

# Java - Guia Rápido

## Anúncios

[Página anterior](#)[Próxima página](#)

## Java - Visão geral

A linguagem de programação Java foi originalmente desenvolvida pela Sun Microsystems e foi iniciada por James Gosling e lançada em 1995 como componente principal da plataforma Java da Sun Microsystems (Java 1.0 [J2SE]).

A versão mais recente do Java Standard Edition é Java SE 8. Com o avanço do Java e sua ampla popularidade, várias configurações foram criadas para atender a vários tipos de plataformas. Por exemplo: J2EE para Aplicativos Corporativos, J2ME para Aplicativos Móveis.

As novas versões do J2 foram renomeadas como Java SE, Java EE e Java ME, respectivamente. Java é garantido como **Write Once, Run Anywhere**.

Java é -

**Orientado a Objetos** - Em Java, tudo é um Objeto. Java pode ser facilmente estendido, uma vez que é baseado no modelo de objeto.

**Independente de plataforma** - Ao contrário de muitas outras linguagens de programação, incluindo C e C ++, quando o Java é compilado, ele não é compilado em uma máquina específica da plataforma, e sim no código de byte independente da plataforma. Esse código de byte é distribuído pela web e interpretado pela Máquina Virtual (JVM) na plataforma em que está sendo executada.

**Simples** - Java é projetado para ser fácil de aprender. Se você entender o conceito básico de OOP Java, seria fácil dominar.

**Seguro** - Com o recurso seguro do Java, ele permite desenvolver sistemas livres de vírus e livres de violação. As técnicas de autenticação são baseadas em criptografia de chave pública.

**Arquitetura neutra** - O compilador Java gera um formato de arquivo objeto neutro de arquitetura, o que torna o código compilado executável em muitos processadores, com a presença do sistema de tempo de execução Java.

**Portátil** - Sendo neutro quanto à arquitetura e não tendo nenhum aspecto dependente de implementação da especificação, o Java portable. O compilador em Java é escrito em ANSI C com um limite de portabilidade limpo, que é um subconjunto POSIX.

**Robusto** - Java faz um esforço para eliminar situações propensas a erros, enfatizando principalmente a verificação de erros de tempo de compilação e

verificação de tempo de execução.

**Multithreaded** - Com o recurso multithreadado do Java, é possível escrever programas que podem executar muitas tarefas simultaneamente. Esse recurso de design permite que os desenvolvedores construam aplicativos interativos que podem funcionar sem problemas.

**Interpretado** - O código de byte Java é traduzido rapidamente para as instruções da máquina nativa e não é armazenado em nenhum lugar. O processo de desenvolvimento é mais rápido e analítico, pois a vinculação é um processo incremental e leve.

**Alto desempenho** - Com o uso de compiladores Just-In-Time, o Java permite alto desempenho.

**Distribuído** - Java é projetado para o ambiente distribuído da internet.

**Dinâmico** - Java é considerado mais dinâmico que C ou C ++, pois foi projetado para se adaptar a um ambiente em evolução. Os programas Java podem transportar uma quantidade extensa de informações em tempo de execução que podem ser usadas para verificar e resolver acessos a objetos em tempo de execução.

James Gosling iniciou o projeto de linguagem Java em junho de 1991 para <sup>História do Java</sup> uso em um de seus muitos projetos de set-top box. A linguagem, inicialmente chamada de "Carvalho", depois de um carvalho que ficava do lado de fora do escritório de Gosling, também era chamada "Verde" e acabou sendo renomeada como Java, a partir de uma lista de palavras aleatórias.

A Sun lançou a primeira implementação pública como Java 1.0 em 1995. Ela prometia **Write Once, Run Anywhere** (WORA), proporcionando tempos de execução sem custo em plataformas populares.

Em 13 de novembro de 2006, a Sun lançou grande parte do Java como software livre e de código aberto sob os termos da GNU General Public License (GPL).

Em 8 de maio de 2007, a Sun concluiu o processo, tornando todo o código principal do Java livre e de código aberto, além de uma pequena parte do código para o qual a Sun não detinha os direitos autorais.

Para executar os exemplos discutidos neste tutorial, você <sup>Ferramentas que você precisará</sup> precisará de um computador Pentium 200 MHz com um mínimo de 64 MB de RAM (128 MB de RAM recomendados).

Você também precisará dos seguintes softwares -

Sistema operacional Linux 7.1 ou Windows xp / 7/8

Java JDK 8

Bloco de notas da Microsoft ou qualquer outro editor de texto

Este tutorial fornecerá as habilidades necessárias para criar GUI, rede e aplicativos da Web

usando Java.

O próximo capítulo irá guiá-lo para como você pode obter o Java e sua documentação. Finalmente, ele instrui sobre como instalar o Java e preparar um ambiente para desenvolver aplicativos Java.

## Java - Configuração do Ambiente

Neste capítulo, discutiremos os diferentes aspectos da configuração de um ambiente agradável para Java.

Se você ainda estiver disposto a configurar seu ambiente para a linguagem de programação Java, esta seção orientará você sobre como fazer o download e configurar o Java em sua máquina. A seguir estão as etapas para configurar o ambiente.

O Java SE está disponível gratuitamente no link [Download Java](#). Você pode baixar uma versão baseada em seu sistema operacional.

Siga as instruções para baixar o Java e execute o **.exe** para instalar o Java em sua máquina. Depois de instalar o Java em sua máquina, você precisará configurar variáveis de ambiente para apontar para diretórios de instalação corretos -

### Configurando o caminho para o Windows

Supondo que você tenha instalado o Java no diretório *c: \ Arquivos de Programas \ java \ jdk* -

Clique com o botão direito em 'Meu Computador' e selecione 'Propriedades'.

Clique no botão 'Variáveis de ambiente' na guia 'Avançado'.

Agora, altere a variável 'Path' para que também contenha o caminho para o executável Java. Exemplo, se o caminho estiver atualmente definido como 'C: \ WINDOWS \ SYSTEM32', altere o caminho para ler 'C: \ WINDOWS \ SYSTEM32; c: \ Arquivos de Programas \ java \ jdk \ bin'.

### Configurando o caminho para Linux, UNIX, Solaris, FreeBSD

A variável de ambiente PATH deve ser definida para apontar para onde os binários Java foram instalados. Consulte a documentação do seu shell, se você tiver problemas para fazer isso.

Por exemplo, se você usar *bash* como seu shell, adicione a seguinte linha ao final de seu '.bashrc: export PATH = / path / to / java: \$ PATH'

Para escrever seus programas Java, você precisará de um editor de texto. Existem IDEs ainda mais sofisticados disponíveis no mercado. Mas por enquanto, você pode considerar um dos seguintes -

**Bloco de Notas** - Na máquina Windows, você pode usar qualquer editor de texto simples como o Bloco de Notas (Recomendado para este tutorial), o TextPad.

**Netbeans** - Um IDE Java de código aberto e gratuito que pode ser baixado em

<https://www.netbeans.org/index.html> .

**Eclipse** - Um IDE Java desenvolvido pela comunidade de software livre do Eclipse e pode ser baixado em <https://www.eclipse.org/> .

O próximo capítulo ensinará como escrever e executar seu primeiro programa Java e algumas das importantes sintaxes básicas do Java necessárias para o desenvolvimento de aplicativos. O que é o próximo?

## Java - Sintaxe Básica

Quando consideramos um programa Java, ele pode ser definido como uma coleção de objetos que se comunicam por meio da invocação dos métodos um do outro. Vamos agora examinar brevemente o que significa classe, objeto, métodos e variáveis de instância.

**Objeto** - Objetos possuem estados e comportamentos. Exemplo: Um cão tem estados - cor, nome, raça e comportamento, como abanar o rabo, latir, comer. Um objeto é uma instância de uma classe.

**Classe** - Uma classe pode ser definida como um template / blueprint que descreve o comportamento / estado que o objeto de seu tipo suporta.

**Métodos** - Um método é basicamente um comportamento. Uma classe pode conter muitos métodos. É nos métodos onde as lógicas são escritas, os dados são manipulados e todas as ações são executadas.

**Variáveis de instância** - cada objeto tem seu conjunto exclusivo de variáveis de instância. O estado de um objeto é criado pelos valores atribuídos a essas variáveis de instância.

Vamos olhar para um código simples que irá imprimir as primeiras palavras **Hello World** .

### Exemplo

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     */  
  
    public static void main(String []args) {  
        System.out.println("Hello World"); // prints Hello World  
    }  
}
```

Vamos ver como salvar o arquivo, compilar e executar o programa. Por favor, siga os passos subsequentes -

Abra o bloco de notas e adicione o código como acima.

Salve o arquivo como: MyFirstJavaProgram.java.

Abra uma janela de prompt de comando e vá para o diretório em que você salvou a classe. Suponha que seja C: \.

Digite 'javac MyFirstJavaProgram.java' e pressione enter para compilar seu código. Se não houver erros no seu código, o prompt de comando levará você para a próxima linha (Suposição: A variável de caminho está definida).

Agora, digite 'java MyFirstJavaProgram' para executar seu programa.

Você poderá ver "Hello World" impresso na janela.

## Saída

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
Hello World
```

Sobre os programas Java, é muito importante ter em mente os seguintes pontos.

**Case Sensitivity** - Java diferencia maiúsculas de minúsculas, o que significa que o identificador **Hello** e **hello** teria um significado diferente em Java.

**Nomes de classe** - Para todos os nomes de classes, a primeira letra deve estar em maiúsculas. Se várias palavras forem usadas para formar um nome da classe, a primeira letra de cada palavra interna deverá estar em maiúscula.

**Exemplo:** *class MyFirstJavaClass*

**Nomes de Métodos** - Todos os nomes de métodos devem começar com uma letra minúscula. Se várias palavras forem usadas para formar o nome do método, a primeira letra de cada palavra interna deverá estar em maiúscula.

**Exemplo:** *public void myMethodName ()*

**Nome do arquivo de programa** - o nome do arquivo de programa deve corresponder exatamente ao nome da classe.

Ao salvar o arquivo, salve-o usando o nome da classe (Lembre-se de que o Java faz distinção entre maiúsculas e minúsculas) e acrescente '.java' ao final do nome (se o nome do arquivo e o nome da classe não corresponderem, seu programa não compilará ).

**Exemplo:** Suponha que 'MyFirstJavaProgram' seja o nome da classe. Em seguida, o arquivo deve ser salvo como '*MyFirstJavaProgram.java*'

**public static void main (String args [])** - O processamento de programas em Java é iniciado a partir do método main (), que é uma parte obrigatória de todo programa Java.

Todos os componentes Java exigem nomes. Nomes usados para identificadores Java classes, variáveis e métodos são chamados de **identificadores** .

Em Java, há vários pontos para lembrar sobre identificadores. Eles são os seguintes -

Todos os identificadores devem começar com uma letra (A a Z ou a a z), caractere de moeda (\$) ou sublinhado (\_).

Após o primeiro caractere, os identificadores podem ter qualquer combinação de caracteres.

Uma palavra-chave não pode ser usada como um identificador.

Mais importante ainda, os identificadores diferenciam maiúsculas de minúsculas.

Exemplos de identificadores legais: idade, \$ salário, \_valor, \_\_1\_valor.

Exemplos de identificadores ilegais: 123abc, -salary.

Como outras linguagens, é possível modificar classes, métodos, etc., Modificadores Java usando modificadores. Existem duas categorias de modificadores -

**Modificadores de acesso** - padrão, público, protegido, privado

**Modificadores sem acesso** - final, resume, strictfp

Nós veremos mais detalhes sobre os modificadores na próxima seção.

A seguir estão os tipos de variáveis em Java -

Variáveis Java

Variáveis Locais

Variáveis de Classe (Variáveis Estáticas)

Variáveis de instância (variáveis não estáticas)

Matrizes são objetos que armazenam múltiplas variáveis do mesmo tipo. No entanto, um array em si é um objeto no heap. Veremos como declarar, construir e inicializar nos próximos capítulos.

Enums foram introduzidos no Java 5.0. As enumerações restringem uma variável para ter um de apenas alguns valores predefinidos. Os valores nesta lista enumerada são chamados de enums.

Com o uso de enums, é possível reduzir o número de erros no seu código.

Por exemplo, se considerarmos uma aplicação para uma loja de suco fresco, seria possível restringir o tamanho do vidro para pequeno, médio e grande. Isso garantiria que não permitiria que alguém encomendasse qualquer tamanho diferente de pequeno, médio ou grande.

