JAVA 4

# Java 2 API, Strings e Exceções

OS PRINCIPAIS PACOTES QUE COMPÕEM A API JAVA são apresentados neste módulo. Os módulos são apresentados de maneira superficial. Em detalhes serão mostrados exemplos de classes encapsuladoras, as classes String e StringBuffer e as classes usadas no controle de exceções.

## Tópicos abordados neste módulo

- API Java
- Principais classes
- Strings
- Exceções

## Índice



## Objetivos

No final deste módulo você deverá ser capaz de:

- Identificar os principais pacotes da API Java
- Saber quando usar String e quando usar StringBuffer
- Conhecer a sintaxe das exceções
- Criar novas classes de Exceções
- Usar exceções para controlar condições excepcionais tempo de execução.

## 1.1. Java 2 Application Programming Interface

Java organiza as classes da sua biblioteca padrão em pacotes. Este conjunto de pacotes com suas classes forma a interface para a programação de aplicações, ou *API – Application Programming Interface*. Esse conjunto padrão é freqüentemente chamado de Core API, ou API de núcleo, porque faz parte dos ambientes de execução e programação Java em todas as plataformas que suportam a versão 2. Várias outras APIs existem que estendem a linguagem Java. Desde APIs proprietárias, como a IFC da Netscape ou a AFC, da Microsoft, como APIs da própria Sun como Java Foundation Classes (JFC), Java Multimedia API, Java Management API, Java Card, etc.

Os principais pacotes da API Java 2 estão descritos a seguir. São usados para diversos tipos de tarefas. Vários outros pacotes e subpacotes menos usados não estão descritos aqui.

Java oferece suporte à computação distribuída através de pacotes especiais, alguns dos quais não são distribuídos originalmente com o Java 2, como o pacote usado no desenvolvimento de servlets (javax.servlet). Para a construção de aplicações de rede, diversas classes e interfaces permitem a realização de operações que incluem desde o tratamento em baixo nível do fluxo de dados e de caracteres à operações complexas como a comunicação com objetos remotos, escritos em outra linguagem, de maneira totalmente transparente.

Os pacotes que começam com java.* são chamados de pacotes do núcleo ("core"). Os que começam com javax são extensões padrão ao núcleo ("core extensions"). Os pacotes sun.* e omg.* fazem parte de toda distribuição Java mas não são considerados da API Java fundamental. O pacote sun.* contém as ferramentas padrão da Sun como o compilador, o driver JDBC-ODBC, etc. e o pacote omg.* contém o suporte a CORBA.

### java.applet

Contém a classe Applet e outras classes pouco usadas (são usadas internamente) como AppletStub e AppletContext. Contém uma classe AudioClip que representa um clipe musical.

## java.awt

É o pacote que contém classes para o desenvolvimento de interfaces gráficas. Contém classes para lidar com a apresentação de informações em uma GUI, criação e posicionamento de componentes, tratamento de eventos, impressão e atalhos de teclado.

## java.awt.image

Contém filtros e outras classes de manipulação de imagens.

## java.awt.peer

Contém interfaces para um conjunto de ferramentas gráficas independentes de plataforma. As classes deste pacote raramente são usadas a não ser por classes do AWT ou extensões.

## java.awt.datatransfer

As classes deste pacote definem uma estrutura genérica para a transferência de dados entre aplicações como classes que suportam um modelo de dados cut-and-paste de uma área de transferência.

## java.awt.event

Classes e interfaces que suportam o modelo de eventos baseado em delegação. São três as categorias das classes e interfaces: classes que representam eventos da GUI; interfaces que escutam eventos (definem métodos que devem ser implementados pelos objetos interessados em serem notificados na ocorrência de um evento; e implementações triviais das interfaces de escuta (classes adaptadoras de eventos).

## java.beans

Contém classes e interfaces para criar e usar componentes reutilizáveis chamados beans. Podem ser usadas de três formas: para criar ferramentas de construção de aplicações (para criar aplicações quase sem programar); para desenvolver novos beans para serem usados nesses construtores de aplicações e; para desenvolver aplicações que usam beans.

## java.lang

Contém as classes fundamentais da linguagem Java como Object, String, Thread, System e Exception. Também contém classes empacotadoras de tipos primitivos (Byte, Short, Integer, Long, Float, Double, Character, Boolean e Void), para que os mesmos possam ser tratados como objetos, quando necessário.

