - Lidar com a excepção `IOException` na criação do scanner.
 - Após uso do `PrintWriter`, lidar com a excepção de erro de escrita com o método `close()`.

Mecanismo de Excepções

Fundamentos

Excepções em Java

Especificar Excepções em Métodos

Classificação de Excepções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

- **Argumentos do programa** (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- **Entradas do utilizador** (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- **Leitura de ficheiros:**
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- **Escrita de ficheiros:**
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` & `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções
Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva
(caso 1)

Programação Defensiva
(caso 2)

Programação por Contrato
Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Mecanismo de Exceções

Fundamentos

Exceções em Java

Especificar Exceções em Métodos

Classificação de Exceções

Discussão

Gestão de Falhas em Módulos

Técnica da Avestruz

Programação Defensiva (caso 1)

Programação Defensiva (caso 2)

Programação por Contrato

Discussão

Gestão de Falhas em Programas

- Argumentos do programa (`main(String[] args);`)
 - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in);`)
 - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
 - Nas operações de `next...` lidar com as exceções: `InputMismatchException` e `NoSuchElementException`.
- Escrita de ficheiros:
 - Lidar com a exceção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
 - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.