## Exemplo

```
class FreshJuice {
    enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
    FreshJuiceSize size;
}

public class FreshJuiceTest {

    public static void main(String args[]) {
        FreshJuice juice = new FreshJuice();
        juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
        System.out.println("Size: " + juice.size);
    }
}
```

O exemplo acima irá produzir o seguinte resultado -

## Saída

Size: MEDIUM

**Nota** - As enumerações podem ser declaradas como próprias ou dentro de uma classe. Métodos, variáveis, construtores podem ser definidos dentro de enums também.

A lista a seguir mostra as palavras reservadas em Java. Essas palavras-chave Java palavras reservadas não podem ser usadas como constantes ou variáveis ou quaisquer outros nomes de identificadores.

abstrato	afirmar	booleano	pausa
byte	caso	pegar	Caracteres
classe	const	continuar	padrão
Faz	Duplo	outro	enum
estende	final	finalmente	flutuador
para	vamos para	E se	implementa
importar	instancia de	int	interface
longo	nativo	Novo	pacote
privado	protegido	público	Retorna
curto	estático	strictfp	super
interruptor	sincronizado	esta	lançar
lança	transiente	experimental	vazio
volátil	enquanto		

Java suporta comentários de linha única e multilinha muito semelhantes a C e C ++. Todos os caracteres disponíveis dentro de qualquer comentário são ignorados pelo compilador Java.

## Exemplo

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     * This is an example of multi-line comments.  
     */  
  
    public static void main(String []args) {  
        // This is an example of single line comment  
        /* This is also an example of single line comment. */  
        System.out.println("Hello World");  
    }  
}
```

```
}
```

## Saída

```
Hello World
```

Uma linha contendo apenas espaço em branco, possivelmente com um comentário, é conhecida como uma linha em branco, e o Java a ignora totalmente.

Em Java, as classes podem ser derivadas de classes. Basicamente, se você precisa criar uma nova classe e aqui já é uma classe que tem algum código que você precisa, então é possível derivar sua nova classe do código já existente.

Esse conceito permite reutilizar os campos e métodos da classe existente sem precisar reescrever o código em uma nova classe. Nesse cenário, a classe existente é chamada de **superclasse** e a classe derivada é chamada de **subclasse**.

Na linguagem Java, uma interface pode ser definida como um contrato entre objetos sobre como se comunicar uns com os outros. Interfaces desempenham um papel vital quando se trata do conceito de herança.

Uma interface define os métodos, uma classe derivada (subclasse) deve usar. Mas a implementação dos métodos depende totalmente da subclasse.

A próxima seção explica sobre objetos e classes na programação Java. No final da sessão, você será capaz de obter uma imagem clara sobre o que são objetos e quais são as classes em Java.

# Java - Objeto e Classes

Java é uma linguagem orientada a objetos. Como uma linguagem que possui o recurso Orientado a Objetos, o Java suporta os seguintes conceitos fundamentais -

Polimorfismo

Herança

Encapsulamento

Abstração

Classes

Objetos

Instância

Método

Análise de Mensagem

Neste capítulo, vamos examinar os conceitos - Classes e Objetos.

**Objeto** - Objetos possuem estados e comportamentos. Exemplo: Um cão tem estados - cor, nome, raça, bem como comportamentos - abanando o rabo, latindo, comendo. Um objeto é uma instância de uma classe.



**Classe** - Uma classe pode ser definida como um template / blueprint que descreve o comportamento / estado que o objeto de seu tipo suporta.

Vamos agora olhar profundamente para o que são objetos. Se considerarmos o mundo real, podemos encontrar muitos objetos ao nosso redor, carros, cães, seres humanos, etc. Todos esses objetos têm um estado e um comportamento.

Se considerarmos um cão, então seu estado é - nome, raça, cor e o comportamento é - latir, abanar o rabo, correr.

Se você comparar o objeto de software com um objeto do mundo real, eles terão características muito semelhantes.

Objetos de software também possuem um estado e um comportamento. O estado de um objeto de software é armazenado em campos e o comportamento é mostrado por meio de métodos.

Assim, no desenvolvimento de software, os métodos operam no estado interno de um objeto e a comunicação objeto-a-objeto é feita por meio de métodos.

Uma classe é um plano a partir do qual objetos individuais são criados. Classes em Java

A seguir, uma amostra de uma aula.

## Exemplo

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
  
    void hungry() {  
    }  
  
    void sleeping() {  
    }  
}
```

Uma classe pode conter qualquer um dos seguintes tipos de variáveis.

**Variáveis locais** - Variáveis definidas dentro de métodos, construtores ou blocos são chamadas de variáveis locais. A variável será declarada e inicializada dentro do método e a variável será destruída quando o método for concluído.

**Variáveis de instância** - **variáveis de** instância são variáveis dentro de uma classe, mas fora de qualquer método. Essas variáveis são inicializadas quando a classe é instanciada. Variáveis de instância podem ser acessadas de dentro de qualquer método, construtor ou blocos dessa classe em particular.

**Variáveis de classe** - **Variáveis de** classe são variáveis declaradas dentro de uma classe, fora de qualquer método, com a palavra-chave estática.

Uma classe pode ter qualquer número de métodos para acessar o valor de vários tipos de métodos. No exemplo acima, `barking ()`, `hungry ()` e `sleeping ()` são métodos.

A seguir, alguns dos tópicos importantes que precisam ser discutidos ao examinar classes da linguagem Java.

Ao discutir sobre classes, um dos subtópicos mais importantes seria Construtores construtores. Toda classe tem um construtor. Se não escrevermos explicitamente um construtor para uma classe, o compilador Java construirá um construtor padrão para essa classe.

Cada vez que um novo objeto é criado, pelo menos um construtor será invocado. A principal regra dos construtores é que eles devem ter o mesmo nome da classe. Uma classe pode ter mais de um construtor.

A seguir, um exemplo de um construtor -

## Exemplo

```
public class Puppy {  
    public Puppy() {  
    }  
  
    public Puppy(String name) {  
        // This constructor has one parameter, name .  
    }  
}
```

Java também suporta Singleton Classes, onde você seria capaz de criar apenas uma instância de uma classe.

**Nota** - Temos dois tipos diferentes de construtores. Vamos discutir construtores detalhadamente nos capítulos subsequentes.

Como mencionado anteriormente, uma classe fornece os esquemas Criando um Objeto para objetos. Então, basicamente, um objeto é criado a partir de uma classe. Em Java, a nova palavra-chave é usada para criar novos objetos.

Existem três etapas ao criar um objeto de uma classe -

**Declaração** - Uma declaração de variável com um nome de variável com um tipo de objeto.

**Instantiation** - A palavra-chave 'new' é usada para criar o objeto.

**Inicialização** - A palavra-chave 'new' é seguida por uma chamada para um construtor. Esta chamada inicializa o novo objeto.

A seguir, um exemplo de criação de um objeto

## Exemplo

```
public class Puppy {  
    public Puppy(String name) {  
        // This constructor has one parameter, name .  
        System.out.println("Passed Name is :" + name );  
    }  
}
```

```

    }

    public static void main(String []args) {
        // Following statement would create an object myPuppy
        Puppy myPuppy = new Puppy( "tommy" );
    }
}

```

Se compilarmos e executarmos o programa acima, ele produzirá o seguinte resultado -

## Saída

```
Passed Name is :tommy
```

Variáveis e métodos de instância são acessados Acessando Variáveis e Métodos da Instância através de objetos criados. Para acessar uma variável de instância, seguir é o caminho completo -

```

/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();

```

## Exemplo

Este exemplo explica como acessar variáveis de instância e métodos de uma classe.

```

public class Puppy {
    int puppyAge;

    public Puppy(String name) {
        // This constructor has one parameter, name .
        System.out.println("Name chosen is :" + name );
    }

    public void setAge( int age ) {
        puppyAge = age;
    }

    public int getAge( ) {
        System.out.println("Puppy's age is :" + puppyAge );
        return puppyAge;
    }

    public static void main(String []args) {
        /* Object creation */
        Puppy myPuppy = new Puppy( "tommy" );

        /* Call class method to set puppy's age */
        myPuppy.setAge( 2 );

        /* Call another class method to get puppy's age */
        myPuppy.getAge( );

        /* You can access instance variable as follows as well */
        System.out.println("Variable Value :" + myPuppy.puppyAge );
    }
}

```

```
}  
}
```

Se compilarmos e executarmos o programa acima, ele produzirá o seguinte resultado -

## Saída

```
Name chosen is :tommy  
Puppy's age is :2  
Variable Value :2
```

Como a última parte desta seção, vamos agoraRegras de declaração de arquivo de origem examinar as regras de declaração do arquivo fonte. Essas regras são essenciais ao declarar classes, instruções de *importação* e instruções de *pacote* em um arquivo de origem.

Pode haver apenas uma classe pública por arquivo de origem.

Um arquivo de origem pode ter várias classes não públicas.

O nome da classe pública também deve ser o nome do arquivo de origem, que deve ser anexado por **.java** no final. Por exemplo: o nome da classe é *public class Employee {}*, então o arquivo de origem deve ser Employee.java.

Se a classe estiver definida dentro de um pacote, a instrução package deverá ser a primeira instrução no arquivo de origem.

Se as instruções de importação estiverem presentes, elas deverão ser gravadas entre a declaração do pacote e a declaração da classe. Se não houver instruções de pacote, a instrução de importação deve ser a primeira linha no arquivo de origem.

As declarações de importação e pacote implicarão em todas as classes presentes no arquivo de origem. Não é possível declarar diferentes instruções de importação e / ou pacote para classes diferentes no arquivo de origem.

As classes têm vários níveis de acesso e existem diferentes tipos de classes; classes abstratas, classes finais, etc. Nós explicaremos tudo isso no capítulo de modificadores de acesso.

Além dos tipos de classes mencionados acima, o Java também possui algumas classes especiais chamadas Classes internas e Anônimas.

Em palavras simples, é uma maneira de categorizar as classes e interfaces.Pacote Java Ao desenvolver aplicativos em Java, centenas de classes e interfaces serão gravadas, portanto, categorizar essas classes é essencial, além de tornar a vida muito mais fácil.

Em Java, se um nome completo, que inclui o pacote e o nome daDeclarações de Importação classe, for fornecido, o compilador poderá localizar facilmente o código-fonte ou as classes. A instrução de importação é uma maneira de fornecer o local adequado para o compilador localizar essa classe específica.

Por exemplo, a linha a seguir pediria ao compilador para carregar todas as classes disponíveis no diretório `java_installation / java / io` -

```
import java.io.*;
```

Para nosso estudo de caso, criaremos duas classes. Eles são Um estudo de caso simples `Employee` e `EmployeeTest`.

Primeiro abra o bloco de notas e adicione o seguinte código. Lembre-se que esta é a classe `Employee` e a classe é uma classe pública. Agora, salve esse arquivo de origem com o nome `Employee.java`.

A classe `Employee` possui quatro variáveis de instância - nome, idade, designação e salário. A classe tem um construtor explicitamente definido, que recebe um parâmetro.

## Exemplo

```
import java.io.*;
public class Employee {

    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name) {
        this.name = name;
    }

    // Assign the age of the Employee to the variable age.
    public void empAge(int empAge) {
        age = empAge;
    }

    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig) {
        designation = empDesig;
    }

    /* Assign the salary to the variable salary.*/
    public void empSalary(double empSalary) {
        salary = empSalary;
    }

    /* Print the Employee details */
    public void printEmployee() {
        System.out.println("Name:" + name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}
```

Conforme mencionado anteriormente neste tutorial, o processamento inicia a partir do método principal. Portanto, para que possamos executar esta classe `Employee`, deve haver um método `main` e os objetos devem ser criados. Nós estaremos criando uma classe separada para essas tarefas.

A seguir está a classe `EmployeeTest` , que cria duas instâncias da classe `Employee` e

chama os métodos para cada objeto para atribuir valores a cada variável.

Salve o seguinte código no arquivo EmployeeTest.java.

```
import java.io.*;
public class EmployeeTest {

    public static void main(String args[]) {
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
        Employee empTwo = new Employee("Mary Anne");

        // Invoking methods for each object created
        empOne.empAge(26);
        empOne.empDesignation("Senior Software Engineer");
        empOne.empSalary(1000);
        empOne.printEmployee();

        empTwo.empAge(21);
        empTwo.empDesignation("Software Engineer");
        empTwo.empSalary(500);
        empTwo.printEmployee();
    }
}
```

Agora, compile as classes e execute o *EmployeeTest* para ver o resultado da seguinte forma:

## Saída

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

Na próxima sessão, discutiremos os tipos básicos de dados em Java e o que é o próximo? como eles podem ser usados no desenvolvimento de aplicativos Java.