## java.lang.reflect

Contém classes que permitem que um programa em Java examine a estrutura de classes e obtenha informações completas sobre qualquer objeto, vetor, método, construtor ou campo de dados. É usado na programação genérica.

## java.math

Contém duas classes apenas que oferecem suporte a operações aritméticas sobre inteiros de tamanho arbitrário e números de ponto-flutuante de precisão arbitrária.

## java.io

Este pacote oferece suporte a operações e entrada e saída. Suas classes e interfaces podem ser classificadas em três grupos. O primeiro, que consiste da maior parte do pacote, é utilizado na construção e filtragem de fluxos (*streams*) de dados. Contém classes e interfaces que representam fluxos de entrada e saída de *bytes*, caracteres e objetos. O segundo grupo contém classes e interfaces que suportam a *serialização* de objetos (conversão de objetos em fluxos de *bytes* registrados com número de série para possibilitar o armazenamento persistente). O último grupo é usado para representar um sistema de arquivos.

## java.net

Este pacote oferece suporte a aplicações de rede.. Contém classes e interfaces que permitem a implementação de clientes, servidores e protocolos TCP/IP utilizando *sockets*, datagramas UDP, endereços Internet, conexões HTTP e URLs.

## java.rmi

Suporta a arquitetura de objetos remotos RMI – *Remote Method Invocation*, que permite o desenvolvimento de aplicações que invocam métodos em objetos localizados em máquinas virtuais diferentes. Baseia-se no protocolo JRMP – Java Remote Method Protocol que permite a comunicação entre máquinas virtuais Java.

## java.sql

Pacote que suporta JDBC – *Java Database Connectivity*. Com estas classes e interfaces é possível desenvolver programas em Java que se comunicam com qualquer banco de dados relacional que suporte as operações mínimas do SQL92. Também oferece suporte ao desenvolvimento de *drivers* JDBC.

## javax.servlet.

Extensão de núcleo (core extension) da API Java que oferece suporte ao desenvolvimento de componentes de servidor – os *servlets*. Servlets são componentes que rodam dentro de uma aplicação de servidor. Servlets HTTP podem ser usados como alternativa eficiente, aberta e independente de plataforma à tecnologias atualmente utilizadas nos servidores HTTP como CGI[1], ASP[2], ISAPI[3] ou Cold Fusion[4].

## org.omg.CORBA e org.omg.cosNaming.

Extensão padrão do Java 2 (standard extension). Estes pacotes oferecem classes, interfaces e subpacotes que permitem o desenvolvimento de aplicações Java que usam a tecnologia e objetos remotos CORBA.

---

[1] Common Gateway Interface. especificação da W3C (World Wide Web Consortium) para aplicações no servidor invocadas pelo browser.

[2] Active Server Pages. Tecnologia da Microsoft que oferece uma alternativa ao CGI através de scripts embutidos em páginas Web.

[3] Internet Server Application Programmer's Interface. Alternativa ao CGI através de módulos ou bibliotecas dinâmicas desenvolvidas usando uma interface de programação proprietária. ISAPI é suportado pelos servidores Microsoft. Tecnologias concorrentes são NSAPI (servidor Netscape) e Apache Server API.

[4] Alternativa à tecnologia CGI (proprietária). Desenvolvida pela Allaire Inc.

### javax.rmi

Extensão de núcleo (core extension) da API Java que oferece suporte a RMI usando o protocolo IIOP (Internet Inter-ORB Protocol) e o serviço de nomes da plataforma Java permitindo que objetos RMI se comuniquem com objetos CORBA.

### java.naming

Pacote que oferece suporte a serviços de nomes e diretório. Utilizado neste trabalho para registrar nomes de objetos RMI quando utilizado via IIOP.

### java.security

Contém classes e interfaces que representam certificados, chaves, assinaturas, etc. É necessário desenvolver ou adquirir as implementações que não são incluídas.

### java.security.acl

Interfaces de alto nível para a manipulação de listas de controle de acesso.

### java.security.interfaces

Interfaces básicas usadas pelo restante da API de segurança.

### java.sql

Classes e interfaces que representam conexões, declarações, conjuntos de resultados e drivers usados em um acesso a banco de dados relacional.

### java.text

Classes e interfaces usadas para internacionalização: formação de datas, representações decimais, mensagens textuais de acordo com parâmetros de um determinado local. Permite o desenvolvimento de aplicações internacionais.