## Java - Tipos de Dados Básicos

Variáveis não são nada além de locais de memória reservados para armazenar valores. Isto significa que quando você cria uma variável você reserva algum espaço na memória.

Com base no tipo de dados de uma variável, o sistema operacional aloca memória e decide o que pode ser armazenado na memória reservada. Portanto, atribuindo diferentes tipos de dados a variáveis, você pode armazenar números inteiros, decimais ou caracteres nessas variáveis.

Existem dois tipos de dados disponíveis em Java -

Tipos de dados primitivos

Tipos de dados de referência / objeto

Existem oito tipos de dados primitivos suportados pelo Java. Os tipos de dados primitivos são predefinidos pela linguagem e nomeados por uma palavra-chave. Vamos agora examinar os oito tipos de dados primitivos em detalhes.

## byte

O tipo de dados Byte é um inteiro de complemento de dois bits assinado de 8 bits

O valor mínimo é -128 ( $-2^7$ )

O valor máximo é 127 (inclusive) ( $2^7 - 1$ )

O valor padrão é 0

Byte data type é usado para economizar espaço em grandes matrizes, principalmente no lugar de inteiros, uma vez que um byte é quatro vezes menor que um inteiro.

Exemplo: byte a = 100, byte b = -50

## curto

Tipo de dados curto é um número inteiro de complemento de dois de 16 bits assinado

O valor mínimo é -32.768 ( $-2^{15}$ )

O valor máximo é 32.767 (inclusive) ( $2^{15} - 1$ )

O tipo de dados curto também pode ser usado para economizar memória como tipo de dados de byte. Um curto é 2 vezes menor que um inteiro

O valor padrão é 0.

Exemplo: short s = 10000, short r = -20000

## int

O tipo de dados Int é um inteiro de complemento de dois de 32 bits assinado.

O valor mínimo é - 2.147.483.648 ( $-2^{31}$ )

O valor máximo é 2.147.483.647 (inclusive) ( $2^{31} - 1$ )

Inteiro é geralmente usado como o tipo de dados padrão para valores integrais, a menos que haja uma preocupação com a memória.

O valor padrão é 0

Exemplo: int a = 100000, int b = -200000

## longo

Tipo de dados longo é um número inteiro de complemento de dois de 64 bits assinado

O valor mínimo é -9,223,372,036,854,775,808 ( $-2^{63}$ )

O valor máximo é 9.223.372.036.854.775.807 (inclusive) ( $2^{63} - 1$ )

Este tipo é usado quando um intervalo maior que int é necessário

O valor padrão é 0L

Exemplo: long a = 100000L, long b = -200000L

## flutuador

Tipo de dados flutuante é um ponto flutuante IEEE 754 de precisão simples de 32 bits

O Float é usado principalmente para economizar memória em grandes matrizes de números de ponto flutuante

O valor padrão é 0,0f

O tipo de dados flutuante nunca é usado para valores precisos, como moeda

Exemplo: float f1 = 234.5f

## Duplo

tipo de dados duplo é um ponto flutuante IEEE 754 de precisão dupla de 64 bits

Esse tipo de dados geralmente é usado como o tipo de dados padrão para valores decimais, geralmente a opção padrão

O tipo de dados duplo nunca deve ser usado para valores precisos, como moeda

O valor padrão é 0,0d

Exemplo: double d1 = 123,4

## booleano

tipo de dados booleano representa um bit de informação

Existem apenas dois valores possíveis: verdadeiro e falso

Esse tipo de dados é usado para sinalizadores simples que rastreiam condições verdadeiras / falsas

O valor padrão é falso

Exemplo: boolean one = true

## Caracteres

tipo de dados char é um único caractere Unicode de 16 bits

O valor mínimo é '\u0000' (ou 0)

O valor máximo é '\uffff' (ou 65.535 inclusive)

O tipo de dados char é usado para armazenar qualquer caractere



Exemplo: `char letterA = 'A'`

Variáveis de referência são criadas usando construtores. Tipos de dados de referência definidos das classes. Eles são usados para acessar objetos. Essas variáveis são declaradas como sendo de um tipo específico que não pode ser alterado. Por exemplo, `funcionário`, `filhote`, etc.

Objetos de classe e vários tipos de variáveis de matriz estão sob o tipo de dados de referência.

O valor padrão de qualquer variável de referência é nulo.

Uma variável de referência pode ser usada para referenciar qualquer objeto do tipo declarado ou qualquer tipo compatível.

Exemplo: `animal animal = novo animal ("girafa");`

Um literal é uma representação do código-fonte de um valor fixo. Eles são representados diretamente no código sem qualquer cálculo.

Literais podem ser atribuídos a qualquer variável de tipo primitivo. Por exemplo -

```
byte a = 68;
char a = 'A';
```

`byte`, `int`, `long` e `short` podem ser expressos em sistemas numéricos decimal (base 10), hexadecimal (base 16) ou octal (base 8).

O prefixo `0` é usado para indicar octal e o prefixo `0x` indica hexadecimal ao usar esses sistemas numéricos para literais. Por exemplo -

```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```

Os literais de string em Java são especificados como na maioria das outras linguagens, incluindo uma sequência de caracteres entre um par de aspas duplas. Exemplos de literais de string são -

## Exemplo

```
"Hello World"
"two\nlines"
"\\"This is in quotes\\""
```

Os tipos de literais string e char podem conter qualquer caractere Unicode. Por exemplo -

```
char a = '\u0001';
String a = "\u0001";
```

A linguagem Java suporta algumas sequências de escape especiais para literais de String e de caracteres também. Eles são

Notação	Personagem representado
---------	-------------------------

\ n	Newline (0x0a)
\ r	Retorno de carro (0x0d)
\ f	Formfeed (0x0c)
\ b	Backspace (0x08)
\ s	Espaço (0x20)
\ t	aba
\ "	Citação dupla
\ '	Citação única
\\	barra invertida
\ ddd	Caractere octal (ddd)
\ uxxxx	Caractere UNICODE hexadecimal (xxxx)

Este capítulo explicou os vários tipos de dados. O próximo tópico <sup>O</sup> que é o próximo? explica diferentes tipos de variáveis e seu uso. Isso lhe dará um bom entendimento de como eles podem ser usados nas classes, interfaces, etc. do Java.

## Java - Tipos Variáveis

Uma variável nos fornece armazenamento nomeado que nossos programas podem manipular. Cada variável em Java possui um tipo específico, que determina o tamanho e o layout da memória da variável; o intervalo de valores que podem ser armazenados nessa memória; e o conjunto de operações que podem ser aplicadas à variável.

Você deve declarar todas as variáveis antes que elas possam ser usadas. A seguir, a forma básica de uma declaração variável -

```
data type variable [ = value][, variable [ = value] ...] ;
```

Aqui, o *tipo de dados* é um dos tipos de dados do Java e a *variável* é o nome da variável. Para declarar mais de uma variável do tipo especificado, você pode usar uma lista separada por vírgulas.

A seguir, exemplos válidos de declaração e inicialização de variáveis em Java -

Exemplo

```
int a, b, c;           // Declares three ints, a, b, and c.
int a = 10, b = 10;    // Example of initialization
byte B = 22;           // initializes a byte type variable B.
double pi = 3.14159;   // declares and assigns a value of PI.
char a = 'A';          // the char variable a is initialized with value 'a'
```

Este capítulo explicará vários tipos de variáveis disponíveis na linguagem Java. Existem três tipos de variáveis em Java -

Variáveis locais

Variáveis de instância

Variáveis de classe / estática

Variáveis locais são declaradas em métodos, construtores ou blocos.

As variáveis locais são criadas quando o método, construtor ou bloco é inserido e a variável será destruída quando sair do método, construtor ou bloco.

Modificadores de acesso não podem ser usados para variáveis locais.

As variáveis locais são visíveis apenas no método declarado, construtor ou bloco.

Variáveis locais são implementadas no nível da pilha internamente.

Não há valor padrão para variáveis locais, portanto variáveis locais devem ser declaradas e um valor inicial deve ser atribuído antes do primeiro uso.

## Exemplo

Aqui, a *idade* é uma variável local. Isso é definido dentro do método *pupAge ()* e seu escopo é limitado apenas a esse método.

```
public class Test {  
    public void pupAge() {  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Puppy age is: 7
```

## Exemplo

O exemplo a seguir usa a *idade* sem inicializá-lo, portanto, ocorreria um erro no momento da compilação.

```
public class Test {  
    public void pupAge() {  
        int age;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

```
}
```

Isso produzirá o seguinte erro ao compilá-lo -

## Saída

```
Test.java:4:variable number might not have been initialized
```

```
age = age + 7;
```

```
    ^
```

```
1 error
```

Variáveis de instância são declaradas em uma classe, masVariáveis de instância fora de um método, construtor ou qualquer bloco.

Quando um espaço é alocado para um objeto no heap, um slot para cada valor de variável de instância é criado.

Variáveis de instância são criadas quando um objeto é criado com o uso da palavra-chave 'new' e destruído quando o objeto é destruído.

As variáveis de instância armazenam valores que devem ser referenciados por mais de um método, construtor ou bloco, ou partes essenciais do estado de um objeto que devem estar presentes em toda a classe.

Variáveis de instância podem ser declaradas no nível de classe antes ou depois do uso.

Modificadores de acesso podem ser dados para variáveis de instância.

As variáveis de instância são visíveis para todos os métodos, construtores e blocos na classe. Normalmente, recomenda-se tornar essas variáveis privadas (nível de acesso). No entanto, a visibilidade de subclasses pode ser dada para essas variáveis com o uso de modificadores de acesso.

Variáveis de instância possuem valores padrão. Para números, o valor padrão é 0, para Booleans, é falso e, para referências de objetos, é nulo. Valores podem ser atribuídos durante a declaração ou dentro do construtor.

Variáveis de instância podem ser acessadas diretamente chamando o nome da variável dentro da classe. No entanto, dentro de métodos estáticos (quando variáveis de instância recebem acessibilidade), eles devem ser chamados usando o nome totalmente qualificado. *ObjectReference.VariableName* .

## Exemplo

```
import java.io.*;
public class Employee {

    // this instance variable is visible for any child class.
    public String name;

    // salary variable is visible in Employee class only.
    private double salary;

    // The name variable is assigned in the constructor.
```

```

public Employee (String empName) {
    name = empName;
}

// The salary variable is assigned a value.
public void setSalary(double empSal) {
    salary = empSal;
}

// This method prints the employee details.
public void printEmp() {
    System.out.println("name : " + name );
    System.out.println("salary :" + salary);
}

public static void main(String args[]) {
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
}
}

```

Isso produzirá o seguinte resultado -

## Saída

```

name : Ransika
salary :1000.0

```

Variáveis de classe também conhecidas como variáveis Variáveis de Classe / Estática estáticas são declaradas com a palavra-chave static em uma classe, mas fora de um método, construtor ou bloco.

Haveria apenas uma cópia de cada variável de classe por classe, independentemente de quantos objetos fossem criados a partir dela.

Variáveis estáticas raramente são usadas além de serem declaradas como constantes. Constantes são variáveis declaradas como públicas / privadas, finais e estáticas. Variáveis constantes nunca mudam de seu valor inicial.

Variáveis estáticas são armazenadas na memória estática. É raro usar variáveis estáticas que não sejam finais declaradas e usadas como constantes públicas ou privadas.

Variáveis estáticas são criadas quando o programa é iniciado e destruído quando o programa é interrompido.

A visibilidade é semelhante às variáveis de instância. No entanto, a maioria das variáveis estáticas é declarada pública, pois deve estar disponível para os usuários da classe.

Valores padrão são os mesmos que variáveis de instância. Para números, o valor padrão é 0; para os booleanos, é falso; e para referências de objetos, é nulo. Valores podem ser atribuídos durante a declaração ou dentro do construtor. Além disso, os valores podem ser atribuídos em blocos de inicialização estáticos especiais.

Variáveis estáticas podem ser acessadas chamando com o nome da classe *ClassName.VariableName* .

Ao declarar variáveis de classe como `public static final`, os nomes das variáveis (constantes) são todos em maiúsculas. Se as variáveis estáticas não forem públicas e finais, a sintaxe de nomenclatura é a mesma das variáveis locais e de instância.