### java.util.zip

Classes que computam somas de verificação em fluxos de dados, realizam compressão e arquivamento de fluxos de dados nos formatos ZIP e GZIP.

## 1.2. Como usar os pacotes da API

Todo programa importa automaticamente o caminho para as classes que compõem a biblioteca `java.lang` (obs: é necessário importar `java.lang.reflect` se precisar ser usada). Para usar as demais classes dentro de um programa, devem ser chamadas pelo nome completo (por exemplo: `java.awt.Button`, `java.util.Date`) ou pelo nome abreviado desde que haja, no início do programa, uma declaração `import`, da forma:

```
import <classe usada>;
```

Sintaxes típicas do uso de `import` são:

```
import java.awt.Image;
import java.io.*;
```

O primeiro, permite que se use a classe `Image` dentro do programa (sem precisar usar `java.awt.Image` em todos os lugares). Já o segundo uso, permite que se use todas as classes do pacote `java.io` na forma abreviada.

O compilador Java usa a variável de ambiente `CLASSPATH` para descobrir onde procurar as classes da API Java. Ela combina a informação do `CLASSPATH` com a da declaração `import` para localizar uma classe. Por exemplo: `import java.awt.Frame` procura a classe `Frame` em

```
$CLASSPATH/java/awt/Frame.class
```

Se `CLASSPATH` for, por exemplo, `C:\jdk1.1.3\lib\classes`, o caminho absoluto para localizar a classe `Frame` será:

```
C:\java\lib\classes\java\awt\Frame.class
```

## 1.3. A Biblioteca Fundamental Java (java.lang)

A biblioteca mais importante de toda a API é `java.lang`. Todas as classes de `java.lang` são automaticamente importadas sem a necessidade de uma declaração import. A seguir estão relacionadas algumas das mais importantes classes deste pacote (consulte a documentação para maiores detalhes e outras classes):

## Object

É a raiz de toda a hierarquia de classes em Java. Toda classe, existente ou não em Java, herda os parâmetros definidos em `Object`. É raro usar `Object` diretamente para criar instâncias. É mais comum usar `Object` para criar subclasses, quando não se estende outra classe, ou para usá-la como referência universal, como argumentos de métodos, construtores ou vetores.

## Class

Classe que representa classes. Contém métodos para carregar classes dinamicamente e obter informações sobre elas. É muito usada em programação genérica.

## Number, Character e Boolean

`Number` é uma classe abstrata que é superclasse de `Integer`, `Long`, `Byte`, `Short`, `Float` e `Double`, que juntamente com `Character`, `Void` e `Boolean` completam a coleção de classes *empacotadoras* de tipos primitivos. Como os tipos primitivos em Java não são objetos, eles se beneficiam destas classes quando precisam ser usados como objetos para passar valores por referência, serem incluídos em objetos `java.util.Hashtable`, `java.util.Vector` ou métodos que só recebem objetos.

`Integer` contém uma função (método estático) bastante usada para converter uma `String` em um inteiro: `int Integer.parseInt(String s)` recebe uma representação de número inteiro em formato `String` e devolve o valor inteiro. Os outros tipos não têm uma função assim e para fazer conversões é preciso *empacotá-los* em objetos para que se possa invocar métodos de conversão sobre eles. Abaixo estão alguns exemplos:

```
double d = new Double("12.783e-16").doubleValue();
float f = new Float("3.14159").floatValue();
long k = new Long("8516962").longValue();
boolean b = new Boolean("true").booleanValue();
char c = "B".charAt(0);
```

## Math

É uma classe final que define constantes para os valores matemáticos $\pi$ e $e$, além de definir um grande conjunto de funções matemáticas (métodos estáticos) para

a trigonometria, exponenciação e outras operações de ponto flutuante. Também contém métodos para calcular máximos e mínimos e gerar números pseudo-aleatórios. Exemplos:

```
double d = Math.sqrt(2);
int maior = Math(7, 9);
dado = (int)(Math.random() * 6)
```

## System e Runtime

`System` define três variáveis estáticas que representam a entrada padrão (`in`), a saída padrão (`out`) e a saída padrão de erro (`err`). Além disso, define métodos que oferecem uma interface independente de plataforma para funções do sistema. Com `System` é possível, por exemplo, executar uma aplicação externa e controlar o processo resultante, recuperar as fontes do sistema, etc. `Runtime` encapsula várias funções do sistema que são dependentes de plataforma, como a coleta de lixo, e contém vários métodos que são chamados por `System`.