## Exemplo

```
import java.io.*;
public class Employee {

    // salary variable is a private static variable
    private static double salary;

    // DEPARTMENT is a constant
    public static final String DEPARTMENT = "Development ";

    public static void main(String args[]) {
        salary = 1000;
        System.out.println(DEPARTMENT + "average salary:" + salary);
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Development average salary:1000
```

**Nota** - Se as variáveis são acessadas de uma classe externa, a constante deve ser acessada como `Employee.DEPARTMENT`

Você já usou modificadores de acesso (público e privado) nesteO que é o próximo? capítulo. O próximo capítulo explicará os modificadores de acesso e os modificadores de não acesso em detalhes.

# Java - Modifier Modifiers

Modificadores são palavras-chave que você adiciona a essas definições para alterar seus significados. A linguagem Java possui uma ampla variedade de modificadores, incluindo os seguintes -

Modificadores de Acesso Java

Modificadores sem acesso

Para usar um modificador, você inclui sua palavra-chave na definição de uma classe, método ou variável. O modificador precede o restante da instrução, como no exemplo a seguir.

## Exemplo

```
public class className {
    // ...
}
```

```

}

private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {
    // body of method
}

```

Java fornece vários modificadores de acesso para definir níveis de acesso para classes, variáveis, métodos e construtores. Os quatro níveis de acesso são -

Visível para o pacote, o padrão. Nenhum modificador é necessário.

Visível apenas para a turma (privado).

Visível para o mundo (público).

Visível para o pacote e todas as subclasses (protegidas).

Java fornece vários modificadores sem acesso para obter muitas outras funcionalidades.

O modificador *estático* para criar métodos e variáveis de classe.

O modificador *final* para finalizar as implementações de classes, métodos e variáveis.

O modificador *abstrato* para criar classes e métodos abstratos.

Os modificadores *sincronizados* e *voláteis* , que são usados para encadeamentos.

Na próxima seção, discutiremos sobre os operadores básicos usados na linguagem Java. O capítulo dará uma visão geral de como esses operadores podem ser usados durante o desenvolvimento de aplicativos.

## Java - Operadores Básicos

Java fornece um rico conjunto de operadores para manipular variáveis. Podemos dividir todos os operadores Java nos seguintes grupos -

Operadores aritméticos

Operadores Relacionais

Operadores bit a bit

Operadores lógicos

Operadores de atribuição

Operadores Diversos

Operadores aritméticos são usados em expressões matemáticas da mesma maneira que são usados em álgebra. A tabela a seguir lista os operadores aritméticos -

Suponha que a variável inteira A detenha 10 e a variável B detenha 20, então -

Mostrar exemplos

Operador	Descrição	Exemplo
+ (Adição)	Adiciona valores em ambos os lados do operador.	A + B dará 30
- (subtração)	Subtrai o operando direito do operando esquerdo.	A - B vai dar -10
* (Multiplicação)	Multiplica valores em ambos os lados do operador.	A * B dará 200
/ (Divisão)	Divide o operando esquerdo pelo operando direito.	B / A vai dar 2
% (Módulo)	Divide o operando esquerdo pelo operando direito e retorna o restante.	B% A dará 0
++ (incremento)	Aumenta o valor do operando em 1.	B ++ dá 21
-- (Decremento)	Diminui o valor do operando em 1.	B-- dá 19

Existem os seguintes operadores relacionais suportados pela linguagem Java. Os Operadores Relacionais

Suponha que a variável A detenha 10 e a variável B detenha 20, então -

Mostrar exemplos

Operador	Descrição	Exemplo
== (igual a)	Verifica se os valores de dois operandos são iguais ou não, se sim, a condição se torna verdadeira.	(A == B) não é verdade.
!= (não igual a)	Verifica se os valores de dois operandos são iguais ou não, se os valores não forem iguais, a condição se tornará verdadeira.	(A != B) é verdade.
> (maior que)	Verifica se o valor do operando esquerdo é maior que o valor do operando direito, se sim, a condição se torna verdadeira.	(A > B) não é verdade.
< (menor que)	Verifica se o valor do operando esquerdo é menor que o valor do operando direito, se sim, a condição se torna verdadeira.	(A < B) é verdade.
>= (maior que ou igual a)	Verifica se o valor do operando esquerdo é maior ou igual ao valor do operando	(A >= B) não é verdade.



	direito, se sim, a condição se torna verdadeira.	
$\leq$ (menor ou igual a)	Verifica se o valor do operando esquerdo é menor ou igual ao valor do operando direito, se sim, a condição se torna verdadeira.	$(A \leq B)$ é verdade.

Java define vários operadores bit a bit, que podem ser aplicados aos tipos inteiros long, int, short, char e byte.

O operador bit a bit trabalha em bits e executa a operação bit a bit. Assuma se  $a = 60$  e  $b = 13$ ; agora em formato binário eles serão os seguintes -

$a = 0011\ 1100$

$b = 0000\ 1101$

-----

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a \wedge b = 0011\ 0001$

$\sim a = 1100\ 0011$

A tabela a seguir lista os operadores bit a bit -

Assuma que a variável inteira A contém 60 e a variável B contém 13 então -

Mostrar exemplos

Operador	Descrição	Exemplo
$\&$ (bit a bit e)	Binary AND Operator copia um pouco para o resultado, se existir em ambos os operandos.	$(A \& B)$ vai dar 12, que é 0000 1100
$ $ (bit a bit ou)	Operador binário OR copia um bit se existir em um dos operandos.	$(A   B)$ vai dar 61, que é 0011 1101
$\wedge$ (XOR bit a bit)	Operador XOR binário copia o bit se estiver definido em um operando, mas não em ambos.	$(A \wedge B)$ vai dar 49, que é 0011 0001
$\sim$ (negação bit a bit)	Binary Ones Complement Operator é unário e tem o efeito de 'flipping' bits.	$(\sim A)$ dará -61, que é 1100 0011 no formulário de complemento de 2, devido a um número binário assinado.
$\ll$ (turno da esquerda)	Operador Shift Esquerdo. O valor dos operandos esquerdos é movido para a esquerda pelo número de bits	Um $\ll 2$ vai dar 240, que é 1111 0000

	especificado pelo operando direito.	
>> (deslocamento para a direita)	Operador de deslocamento à direita binário. O valor dos operandos esquerdos é movido para a direita pelo número de bits especificado pelo operando direito.	A >> 2 dará 15, que é 1111
>>> (preenchimento zero de deslocamento à direita)	Desloque o operador de preenchimento com zero à direita. O valor dos operandos à esquerda é movido para a direita pelo número de bits especificado pelo operando à direita e os valores deslocados são preenchidos com zeros.	Um >>> 2 dará 15, que é 0000 1111

A tabela a seguir lista os operadores lógicos -

Os operadores lógicos

Suponha que as variáveis booleanas A sejam verdadeiras e as variáveis B sejam falsas, então -

Mostrar exemplos

Operador	Descrição	Exemplo
&& (lógico e)	Chamado Lógico E operador. Se ambos os operandos forem diferentes de zero, a condição se tornará verdadeira.	(A && B) é falso
(lógico ou)	Chamado Lógico OU Operador. Se algum dos dois operandos for diferente de zero, a condição se tornará verdadeira.	(A    B) é verdade
! (lógico não)	Operador NÃO Lógico Chamado. Use para reverter o estado lógico de seu operando. Se uma condição for verdadeira, o operador Lógico NOT tornará falso.	! (A && B) é verdade

A seguir estão os operadores de atribuição suportados pela linguagem Java - Os operadores de atribuição

Mostrar exemplos

Operador	Descrição	Exemplo
=	Operador de atribuição simples. Atribui valores dos operandos do lado direito ao operando do lado esquerdo.	C = A + B atribuirá valor de A + B a C
+ =	Adicionar operador de atribuição E. Adiciona o operando direito ao operando esquerdo e atribui o resultado ao operando esquerdo.	C + = A é equivalente a C = C + A

- =	Subtrair e operador de atribuição. Subtrai o operando direito do operando esquerdo e atribui o resultado ao operando esquerdo.	C - = A é equivalente a C = C - A
* =	Operador Multiply AND assignment. Multiplica o operando direito com o operando esquerdo e atribui o resultado ao operando esquerdo.	C * = A é equivalente a C = C * A
/ =	Divide AND operador de atribuição. Ele divide o operando esquerdo pelo operando direito e atribui o resultado ao operando esquerdo.	C / = A é equivalente a C = C / A
% =	Módulo E operador de atribuição. Ele recebe módulo usando dois operandos e atribui o resultado ao operando esquerdo.	C % = A é equivalente a C = C % A
<< =	Esquerda e operador de atribuição.	C << = 2 é o mesmo que C = C << 2
>> =	Operador de deslocamento e atribuição à direita.	C >> = 2 é o mesmo que C = C >> 2
& =	Operador Bitwise AND assignment.	C & = 2 é o mesmo que C = C e 2
^ =	operador de atribuição OU OR exclusivo bit a bit.	C ^ = 2 é o mesmo que C = C ^ 2
=	Operador bit a bit OR e de atribuição.	C   = 2 é o mesmo que C = C   2

Existem poucos outros operadores suportados pela linguagem Java. Operadores Diversos

## Operador Condicional (?:)

O operador condicional também é conhecido como o **operador ternário**. Esse operador consiste em três operandos e é usado para avaliar expressões booleanas. O objetivo do operador é decidir qual valor deve ser atribuído à variável. O operador está escrito como -

```
variable x = (expression) ? value if true : value if false
```

A seguir, um exemplo -

### Exemplo

```
public class Test {

    public static void main(String args[]) {
        int a, b;
        a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b );

        b = (a == 10) ? 20: 30;
```

```
        System.out.println( "Value of b is : " + b );
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Value of b is : 30
Value of b is : 20
```

## instanceof Operator

Este operador é usado apenas para variáveis de referência de objetos. O operador verifica se o objeto é de um tipo específico (tipo de classe ou tipo de interface). instanceof operator está escrito como -

```
( Object reference variable ) instanceof (class/interface type)
```

Se o objeto referenciado pela variável no lado esquerdo do operador passar a verificação IS-A para o tipo de classe / interface no lado direito, o resultado será verdadeiro. A seguir, um exemplo -

### Exemplo

```
public class Test {

    public static void main(String args[]) {

        String name = "James";

        // following will return true since name is type of String
        boolean result = name instanceof String;
        System.out.println( result );
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
true
```

Esse operador ainda retornará true, se o objeto que está sendo comparado for a atribuição compatível com o tipo à direita. A seguir, mais um exemplo -

### Exemplo

```
class Vehicle {}

public class Car extends Vehicle {

    public static void main(String args[]) {

        Vehicle a = new Car();
        boolean result = a instanceof Car;
        System.out.println( result );
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
true
```

A precedência do operador determina o agrupamento de termos em uma expressão. Isso afeta como uma expressão é avaliada. Certos operadores têm precedência mais alta que outros; por exemplo, o operador de multiplicação tem precedência mais alta que o operador de adição -

Por exemplo,  $x = 7 + 3 * 2$ ; aqui  $x$  é atribuído 13, não 20 porque o operador  $*$  tem precedência mais alta que  $+$ , então ele primeiro é multiplicado com  $3 * 2$  e, em seguida, adiciona 7.

Aqui, os operadores com a maior precedência aparecem na parte superior da tabela, e aqueles com os mais baixos aparecem na parte inferior. Dentro de uma expressão, os operadores de precedência mais alta serão avaliados primeiro.

Categoria	Operador	Associatividade
Postfix	> () []. (operador de ponto)	Da esquerda para direita
Unário	> ++ - ! ~	Direita para esquerda
Multiplicativo	> * /	Da esquerda para direita
Aditivo	> + -	Da esquerda para direita
Mudança	>>> >>> <<	Da esquerda para direita
Relacional	>>> = <<=	Da esquerda para direita
Igualdade	> = !=	Da esquerda para direita
Bit a bit E	> &	Da esquerda para direita
Bit a bit XOR	> ^	Da esquerda para direita
Bit a bit OU	>	Da esquerda para direita
E lógico	> &&	Da esquerda para direita

OR lógico	>	Da esquerda para direita
Condicional	?:	Direita para esquerda
Tarefa	> = + = - = * = / = % = >> = << = & = ^ =   =	Direita para esquerda

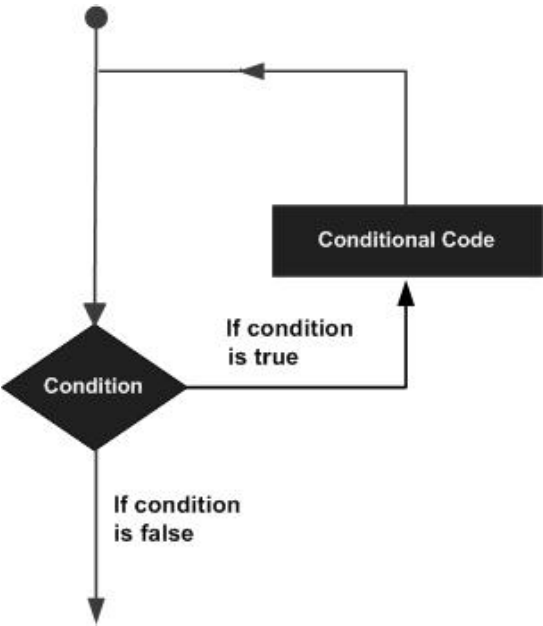
O próximo capítulo explicará sobre o controle de loop na programaçãoO que é o próximo? Java. O capítulo descreverá vários tipos de loops e como esses loops podem ser usados no desenvolvimento de programas Java e para quais fins eles estão sendo usados.