## Process, Thread e Runnable

`Process` define uma interface independente de plataforma para processos que rodam externamente ao interpretador Java. `Thread` implementa suporte para múltiplas linhas de controle rodando no mesmo interpretador Java. Para criar uma linha de controle concorrente é preciso ou estender a classe `Thread` ou implementar a interface `Runnable` e passar o objeto resultante a um construtor do `Thread`. `Thread` define métodos para controle de prioridades, interrupção e agendamento de *threads*. `Runnable` declara um único método: `public void run()`, que deve ser implementado pelas classes que desejarem criar linhas de execução concorrentes.

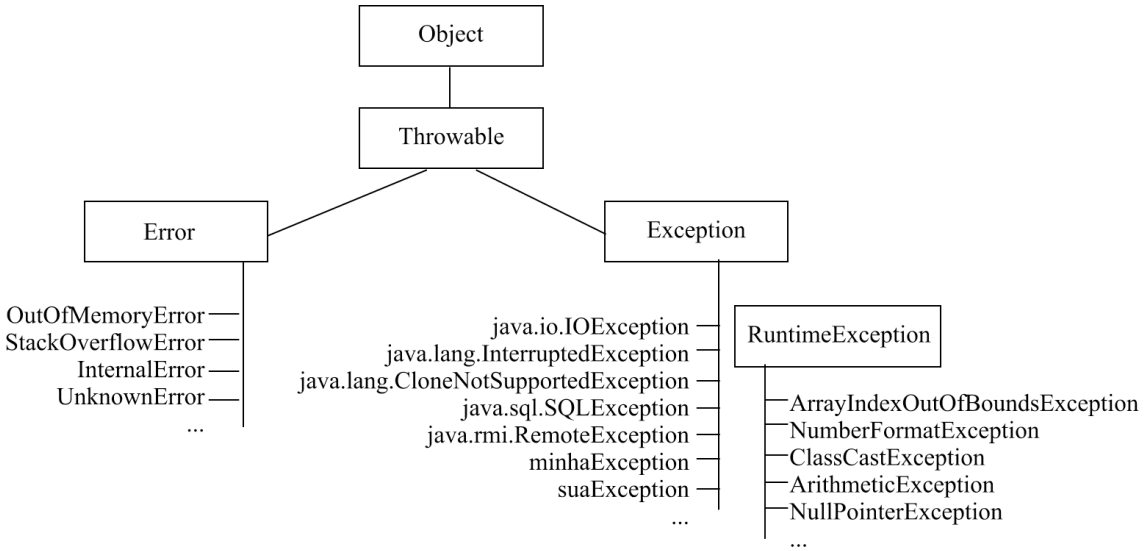
## Throwable, Error, Exception e suas subclasses

É a classe raiz da hierarquia de erros e exceções. Objetos `Throwable` são usados nas declarações `throw`, `catch` e `finally`. É pouco comum usar a classe `Throwable`. A classe `Error` é usada para definir erros dos quais não se espera recuperar, como por exemplo, a falta de memória. `Exception`, por sua vez, já serve para definir condições excepcionais que se espera ser possível corrigir sem abandonar a execução do programa, como, por exemplo, o fato de não conseguir encontrar um determinado arquivo.

A classe `Exception` ainda se divide em duas outras hierarquias importantes. A primeira são as classes derivadas diretamente de `Exception`, como as 7 exceções contidas no próprio pacote `java.lang`, várias outras definidas em outros pacotes, e as exceções definidas pelo usuário para representar condições excepcionais ocorridas em um programa que funciona corretamente (como uma conexão de rede indisponível, um arquivo não encontrado, uma URL digitada incorretamente, etc.). A segunda hierarquia é formada pelas classes derivadas de `RuntimeException`, que representa exceções que podem ocorrer durante o tempo de execução em um programa com *bugs*. As exceções deste tipo são avisos ao programador e ele deve corrigi-las.

A figura abaixo ilustra a hierarquia das exceções:




### Cloneable

É uma interface que não contém método algum. Serve para sinalizar que o objeto criado pela classe que implementa esta interface pode ser copiado usando o método `Object.clone()`.

## 1.4. String e StringBuffer

São objetos que representam cadeias de caracteres. `String` é um tipo não-modificável e `StringBuffer` pode ter seu conteúdo alterado.