## Java - controle de loop

Pode haver uma situação em que você precise executar um bloco de código várias vezes. Em geral, as instruções são executadas seqüencialmente: a primeira instrução em uma função é executada primeiro, seguida pela segunda e assim por diante.

As linguagens de programação fornecem várias estruturas de controle que permitem caminhos de execução mais complicados.

Uma instrução de **loop** nos permite executar uma instrução ou grupo de instruções várias vezes e seguir é a forma geral de uma instrução de loop na maioria das linguagens de programação -



A linguagem de programação Java fornece os seguintes tipos de loop para lidar com os requisitos de loop. Clique nos links a seguir para verificar seus detalhes.

Sr. Não.	Loop e Descrição
-------------	------------------

1	<b>enquanto loop</b> Repete uma instrução ou grupo de instruções enquanto uma determinada condição é verdadeira. Ele testa a condição antes de executar o corpo do loop.
2	<b>para loop</b> Execute uma sequência de instruções várias vezes e abrevie o código que gerencia a variável de loop.
3	<b>fazer ... while loop</b> Como uma declaração while, exceto que testa a condição no final do corpo do loop.

As instruções de controle de loop alteram a execução de sua sequência normal. Quando a execução deixa um escopo, todos os objetos automáticos que foram criados nesse escopo são destruídos.

Java suporta as seguintes instruções de controle. Clique nos links a seguir para verificar seus detalhes.

Sr. Não.	Declaração de controle e descrição
1	<b>declaração de quebra</b> Encerra a instrução <b>loop</b> ou <b>switch</b> e transfere a execução para a instrução imediatamente após o loop ou comutador.
2	<b>continuar instrução</b> Faz com que o loop pule o restante de seu corpo e repita imediatamente sua condição antes de reiterar.

A partir do Java 5, o loop for aprimorado foi introduzido. Isso é usado principalmente para percorrer a coleção de elementos, incluindo matrizes.

## Sintaxe

A seguir está a sintaxe do loop aprimorado -

```
for(declaration : expression) {
    // Statements
}
```

**Declaração** - A variável de bloco recém-declarada é de um tipo compatível com os elementos da matriz que você está acessando. A variável estará disponível dentro do bloco for e seu valor será o mesmo que o elemento da matriz atual.

**Expressão** - Isso avalia o array que você precisa percorrer. A expressão pode ser uma variável de matriz ou chamada de método que retorna uma matriz.

## Exemplo

```
public class Test {  
  
    public static void main(String args[]) {  
        int [] numbers = {10, 20, 30, 40, 50};  
  
        for(int x : numbers ) {  
            System.out.print( x );  
            System.out.print(",");  
        }  
        System.out.print("\n");  
        String [] names = {"James", "Larry", "Tom", "Lacy"};  
  
        for( String name : names ) {  
            System.out.print( name );  
            System.out.print(",");  
        }  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

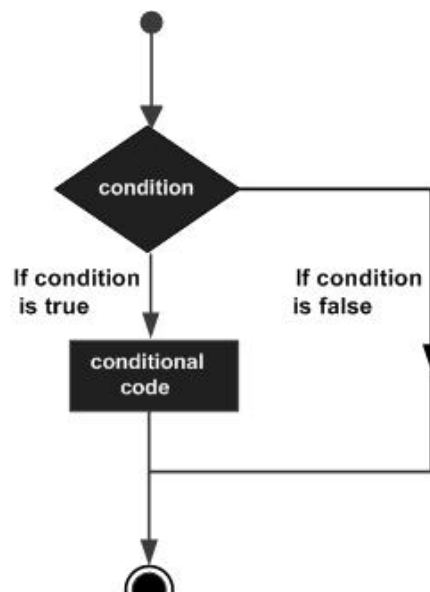
```
10, 20, 30, 40, 50,  
James, Larry, Tom, Lacy,
```

No próximo capítulo, estaremos aprendendo sobre as declarações de `if` que é o próximo? tomada de decisão na programação Java.

## Java - Tomada de Decisão

Estruturas de tomada de decisão têm uma ou mais condições para serem avaliadas ou testadas pelo programa, juntamente com uma declaração ou instruções que devem ser executadas se a condição for determinada como verdadeira e, opcionalmente, outras instruções a serem executadas se a condição for determinada to be false.

A seguir está a forma geral de uma estrutura de tomada de decisão típica encontrada na maioria das linguagens de programação -







A linguagem de programação Java fornece os seguintes tipos de instruções de tomada de decisão. Clique nos links a seguir para verificar seus detalhes.

Sr. Não.	Declaração & Descrição
1	<b>se declaração</b> Uma <b>instrução if</b> consiste em uma expressão booleana seguida de uma ou mais instruções.
2	<b>if ... else statement</b> Uma <b>instrução if</b> pode ser seguida por uma <b>instrução else</b> opcional , que é executada quando a expressão booleana é falsa.
3	<b>aninhado se declaração</b> Você pode usar um <b>if if else se</b> instrução dentro de outro <b>if</b> ou <b>else if</b> statement (s).
4	<b>mudar a indicação</b> Uma instrução <b>switch</b> permite que uma variável seja testada quanto à igualdade em relação a uma lista de valores.

Nós cobrimos **operador condicional?** : no capítulo anterior, que pode: Operador O ? ser usado para substituir **if ... else** statements. Tem a seguinte forma geral -

```
Exp1 ? Exp2 : Exp3;
```

Onde Exp1, Exp2 e Exp3 são expressões. Observe o uso e a colocação dos dois pontos.

Para determinar o valor da expressão inteira, inicialmente exp1 é avaliado.

Se o valor de exp1 for true, o valor de Exp2 será o valor da expressão inteira.

Se o valor de exp1 for falso, Exp3 será avaliado e seu valor se tornará o valor da expressão inteira.

## O que é o próximo?

No próximo capítulo, discutiremos sobre a classe Number (no pacote java.lang) e suas subclasses na linguagem Java.

Examinaremos algumas das situações em que você usará instâncias dessas classes em vez dos tipos de dados primitivos, além de classes, como formatação, funções matemáticas que você precisa conhecer ao trabalhar com o Numbers.

## Classe Java - Números

Normalmente, quando trabalhamos com o Numbers, usamos tipos de dados primitivos,

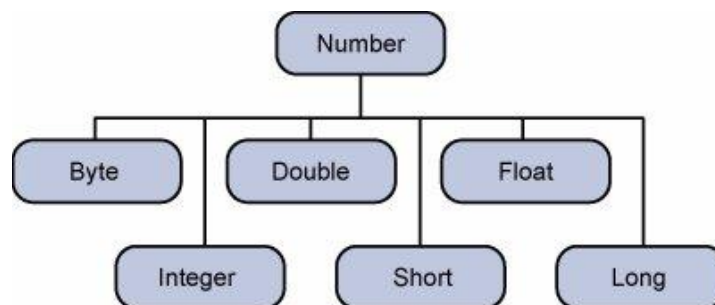
como byte, int, long, double, etc.

## Exemplo

```
int i = 5000;  
float gpa = 13.65;  
double mask = 0xaf;
```

No entanto, em desenvolvimento, nos deparamos com situações em que precisamos usar objetos em vez de tipos de dados primitivos. Para conseguir isso, o Java fornece **classes de wrapper**.

Todas as classes do wrapper (Integer, Long, Byte, Double, Float, Short) são subclasses da classe abstrata Number.



O objeto da classe wrapper contém ou envolve seu respectivo tipo de dados primitivo. A conversão de tipos de dados primitivos em objetos é chamada de **boxe** e isso é feito pelo compilador. Portanto, ao usar uma classe wrapper, você só precisa passar o valor do tipo de dados primitivo para o construtor da classe Wrapper.

E o objeto Wrapper será convertido de volta para um tipo de dado primitivo, e esse processo é chamado de unboxing. A classe **Number** faz parte do pacote java.lang.

A seguir, um exemplo de boxe e unboxing -

## Exemplo

```
public class Test {  
  
    public static void main(String args[]) {  
        Integer x = 5; // boxes int to an Integer object  
        x = x + 10; // unboxes the Integer to a int  
        System.out.println(x);  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

15

Quando x é atribuído a um valor inteiro, o compilador empacota o inteiro porque x é um objeto inteiro. Mais tarde, x é descompactado para que eles possam ser adicionados como um inteiro.

## Métodos numéricos

A seguir, a lista dos métodos de instância que todas as subclasses da classe Number implementam -

Sr. Não.	Método e Descrição
1	<b>xxxValue ()</b> Converte o valor <i>deste</i> objeto Number no tipo de dado xxx e o retorna.
2	<b>comparado a()</b> Compara <i>este</i> objeto Number ao argumento.
3	<b>é igual a()</b> Determina se esse objeto numérico é igual ao argumento.
4	<b>valor de()</b> Retorna um objeto Integer contendo o valor da primitiva especificada.
5	<b>para sequenciar()</b> Retorna um objeto String representando o valor de um int ou Integer especificado.
6	<b>parseInt ()</b> Este método é usado para obter o tipo de dados primitivo de uma determinada String.
7	<b>abs ()</b> Retorna o valor absoluto do argumento.
8	<b>ceil ()</b> Retorna o menor inteiro que é maior que ou igual ao argumento. Retornado como um duplo.
9	<b>chão()</b> Retorna o maior número inteiro menor ou igual ao argumento. Retornado como um duplo.
10	<b>rint ()</b> Retorna o inteiro que está mais próximo em valor ao argumento. Retornado como um duplo.
11	<b>volta()</b> Retorna o long ou int mais próximo, conforme indicado pelo tipo de retorno do método para o argumento.

12	<b>min ()</b> Retorna o menor dos dois argumentos.
13	<b>max ()</b> Retorna o maior dos dois argumentos.
14	<b>exp ()</b> Retorna a base dos logaritmos naturais, e, ao poder do argumento.
15	<b>registro()</b> Retorna o logaritmo natural do argumento.
16	<b>prisioneiro de guerra()</b> Retorna o valor do primeiro argumento elevado à potência do segundo argumento.
17	<b>sqrt ()</b> Retorna a raiz quadrada do argumento.
18	<b>pecado()</b> Retorna o seno do valor duplo especificado.
19	<b>cos ()</b> Retorna o cosseno do valor duplo especificado.
20	<b>bronzeado()</b> Retorna a tangente do valor duplo especificado.
21	<b>como em()</b> Retorna o arco seno do valor duplo especificado.
22	<b>acos ()</b> Retorna o arccosine do valor duplo especificado.
23	<b>um bronzeado()</b> Retorna o arco tangente do valor duplo especificado.
24	<b>atan2 ()</b> Converte coordenadas retangulares (x, y) em coordenadas polares (r, teta) e retorna theta.
25	<b>toDegrees ()</b> Converte o argumento em graus.

26	<b>para os rádios ()</b> Converte o argumento para radianos.
27	<b>aleatória()</b> Retorna um número aleatório.

## O que é o próximo?

Na próxima seção, estaremos passando pela classe `Character` em Java. Você estará aprendendo como usar os Caracteres do objeto e o tipo de dados primitivos `char` em Java.

## Java - Classe de Personagem

Normalmente, quando trabalhamos com caracteres, usamos tipos de dados primitivos `char`.