`Strings` são objetos especiais na linguagem Java. Por ser uma das classes mais usadas, possui formas diferentes de para instanciar objetos e certas

restrições para garantir sua integridade. É a única classe em Java utilizada para representar cadeias de caracteres.

Diferentemente de C, uma cadeia de caracteres em Java não é um vetor de `chars` terminado em `\u0000`, mas um objeto. Os objetos do tipo `String`, uma vez criados, não podem ser mais modificados, ou seja, não se pode acrescentar um caractere ou remover outro. Não há métodos públicos que permitem tais modificações. Para modificar o conteúdo de um `String`, é necessário passá-lo como argumento para uma outra classe que manipule caracteres, como a classe `StringBuffer` e depois criar um novo `String` com o conteúdo modificado.

Há vários métodos para operar com `Strings`, os mais usados são:

- `public char charAt(int index)`: retorna o caractere no local indicado por `index` (a primeira posição é 0).
- `public String concat(String str)`: concatena o `String` atual com outro, retornando um novo `String` (mesmo que "+").
- `public boolean endsWith(String sufixo)`: retorna `true` se o `String` termina com o sufixo passado como parâmetro
- `public boolean startsWith(String prefixo)`: retorna `true` se o `String` começa com o prefixo passado como parâmetro.
- `public int indexOf(String str, int inicio)`: retorna o índice do início do `String str`, a partir da posição início. Este método também pode receber como primeiro argumento um char e não ter o segundo argumento, na versão com `char` ou na versão com `String`.
- `public int lastIndexOf(String str, int fim)`: faz o mesmo que `indexOf` só que de trás para frente. Também tem as mesmas variações sobrecarregadas que o método anterior. O fim não faz parte do `String`, que vai de `inicio` a `fim-1`.
- `public int length()`: retorna o comprimento do `String` em caracteres.
- `public String replace(char antigo, char novo)`: troca o caractere `antigo` por um `novo` e devolve um novo `String`.
- `public String substring(int início, int fim)`: retorna um novo `String` iniciando na posição início do `String` atual e terminando em `fim-1`.

- `public String toLowerCase()` e `public String toUpperCase()` retornam um novo `String` em caixa-alta ou caixa-baixa respectivamente.

Observe que todos os métodos que fazem alterações devolvem um novo objeto. A classe `String` produz objetos imutáveis.

A classe `StringBuffer` possui métodos que modificam os objetos do seu tipo, como `insert(int posição, String str)`, `append(String str)`, `reverse()`, `setCharAt(int posição, char caracter)` e `setLength(int novoTamanho)`. `StringBuffers` devem ser convertidos em `Strings` (usando `toString()` ou passando como construtor de um novo `String`) para poderem ser usados.

A concatenação de `Strings` é uma operação extremamente ineficiente. Para realizar concatenações, o sistema cria vários objetos que logo são descartados. Se você realizar muitas concatenções, o uso de `StringBuffer` é recomendado (use `append()`) pois esta última classe não cria novos objetos na concatenação. No final, você sempre pode converter tudo em `String`.

`Strings` podem ser criados da mesma forma como se cria objetos comuns:

```
String s = new String("I am a String!");
```

ou da forma mais usual:

```
String s = "I am a String!";
```

Ambos produzem o mesmo resultado só que o último tem um tratamento especial e é tratado mais como um tipo básico que como um `String`. Se você comparar usando "==" dois `Strings` criados usando `new`, eles serão apontados como diferentes (a referência foi comparada), mas se os dois tiverem sido criados usando a forma de atribuição, o sistema os considera iguais. O sistema dá este tratamento especial a `Strings` porque eles são imutáveis.

## Testes

1. (Roberts/Heller 97) Dado uma string construída usando `s = new String("xyzzy")`, quais das chamadas listadas abaixo modifica o string?
   A. `s.append("aaa");`
   B. `s.trim();`
   C. `s.substring(3);`
   D. `s.replace('z', 'a');`

E. `s.concat(s);`

## 1.5. Exceções

Quando acontece uma situação inesperada na execução de um programa, é necessário que exista uma rotina que lide com a situação e tente contorná-la se possível. Caso tal precaução não seja tomada, o programa deixará de funcionar como esperado, podendo fornecer resultados incorretos ou até abortar a execução.

Para desenvolver um código robusto, é necessário prever tais situações inesperadas e criar meios de contorná-las. Em todas as linguagens há maneiras de fazer isto. Pode ser tão simples quanto criar algumas expressões condicionais (por exemplo: testar se ocorrem divisões por zero) ou utilizando códigos de erro. Esta última alternativa é bastante usada em C e C++ e, apesar de aumentar a robustez do código, torna-o menos legível e mais complicado.