### Exemplo

```
char ch = 'a';

// Unicode for uppercase Greek omega character
char uniChar = '\u039A';

// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

No entanto, em desenvolvimento, nos deparamos com situações em que precisamos usar objetos em vez de tipos de dados primitivos. Para conseguir isso, Java fornece classe de invólucro **Character** para tipo de dados primitivos `char`.

A classe `Character` oferece vários métodos úteis de classe (isto é, estáticos) para manipular caracteres. Você pode criar um objeto `Character` com o construtor `Character` -

```
Character ch = new Character('a');
```

O compilador Java também criará um objeto `Character` para você em algumas circunstâncias. Por exemplo, se você passar um caractere primitivo para um método que espera um objeto, o compilador converterá automaticamente o caractere em caractere para você. Esse recurso é chamado de autoboxing ou unboxing, se a conversão for para o outro lado.

### Exemplo

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch = 'a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

## Sequências de Escape

Um caractere precedido por uma barra invertida (`\`) é uma sequência de escape e tem um

significado especial para o compilador.

O caractere de nova linha (\n) tem sido usado com frequência neste tutorial nas instruções `System.out.println ()` para avançar para a próxima linha depois que a sequência é impressa.

A tabela a seguir mostra as seqüências de escape do Java -

Sequência de fuga	Descrição
\ t	Insere uma guia no texto neste momento.
\ b	Insere um backspace no texto neste momento.
\ n	Insere uma nova linha no texto neste momento.
\ r	Insere um retorno de carro no texto neste momento.
\ f	Insere um feed de formulário no texto neste momento.
\ '	Insere um caractere de aspas simples no texto neste momento.
\ "	Insere um caractere aspas duplas no texto neste momento.
\\	Insere um caractere de barra invertida no texto neste momento.

Quando uma seqüência de escape é encontrada em uma instrução `print`, o compilador a interpreta de acordo.

## Exemplo

Se você quiser colocar aspas entre aspas, você deve usar a seqüência de escape, \", nas citações interiores -

```
public class Test {  
  
    public static void main(String args[]) {  
        System.out.println("She said \"Hello!\" to me.");  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
She said "Hello!" to me.
```

## Métodos de Personagem

A seguir, a lista dos métodos de instância importantes que todas as subclasses da classe `Character` implementam -

Sr. Não.	Método e Descrição
-------------	--------------------

1	<b>isLetter ()</b> Determina se o valor do caractere especificado é uma letra.
2	<b>isDigit ()</b> Determina se o valor de char especificado é um dígito.
3	<b>isWhitespace ()</b> Determina se o valor de char especificado é espaço em branco.
4	<b>isUpperCase ()</b> Determina se o valor do caractere especificado é maiúsculo.
5	<b>isLowerCase ()</b> Determina se o valor do caractere especificado é minúsculo.
6	<b>toUpperCase ()</b> Retorna a forma maiúscula do valor char especificado.
7	<b>toLowerCase ()</b> Retorna a forma minúscula do valor de char especificado.
8	<b>para sequenciar()</b> Retorna um objeto String representando o valor do caractere especificado, ou seja, uma string de um caractere.

Para obter uma lista completa de métodos, consulte a especificação da API `java.lang.Character`.

## O que é o próximo?

Na próxima seção, vamos percorrer a classe `String` em Java. Você aprenderá como declarar e usar `Strings` de forma eficiente, bem como alguns dos métodos importantes da classe `String`.

## Java - Strings Class

`Strings`, que são amplamente utilizados na programação Java, são uma sequência de caracteres. Na linguagem de programação Java, as cadeias são tratadas como objetos.

A plataforma Java fornece a classe `String` para criar e manipular strings.

## Criando Strings

A maneira mais direta de criar uma string é escrever -

```
String greeting = "Hello world!";
```

Sempre que encontrar uma string literal em seu código, o compilador cria um objeto `String`

com seu valor nesse caso, "Hello world!".

Como com qualquer outro objeto, você pode criar objetos String usando a nova palavra-chave e um construtor. A classe String possui 11 construtores que permitem fornecer o valor inicial da string usando fontes diferentes, como uma matriz de caracteres.

## Exemplo

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };  
        String helloString = new String(helloArray);  
        System.out.println( helloString );  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
hello.
```

**Nota** - A classe String é imutável, portanto, uma vez criada, o objeto String não pode ser alterado. Se houver a necessidade de fazer muitas modificações em Strings de caracteres, você deve usar Classes de Buffer de Buffer e Construtor de String .

Os métodos usados para obter informações sobre um objeto são Comprimento da corda conhecidos como **métodos de acesso** . Um método de assessor que você pode usar com strings é o método length (), que retorna o número de caracteres contidos no objeto string.

O programa a seguir é um exemplo de **length ()** , método String class.

## Exemplo

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        System.out.println( "String Length is : " + len );  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
String Length is : 17
```

A classe String inclui um método para concatenar duas strings - Concatenando Cordas

```
string1.concat(string2);
```

Isso retorna uma nova string que é string1 com string2 adicionada a ela no final. Você também pode usar o método concat () com literais de string, como em -



```
"My name is ".concat("Zara");
```

Strings são mais comumente concatenadas com o operador +, como em -

```
"Hello," + " world" + "!"
```

que resulta em -

```
"Hello, world!"
```

Vamos ver o exemplo a seguir -

## Exemplo

```
public class StringDemo {  
  
    public static void main(String args[]) {  
        String string1 = "saw I was ";  
        System.out.println("Dot " + string1 + "Tod");  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Dot saw I was Tod
```

## Criando strings de formato

Você tem os métodos printf () e format () para imprimir a saída com números formatados. A classe String possui um método de classe equivalente, format (), que retorna um objeto String em vez de um objeto PrintStream.

Usar o método static format () de String permite criar uma string formatada que você pode reutilizar, ao contrário de uma instrução de impressão única. Por exemplo, em vez de -

## Exemplo

```
System.out.printf("The value of the float variable is " +  
    "%f, while the value of the integer " +  
    "variable is %d, and the string " +  
    "is %s", floatVar, intVar, stringVar);
```

Você pode escrever -

```
String fs;  
fs = String.format("The value of the float variable is " +  
    "%f, while the value of the integer " +  
    "variable is %d, and the string " +  
    "is %s", floatVar, intVar, stringVar);  
System.out.println(fs);
```

Aqui está a lista de métodos suportados pela classe String -

Métodos de String

Sr. Não.	Método e Descrição
-------------	--------------------

1	<b>char charAt (int index)</b> Retorna o caractere no índice especificado.
2	<b>int compareTo (objeto o)</b> Compara essa string para outro objeto.
3	<b>int compareTo (String outroString)</b> Compara duas cadeias lexicograficamente.
4	<b>int compareToIgnoreCase (String str)</b> Compara duas cadeias lexicograficamente, ignorando as diferenças entre maiúsculas e minúsculas.
5	<b>Concat de String (String str)</b> Concatena a string especificada ao final dessa string.
6	<b>boolean contentEquals (StringBuffer sb)</b> Retorna true se e somente se essa String representar a mesma sequência de caracteres que o StringBuffer especificado.
7	<b>static String copyValueOf (dados do char [])</b> Retorna uma String que representa a sequência de caracteres na matriz especificada.
8	<b>Cadeia estática copyValueOf (dados char [], int offset, int count)</b> Retorna uma String que representa a sequência de caracteres na matriz especificada.
9	<b>boolean endsWith (sufixo de sequência)</b> Testa se esta cadeia termina com o sufixo especificado.
10	<b>equals booleanos (objeto anObject)</b> Compara essa string ao objeto especificado.
11	<b>boolean equalsIgnoreCase (String outroString)</b> Compara essa String a outra String, ignorando as considerações de caso.
12	<b>byte getBytes ()</b> Codifica essa String em uma sequência de bytes usando o conjunto de caracteres padrão da plataforma, armazenando o resultado em uma nova matriz de bytes.
13	<b>byte [] getBytes (string nome_da_fila)</b> Codifica essa String em uma sequência de bytes usando o charset nomeado, armazenando o resultado em uma nova matriz de bytes.

# Java - Arrays

Java fornece uma estrutura de dados, a **matriz**, que armazena uma coleção sequencial de tamanho fixo de elementos do mesmo tipo. Uma matriz é usada para armazenar uma coleção de dados, mas geralmente é mais útil pensar em uma matriz como uma coleção de variáveis do mesmo tipo.

Em vez de declarar variáveis individuais, como `number0`, `number1`, ... e `number99`, você declara uma variável de matriz, como `números`, e usa `números [0]`, `números [1]` e ..., `números [99]` para representar variáveis individuais.

Este tutorial apresenta como declarar variáveis de matriz, criar matrizes e matrizes de processo usando variáveis indexadas.

## Declarando Variáveis de Matriz

Para usar uma matriz em um programa, você deve declarar uma variável para fazer referência à matriz e especificar o tipo de matriz que a variável pode referenciar. Aqui está a sintaxe para declarar uma variável de array -

### Sintaxe

```
dataType[] arrayRefVar;    // preferred way.  
or  
dataType arrayRefVar[];    // works but not preferred way.
```

**Nota** - O estilo **dataType [] arrayRefVar** é o preferido. O estilo **dataType arrayRefVar []** vem da linguagem C / C ++ e foi adotado em Java para acomodar os programadores C / C ++.

### Exemplo

Os trechos de código a seguir são exemplos dessa sintaxe -

```
double[] myList;    // preferred way.  
or  
double myList[];    // works but not preferred way.
```

## Criando Matrizes

Você pode criar uma matriz usando o novo operador com a seguinte sintaxe -

### Sintaxe

```
arrayRefVar = new dataType[arraySize];
```

A declaração acima faz duas coisas -

Cria um array usando `new dataType [arraySize]`.

Atribui a referência da matriz recém-criada à variável `arrayRefVar`.

Declarar uma variável de matriz, criar uma matriz e atribuir a referência da matriz à variável pode ser combinada em uma instrução, como mostrado abaixo -

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternativamente, você pode criar matrizes da seguinte forma -

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

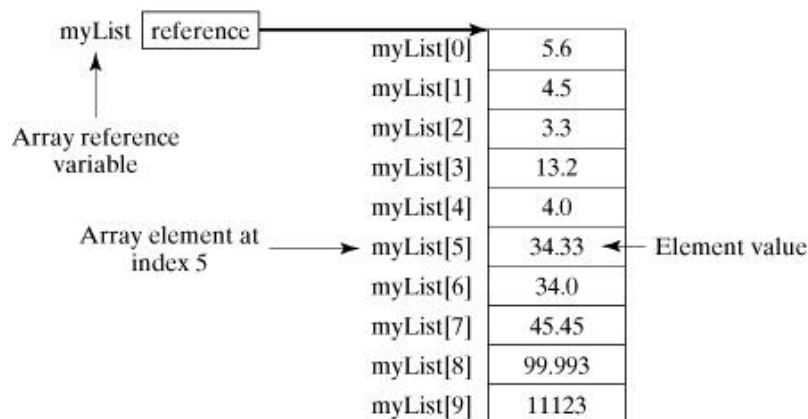
Os elementos da matriz são acessados pelo **índice** . Os índices de matriz são baseados em 0; isto é, eles começam de 0 a **arrayRefVar.length-1** .

## Exemplo

A seguinte declaração declara que uma variável de matriz, myList, cria uma matriz de 10 elementos do tipo double e atribui sua referência a myList -

```
double[] myList = new double[10];
```

A figura a seguir representa o array myList. Aqui, myList contém dez valores duplos e os índices são de 0 a 9.



## Processando Matrizes

Ao processar elementos de matriz, geralmente usamos loop **de** loop ou **foreach** porque todos os elementos em uma matriz são do mesmo tipo e o tamanho da matriz é conhecido.

## Exemplo

Aqui está um exemplo completo mostrando como criar, inicializar e processar matrizes -

```
public class TestArray {  
  
    public static void main(String[] args) {  
        double[] myList = {1.9, 2.9, 3.4, 3.5};  
  
        // Print all the array elements  
        for (int i = 0; i < myList.length; i++) {  
            System.out.println(myList[i] + " ");  
        }  
  
        // Summing all elements  
        double total = 0;  
        for (int i = 0; i < myList.length; i++) {  
            total += myList[i];  
        }  
        System.out.println("Total is " + total);  
  
        // Finding the Largest element
```

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
    if (myList[i] > max) max = myList[i];
}
System.out.println("Max is " + max);
}
```

Isso produzirá o seguinte resultado -

## Saída

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

## Os loops foreach

O JDK 1.5 introduziu um novo loop for conhecido como foreach loop ou enhanced for loop, que permite percorrer o array completo sequencialmente sem usar uma variável de índice.