Para lidar com situações excepcionais Java proporciona uma forma limpa de verificar a ocorrência de erros e tratá-los se possível: as exceções.

O modelo de exceções já é adotado em outras linguagens, como o C++ (versões recentes) e Delphi. Em Java, seu uso é obrigatório em muitos casos como em programas que lidam com entrada e saída, conexões de rede, etc. A linguagem também permite que se defina novas exceções para lidar com situações particulares a um determinado programa.

Com as exceções, podemos programar sem preocupações com os erros que possam ocorrer em tempo de execução usando blocos `try-catch`. O trecho onde poderá haver erros é colocado dentro de um bloco `try { ... }` e depois dele, usamos um ou mais blocos `catch` para capturar exceções e fazer alguma coisa para lidar com elas.

Em muitos casos, o compilador *exige* que determinado bloco seja incluído em blocos `try... catch`, ou que seja declarado, na declaração do método, que tal exceção pode ocorrer. Isto é feito usando a palavra-chave `throws`:

```
public void m() throws Exception { ... }
```

Podemos também provocar uma exceção à força, usando `throw`:

```
try {
   (...)
   Exception e = new Exception();
   throw e;
```

```
    (...)
} catch (e) {
    /* ... */
}
```

Finalmente, podemos também agrupar várias coisas que devem ser feitas em caso de erro em um bloco `finally`. Se houver um erro, e o programa for abortar, antes de sair ele executa tudo o que for incluído no bloco `finally`.

A sintaxe básica para as exceções é a seguinte:

```
try { ... }
[catch (TipoDeExceção e1) {...}]
[catch (TipoDeExceção e2) {...}... ]
[finally { ... } ]
```

Pode haver zero ou mais blocos `catch`, para tratar exceções diferentes. O bloco `finally` também é opcional. Um `try`, porém, deve vir seguido de pelo menos um `catch` ou pelo `finally`.

Exceções são objetos. Quando acontece uma condição excepcional, o programa ou o sistema lança um objeto do tipo que representa a exceção ocorrida.

É fácil criar um programa que provoca exceções. Elas podem ser provocadas explicitamente (usando `throw`), de propósito, pelo programador para sinalizar uma condição excepcional, ou por métodos da API (métodos que declaram, com cláusula `throws`, que provocam exceções). As exceções abaixo estão entre as mais comuns:

- `ArrayIndexOutOfBoundsException` ocorre quando um vetor é acessado além dos seus limites inferior ou superior.
- `NumberFormatException` ocorre quando um `String` não pode ser convertido em número porque está no formato incorreto.

Ambas são subclasses de `Exception`, portanto, podem ser passadas para qualquer `catch` que pega `Exception`. Você sempre pode colocar um

```
catch (Exception e) { ... }
```

no final de todos os seus blocos `try {...}`. Isto irá pegar qualquer exceção, mas não irá pegar os erros (`Error`). Se você fizer isto, você não deve ter nenhum outro bloco `catch` com outra exceção, subclasse de `Exception`, depois deste senão o compilador reclamará dizendo que a declaração seguinte nunca será alcançada.

Isto acontecerá porque `Exception` a pegará primeiro. A ordem dos fatores importa.

Algumas outras exceções bastante comuns são:

- `NullPointerException` ocorre quando uma referência a um objeto ou vetor é usado sem ter um objeto correspondente ou em qualquer outro caso em que uma referência aponta para `null`.
- `SecurityException` é uma das mais comuns nos browsers. Ocorre quando um applet tenta violar as restrições de segurança do browser e tenta fazer algo ilegal como escrever em disco, por exemplo.
- `ArithmeticException` ocorre sempre que acontece uma divisão por zero em operações com inteiros. É a única exceção que ocorre nestes casos.

## Lançamento de exceções

Para lançar uma nova exceção, basta criar um novo objeto do tipo desejado e lança-lo com a palavra-chave `throw`.

```
throw new SuaException();
```

Depois de lançada a exceção, ela deve ou ser tratada no local ou propagada para cima na hierarquia, isto é, o método onde ela ocorre simplesmente declara que ela pode ocorrer e os outros métodos, que usam a classe e chamam esse método é que terão que lidar com a exceção.