## Exemplo

O código a seguir exibe todos os elementos na matriz myList -

```
public class TestArray {

    public static void main(String[] args) {
        double[] myList = {1.9, 2.9, 3.4, 3.5};

        // Print all the array elements
        for (double element: myList) {
            System.out.println(element);
        }
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
1.9
2.9
3.4
3.5
```

## Passando Arrays para Métodos

Assim como você pode passar valores de tipos primitivos para métodos, você também pode passar matrizes para métodos. Por exemplo, o método a seguir exibe os elementos em uma matriz **int** -

## Exemplo

```
public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}
```

Você pode invocá-lo passando uma matriz. Por exemplo, a instrução a seguir chama o método printArray para exibir 3, 1, 2, 6, 4 e 2 -

### Exemplo

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

## Retornando uma matriz de um método

Um método também pode retornar uma matriz. Por exemplo, o método a seguir retorna uma matriz que é a reversão de outra matriz -

### Exemplo

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];

    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
        result[j] = list[i];
    }
    return result;
}
```

## A classe Arrays

A classe java.util.Arrays contém vários métodos estáticos para classificação e pesquisa de arrays, comparação de arrays e preenchimento de elementos de matriz. Esses métodos são sobrecarregados para todos os tipos primitivos.

Sr. Não.	Método e Descrição
1	<p><b>public static int binarySearch (Object [] a, Chave do objeto)</b></p> <p>Pesquisa a matriz especificada de Object (Byte, Int, double, etc.) para o valor especificado usando o algoritmo de pesquisa binária. A matriz deve ser classificada antes de fazer esta chamada. Isso retorna o índice da chave de pesquisa, se ela estiver contida na lista; caso contrário, retorna (- (ponto de inserção + 1)).</p>
2	<p><b>equals estatais booleanos públicos (long [] a, long [] a2)</b></p> <p>Retorna true se as duas matrizes especificadas de longs forem iguais entre si. Dois arrays são considerados iguais se ambos os arrays contiverem o mesmo número de elementos, e todos os pares correspondentes de elementos nos dois arrays forem iguais. Isso retorna true se os dois arrays forem iguais. O mesmo método pode ser usado por todos os outros tipos de dados primitivos (Byte,</p>

	short, Int, etc.)
3	<b>preenchimento void static público (int [] a, int val)</b> Atribui o valor int especificado a cada elemento da matriz especificada de ints. O mesmo método poderia ser usado por todos os outros tipos de dados primitivos (Byte, short, Int, etc.)
4	<b>public static void sort (Object [] a)</b> Classifica a matriz especificada de objetos em uma ordem crescente, de acordo com a ordem natural de seus elementos. O mesmo método poderia ser usado por todos os outros tipos de dados primitivos (Byte, short, Int, etc.)

## Java - data e hora

Java fornece a classe **Date** disponível no pacote **java.util** , essa classe contém a data e a hora atuais.

A classe Date suporta dois construtores, conforme mostrado na tabela a seguir.

Sr. Não.	Construtor e Descrição
1	<b>Encontro( )</b> Esse construtor inicializa o objeto com a data e hora atuais.
2	<b>Data (milissegundos longos)</b> Esse construtor aceita um argumento que é igual ao número de milissegundos decorridos desde a meia-noite de 1º de janeiro de 1970.

A seguir estão os métodos da classe de data.

Sr. Não.	Método e Descrição
1	<b>boolean after (data de data)</b> Retorna true se o objeto Date de invocação contiver uma data posterior à especificada por data, caso contrário, retornará false.
2	<b>boolean antes (data de data)</b> Retorna true se o objeto Date de invocação contiver uma data anterior à especificada por data, caso contrário, retornará false.

3	<b>Clone de objeto ()</b> Duplica o objeto Date de chamada.
4	<b>int compareTo (data de data)</b> Compara o valor do objeto invocador com o da data. Retorna 0 se os valores forem iguais. Retorna um valor negativo se o objeto de chamada for anterior à data. Retorna um valor positivo se o objeto de chamada for posterior a data.
5	<b>int compareTo (objeto obj)</b> Opera de forma idêntica para compareTo (Date) se obj for da classe Date. Caso contrário, ele lança um ClassCastException.
6	<b>equals booleanos (data do objeto)</b> Retorna true se o objeto Date de invocação contiver a mesma hora e data que a especificada por data, caso contrário, retornará false.
7	<b>long getTime ()</b> Retorna o número de milissegundos decorridos desde 1º de janeiro de 1970.
8	<b>int hashCode ()</b> Retorna um código hash para o objeto invocador.
9	<b>void setTime (long time)</b> Define a hora e a data conforme especificado por hora, o que representa um tempo decorrido em milissegundos a partir da meia-noite de 1º de janeiro de 1970.
10	<b>String toString ()</b> Converte o objeto Date em uma string e retorna o resultado.

## Obtendo a data e hora atuais

Este é um método muito fácil de obter data e hora atuais em Java. Você pode usar um objeto Date simples com o *método toString ()* para imprimir a data e hora atuais da seguinte maneira -

### Exemplo

```
import java.util.Date;
public class DateDemo {
```



```
public static void main(String args[]) {  
    // Instantiate a Date object  
    Date date = new Date();  
  
    // display time and date using toString()  
    System.out.println(date.toString());  
}  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
on May 04 09:51:52 CDT 2009
```

## Comparação de datas

A seguir estão as três maneiras de comparar duas datas -

Você pode usar `getTime ()` para obter o número de milissegundos decorridos desde meia-noite de 1 de janeiro de 1970 para os dois objetos e comparar esses dois valores.

Você pode usar os métodos `before ()`, `after ()` e `equals ()`. Como o dia 12 do mês vem antes do dia 18, por exemplo, a nova `Data (99, 2, 12) .antes (new Date (99, 2, 18))` retorna `true`.

Você pode usar o método `compareTo ()`, que é definido pela interface `Comparable` e implementado por `Date`.

## Formatação de data usando SimpleDateFormat

`SimpleDateFormat` é uma classe concreta para formatar e analisar datas de uma maneira sensível ao código do idioma. `SimpleDateFormat` permite que você comece escolhendo quaisquer padrões definidos pelo usuário para formatação de data e hora.

## Exemplo

```
import java.util.*;  
import java.text.*;  
  
public class DateDemo {  
  
    public static void main(String args[]) {  
        Date dNow = new Date( );  
        SimpleDateFormat ft =  
            new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");  
  
        System.out.println("Current Date: " + ft.format(dNow));  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
```

## Códigos simples de formato DateFormat

Para especificar o formato de hora, use uma string de padrão de tempo. Neste padrão, todas as letras ASCII são reservadas como letras padrão, que são definidas como as seguintes -

Personagem	Descrição	Exemplo
G	Designador da era	DE ANÚNCIOS
y	Ano em quatro dígitos	2001
M	Mês no ano	Julho ou 07
d	Dia no mês	10
h	Hora em AM / PM (1 ~ 12)	12
H	Hora no dia (0 ~ 23)	22
m	Minuto em hora	30
s	Segundo em minuto	55
S	Milissegundo	234
E	Dia na semana	terça
D	Dia no ano	360
F	Dia da semana no mês	2 (segunda quarta em julho)
W	Semana no ano	40
W	Semana no mês	1
uma	Marcador AM / PM	PM
k	Hora no dia (1 ~ 24)	24
K	Hora em AM / PM (0 ~ 11)	10
z	Fuso horário	Horário Padrão do Leste
'	Escapar para o texto	Delimitador
"	Citação única	`

## Formatação de data usando printf

A formatação de data e hora pode ser feita facilmente usando o método **printf**. Você usa um formato de duas letras, começando com **te** terminando em uma das letras da tabela, conforme mostrado no código a seguir.

## Exemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date
        String str = String.format("Current Date/Time : %tc", date );

        System.out.printf(str);
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

Seria um pouco bobo se você tivesse que fornecer a data várias vezes para formatar cada parte. Por esse motivo, uma string de formato pode indicar o índice do argumento a ser formatado.

O índice deve seguir imediatamente o% e deve ser terminado por um \$.

## Exemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date
        System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due date:", date);
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Due date: February 09, 2004
```

Alternativamente, você pode usar o <flag. Indica que o mesmo argumento da especificação de formato anterior deve ser usado novamente.

## Exemplo

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display formatted date
    }
}
```

```
System.out.printf("%s %tB %<te, %<tY", "Due date:", date);  
}  
}
```

Isso produzirá o seguinte resultado -

## Saída

Due date: February 09, 2004

## Caracteres de conversão de data e hora

Personagem	Descrição	Exemplo
c	Data e hora completas	Seg 04 de maio 09:51:52 CDT 2009
F	Data ISO 8601	2004-02-09
D	Data de formatação dos EUA (mês / dia / ano)	02/09/2004
T	24 horas	18:05:19
r	12 horas	06:05:19 pm
R	24 horas, sem segundos	18:05
Y	Ano de quatro dígitos (com zeros à esquerda)	2004
y	Últimos dois dígitos do ano (com zeros à esquerda)	04
C	Primeiros dois dígitos do ano (com zeros à esquerda)	20
B	Nome do mês completo	fevereiro
b	Nome do mês abreviado	Fevereiro
m	Mês de dois dígitos (com zeros à esquerda)	02
d	Dia de dois dígitos (com zeros à esquerda)	03
e	Dia de dois dígitos (sem zeros iniciais)	9
UMA	Nome completo do dia da semana	Segunda-feira
uma	Nome abreviado do dia da semana	seg
j	Dia de três dígitos do ano (com zeros à esquerda)	069

H	Hora de dois dígitos (com zeros à esquerda), entre 00 e 23	18
k	Hora de dois dígitos (sem zeros à esquerda), entre 0 e 23	18
Eu	Hora de dois dígitos (com zeros à esquerda), entre 01 e 12	06
eu	Hora de dois dígitos (sem zeros à esquerda), entre 1 e 12	6
M	Minutos de dois dígitos (com zeros à esquerda)	05
S	Segundos de dois dígitos (com zeros à esquerda)	19
eu	Milissegundos de três dígitos (com zeros à esquerda)	047
N	Nanosegundos de nove dígitos (com zeros à esquerda)	047000000
P	Marcador de manhã ou tarde em maiúsculas	PM
p	Manhã minúscula ou marcador da tarde	PM
z	Compensação numérica de RFC 822 do GMT	-0800
Z	Fuso horário	PST
s	Segundos desde 1970-01-01 00:00:00 GMT	1078884319
Q	Milissegundos desde 1970-01-01 00:00:00 GMT	1078884319047

Existem outras classes úteis relacionadas a Data e hora. Para mais detalhes, você pode consultar a documentação do Java Standard.

## Analizando strings em datas

A classe `SimpleDateFormat` possui alguns métodos adicionais, notavelmente `parse()`, que tenta analisar uma string de acordo com o formato armazenado no objeto `SimpleDateFormat` fornecido.

### Exemplo

```
import java.util.*;
import java.text.*;
```

```

public class DateDemo {

    public static void main(String args[]) {
        SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");
        String input = args.length == 0 ? "1818-11-11" : args[0];

        System.out.print(input + " Parses as ");
        Date t;
        try {
            t = ft.parse(input);
            System.out.println(t);
        } catch (ParseException e) {
            System.out.println("Unparseable using " + ft);
        }
    }
}

```

Um exemplo de execução do programa acima produziria o seguinte resultado -

## Saída

```
1818-11-11 Parses as Wed Nov 11 00:00:00 EST 1818
```

## Dormindo por um tempo

Você pode dormir por qualquer período de tempo, de um milésimo de segundo até a vida útil do seu computador. Por exemplo, o programa a seguir iria dormir por 3 segundos -

## Exemplo

```

import java.util.*;
public class SleepDemo {

    public static void main(String args[]) {
        try {
            System.out.println(new Date( ) + "\n");
            Thread.sleep(5*60*10);
            System.out.println(new Date( ) + "\n");
        } catch (Exception e) {
            System.out.println("Got an exception!");
        }
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```
Sun May 03 18:04:41 GMT 2009
Sun May 03 18:04:51 GMT 2009
```

## Medindo o tempo decorrido

Às vezes, você pode precisar medir ponto no tempo em milissegundos. Então, vamos reescrever o exemplo acima mais uma vez -

## Exemplo

```

import java.util.*;
public class DiffDemo {

```

```

public static void main(String args[]) {
    try {
        long start = System.currentTimeMillis( );
        System.out.println(new Date( ) + "\n");