Por exemplo, você poderia definir, em `Círculo`, o método:

```
public void setRaio(double raio) {
    if (raio <= 0) {
        throw new Exception();
   }
}
```

Ao compilar tal classe, o compilador reclama dizendo que: ou você trata esta exceção ou declara que `setRaio throws Exception`. A palavra-chave `throws` é usada apenas em declarações. Não a confunda com `throw`! Então fazemos

```
public void setRaio(double raio) throws Exception {
    if (raio <= 0) {
        throw new Exception();
   }
}
```

e o programa compila sem problemas. Mas depois, ao compilar o programa `Desenha`, que cria novos círculos e define novos raios, temos a mesma mensagem de erro porque o nosso método está chamando `setRaio`. Desta vez, optamos por tratar a exceção:

```
public static void main (String args[]) {
    try {
        Circulo c = new Circulo();
        c.setRaio(-1);
    } catch (Exception e) {
        try {
            c.setRaio(1);
        } catch (Exception e) {}
    }
}
```

## Criação de novas exceções

A exceção `Exception` no exemplo acima nada diz sobre o tipo de exceção que ocorreu. Neste caso, o mais interessante é ter uma exceção especial que caracterize o problema. Para criar novas classes de exceções, devemos estender a classe `Exception`:

```
class RaioNegativoException extends Exception {}
```

e isto é tudo o que é necessário fazer. Podemos definir outros construtores para poder passar mensagens, outros métodos, qualquer coisa que uma classe pode ter, uma Exceção pode ter, mas a sintaxe acima já resolve tudo. Agora declaramos:

```
public void setRaio(double raio) throws RaioNegativoException {
    if (raio <= 0) {
        throw new RaioNegativoException();
   }
}
```

E agora o programa que cria círculos pode saber quando acontece uma exceção deste tipo e tratar de acordo, usando um `catch` específico só para este tipo de exceção.

## Testes

Os testes a seguir são do livro "Java 1.1 Certification Study Guide" de Roberts/Heller.

1. Considere a seguinte hierarquia de classes e fragmentos de código:

```
                    java.lang.Exception
                            \
                       java.io.IOException
                        /                \
java.io.StreamCorruptedException    java. net.MalformedURLException
```

```
 1. try {
 2.   URL u = new URL(s); // assume s is a previously defined String
 3.   Object o = in.readObject(); // in is a valid ObjectInputStream
 4.   System.out.println("Success");
 5. }
 6. catch (MalformedURLException e) {
 7.   System.out.println("Bad URL");
 8. }
 9. catch (StreamCorruptedException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Que linhas são impressas se os métodos das linhas 2 e 3 completam com sucesso sem provocar exceções?

A. Success
B. Bad URL
C. Bad File Contents
D. General Exception
E. doing finally part
F. Carrying on
G.

2. Considere a seguinte hierarquia de classes e fragmentos de código:

```
            java.lang.Throwable
              /              \
    java.lang.Error        java.lang.Exception
```

```
                /                       \
  java.lang.OutOfMemoryError    java.io.IOException
                                /                 \
     java.io.StreamCorruptedException    java.net.MalformedURLException
```

```
 1. try {
 2.   URL u = new URL(s); // assume s is a previously defined String
 3.   Object o = in.readObject(); // in is a valid ObjectInputStream
 4.   System.out.println("Success");
 5. }
 6. catch (MalformedURLException e) {
 7.   System.out.println("Bad URL");
 8. }
 9. catch (StreamCorruptedException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Que linhas são impressas que a linha 3 provoca um OutOfMemoryError?

A. Success
B. Bad URL
C. Bad File Contents
D. General Exception
E. doing finally part
F. Carrying on

3. Considere a seguinte hierarquia de classes e fragmentos de código:

```
                        java.lang.Exception
                                \
                          java.io.IOException
                           /                \
  java.io.StreamCorruptedException    java. net.MalformedURLException
```

```
1. try {
2.   URL u = new URL(s); // assume s is a previously defined String
3.   Object o = in.readObject(); // in is a valid ObjectInputStream
4.   System.out.println("Success");
5. }
6. catch (MalformedURLException e) {
```

```
 7.   System.out.println("Bad URL");
 8. }
 9. catch (StreamCorruptedException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Que linhas são impressas se o método na linha 2 provoca uma MalformedURLException?

A. Success
B. Bad URL
C. Bad File Contents
D. General Exception
E. doing finally part

F. Carrying on