        Thread.sleep(5*60*10);
        System.out.println(new Date( ) + "\n");

        long end = System.currentTimeMillis( );
        long diff = end - start;
        System.out.println("Difference is : " + diff);
    } catch (Exception e) {
        System.out.println("Got an exception!");
    }
}
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Sun May 03 18:16:51 GMT 2009
Sun May 03 18:16:57 GMT 2009
Difference is : 5993

```

## Classe GregorianCalendar

GregorianCalendar é uma implementação concreta de uma classe Calendar que implementa o calendário gregoriano normal com o qual você está familiarizado. Nós não discutimos a classe Calendar neste tutorial, você pode procurar documentação Java padrão para isso.

O método **getInstance ()** do Calendar retorna um GregorianCalendar inicializado com a data e hora atuais no local e no fuso horário padrão. GregorianCalendar define dois campos: AD e BC. Estas representam as duas eras definidas pelo calendário gregoriano.

Existem também vários construtores para objetos GregorianCalendar -

Sr. Não.	Construtor e Descrição
1	<b>Calendário gregoriano()</b> Constrói um GregorianCalendar padrão usando a hora atual no fuso horário padrão com a localidade padrão.
2	<b>GregorianCalendar (ano int, mês int, data int)</b> Constrói um GregorianCalendar com a data especificada definida no fuso horário padrão com a localidade padrão.
3	<b>GregorianCalendar (ano int, mês int, data int, hora int, minuto int)</b> Constrói um GregorianCalendar com a data e hora fornecidas para o fuso horário padrão com a localidade padrão.

Aqui está a lista de alguns métodos de suporte úteis fornecidos pela classe GregorianCalendar -

<b>Sr. Não.</b>	<b>Método e Descrição</b>
1	<b>void add (campo int, int amount)</b> Adiciona o período de tempo especificado (assinado) ao campo de hora determinado, com base nas regras do calendário.
2	<b>computeFields () vazio protegido</b> Converte o UTC como milissegundos em valores de campo de hora.
3	<b>computeTime void protegido ()</b> Substitui o Calendário Converte os valores do campo de tempo em UTC como milissegundos.
4	<b>equals booleanos (objeto obj)</b> Compara este GregorianCalendar a uma referência de objeto.
5	<b>int get (campo int)</b> Obtém o valor para um determinado campo de hora.
6	<b>int getActualMaximum (campo int)</b> Retorna o valor máximo que esse campo poderia ter, dada a data atual.
7	<b>int getActualMinimum (campo int)</b> Retorna o valor mínimo que esse campo poderia ter, considerando a data atual.
8	<b>int getGreatestMinimum (campo int)</b> Retorna o valor mínimo mais alto para o campo especificado, se variar.
9	<b>Data getGregorianChange ()</b> Obtém a data de mudança do calendário gregoriano.
10	<b>int getLeastMaximum (campo int)</b> Retorna o valor máximo mais baixo para o campo especificado, se variar.
11	<b>int getMaximum (campo int)</b> Retorna o valor máximo para o campo especificado.



## Exemplo

```
import java.util.*;
public class GregorianCalendarDemo {

    public static void main(String args[]) {
        String months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
            "Oct", "Nov", "Dec"};

        int year;
        // Create a Gregorian calendar initialized
        // with the current date and time in the
        // default locale and timezone.

        GregorianCalendar gcalendar = new GregorianCalendar();

        // Display current time and date information.
        System.out.print("Date: ");
        System.out.print(months[gcalendar.get(Calendar.MONTH)]);
        System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
        System.out.println(year = gcalendar.get(Calendar.YEAR));
        System.out.print("Time: ");
        System.out.print(gcalendar.get(Calendar.HOUR) + ":");
        System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
        System.out.println(gcalendar.get(Calendar.SECOND));

        // Test if the current year is a leap year
        if(gcalendar.isLeapYear(year)) {
            System.out.println("The current year is a leap year");
        }else {
            System.out.println("The current year is not a leap year");
        }
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Date: Apr 22 2009
Time: 11:25:27
The current year is not a leap year
```

Para obter uma lista completa de constantes disponíveis na classe Calendar, você pode consultar a documentação padrão do Java.

## Java - Expressões Regulares

Java fornece o pacote `java.util.regex` para correspondência de padrões com expressões regulares. As expressões regulares de Java são muito semelhantes à linguagem de programação Perl e são muito fáceis de aprender.

Uma expressão regular é uma sequência especial de caracteres que ajuda você a encontrar ou encontrar outras strings ou conjuntos de strings, usando uma sintaxe especializada mantida em um padrão. Eles podem ser usados para pesquisar, editar ou manipular texto e dados.

O pacote `java.util.regex` consiste principalmente nas três classes a seguir -

**Classe Padrão** - Um objeto Padrão é uma representação compilada de uma

expressão regular. A classe `Pattern` não fornece construtores públicos. Para criar um padrão, você deve primeiro invocar um de seus métodos public static **`compile()`**, que retornará um objeto `Pattern`. Esses métodos aceitam uma expressão regular como o primeiro argumento.

**Classe de Correspondência** - Um objeto de Correspondência é o mecanismo que interpreta o padrão e executa operações de correspondência com uma sequência de entrada. Como a classe `Pattern`, o `Matcher` não define construtores públicos. Você obtém um objeto `Matcher` invocando o método **`matcher()`** em um objeto `Pattern`.

**`PatternSyntaxException`** - Um objeto `PatternSyntaxException` é uma exceção não verificada que indica um erro de sintaxe em um padrão de expressão regular.

## Capturando Grupos

Capturar grupos é uma maneira de tratar vários caracteres como uma única unidade. Eles são criados colocando os caracteres a serem agrupados dentro de um conjunto de parênteses. Por exemplo, a expressão regular `(dog)` cria um único grupo contendo as letras "d", "o" e "g".

Os grupos de captura são numerados contando seus parênteses de abertura da esquerda para a direita. Na expressão `((A) (B (C)))`, por exemplo, existem quatro grupos -

`((ABC)))`

`(UMA)`

`(B (C))`

`(C)`

Para descobrir quantos grupos estão presentes na expressão, chame o método `groupCount` em um objeto correspondente. O método `groupCount` retorna um **`int`** mostrando o número de grupos de captura presentes no padrão do correspondente.

Há também um grupo especial, o grupo 0, que sempre representa a expressão inteira. Este grupo não está incluído no total reportado por `groupCount`.

### Exemplo

O exemplo a seguir ilustra como encontrar uma string numérica a partir da string alfanumérica fornecida -

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    public static void main( String args[] ) {
        // String to be scanned to find the pattern.
        String line = "This order was placed for QT3000! OK?";
        String pattern = "(.*)\\\\d+(.*)";

        // Create a Pattern object
        Pattern r = Pattern.compile(pattern);
```

```
// Now create matcher object.
Matcher m = r.matcher(line);
if (m.find( )) {
    System.out.println("Found value: " + m.group(0) );
    System.out.println("Found value: " + m.group(1) );
    System.out.println("Found value: " + m.group(2) );
}else {
    System.out.println("NO MATCH");
}
}
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

## Sintaxe de Expressão Regular

Aqui está a tabela listando toda a sintaxe do metacaractere de expressão regular disponível em Java -

Subexpressão	Fósforos
^	Corresponde ao início da linha.
\$	Corresponde ao final da linha.
.	Corresponde a qualquer caractere único, exceto a nova linha. Usando <b>m</b> opção permite coincidir com a nova linha também.
[...]	Corresponde a qualquer caractere único entre colchetes.
[^ ...]	Corresponde a qualquer caractere único que não esteja entre colchetes.
\UMA	Começo da cadeia inteira.
\ z	Fim da string inteira.
\ Z	Fim da seqüência inteira, exceto o terminador de linha final permitido.
ré*	Corresponde a 0 ou mais ocorrências da expressão anterior.
re +	Corresponde a 1 ou mais da coisa anterior.
ré?	Corresponde a 0 ou 1 ocorrência da expressão anterior.
re {n}	Corresponde exatamente ao número de ocorrências da expressão anterior.
re {n}	Corresponde a n ou mais ocorrências da expressão anterior.

re {n, m}	Corresponde pelo menos n e no máximo m ocorrências da expressão anterior.
a   b	Corresponde a ou b.
(ré)	Agrupar expressões regulares e lembra o texto correspondente.
(?: re)	Agrupar expressões regulares sem lembrar do texto correspondente.
(?> re)	Corresponde ao padrão independente sem retroceder.
\W	Corresponde aos caracteres da palavra.
\W	Corresponde aos caracteres nonword.
\s	Corresponde ao espaço em branco. Equivalente a [\t\n\r\f].
\S	Corresponde ao não-branco.
\d	Corresponde aos dígitos. Equivalente a [0-9].
\D	Corresponde aos não-dígitos.
\UMA	Corresponde ao início da string.
\Z	Corresponde ao final da string. Se houver uma nova linha, ela corresponderá logo antes da nova linha.
\z	Corresponde ao final da string.
\G	Corresponde ao ponto em que a última partida terminou.
\n	Voltar referência para capturar o número do grupo "n".
\b	Corresponde os limites da palavra quando fora dos colchetes. Corresponde ao backspace (0x08) quando dentro dos colchetes.
\B	Corresponde aos limites nonword.
\n, \t, etc.	Corresponde a novas linhas, retornos de carro, tabulações, etc.
\Q	Escape (quote) todos os caracteres até \E.
\E	Termina a citação iniciada com \Q.

## Métodos da Classe Matcher

Aqui está uma lista de métodos de instância úteis -

### Métodos de Indexação

Os métodos de índice fornecem valores de índice úteis que mostram precisamente onde a correspondência foi encontrada na string de entrada -

Sr.	Método e Descrição
-----	--------------------

<b>Não.</b>	
1	<b>public int start ()</b> Retorna o índice inicial da partida anterior.
2	<b>int int start (int group)</b> Retorna o índice inicial da subsequência capturada pelo grupo determinado durante a operação de correspondência anterior.
3	<b>public int end ()</b> Retorna o deslocamento após o último caractere correspondido.
4	<b>public int end (int group)</b> Retorna o deslocamento após o último caractere da subsequência capturado pelo grupo determinado durante a operação de correspondência anterior.

## Métodos de estudo

Métodos de estudo revisam a string de entrada e retornam um booleano indicando se o padrão é encontrado ou não -

<b>Sr. Não.</b>	<b>Método e Descrição</b>
1	<b>public boolean lookingAt ()</b> Tenta corresponder a sequência de entrada, começando no início da região, contra o padrão.
2	<b>achado booleano público ()</b> Tenta encontrar a subsequência seguinte da sequência de entrada que corresponde ao padrão.
3	<b>achado booleano público (início int)</b> Redefine esse correspondente e, em seguida, tenta localizar a próxima subsequência da sequência de entrada que corresponde ao padrão, iniciando no índice especificado.
4	<b>partidas booleanas públicas ()</b> Tenta combinar a região inteira com o padrão.

## Métodos de Substituição

Os métodos de substituição são métodos úteis para substituir texto em uma string de entrada -

Sr. Não.	Método e Descrição
1	<b>public Matcher appendReplacement (StringBuffer sb, substituição de String)</b> Implementa uma etapa de anexação e substituição não terminal.
2	<b>public StringBuffer appendTail (StringBuffer sb)</b> Implementa uma etapa de anexação e substituição de terminal.
3	<b>public String replaceAll (Substituição de String)</b> Substitui cada subsequência da sequência de entrada que corresponde ao padrão com a sequência de substituição especificada.
4	<b>public String replaceFirst (substituição de string)</b> Substitui a primeira subsequência da sequência de entrada que corresponde ao padrão com a sequência de substituição especificada.
5	<b>public static String quoteReplacement (cadeia s)</b> Retorna uma string de substituição literal para a String especificada. Este método produz uma String que vai funcionar como uma substituição literal <b>s</b> no método appendReplacement da classe Matcher.

## Os métodos de início e fim

A seguir, o exemplo que conta o número de vezes que a palavra "cat" aparece na string de entrada -

### Exemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT = "cat cat cat cattie cat";

    public static void main( String args[] ) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);    // get a matcher object
        int count = 0;

        while(m.find()) {
            count++;
            System.out.println("Match number "+count);
        }
    }
}
```

```
        System.out.println("start(): "+m.start());
        System.out.println("end(): "+m.end());
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

Você pode ver que este exemplo usa limites de palavras para garantir que as letras "c" "a" "t" não sejam meramente uma substring em uma palavra mais longa. Ele também fornece algumas informações úteis sobre onde a string de entrada ocorreu a correspondência.

O método `start` retorna o índice inicial da subsequência capturada pelo grupo dado durante a operação de correspondência anterior, e o `end` retorna o índice do último caractere correspondido, mais um.

### As correspondências e os métodos `lookingAt`

As correspondências e os métodos `lookingAt` tentam corresponder uma sequência de entrada a um padrão. A diferença, no entanto, é que as correspondências exigem que toda a sequência de entrada seja correspondida, enquanto que a `lookingAt` não corresponde.

Ambos os métodos sempre começam no início da string de entrada. Aqui está o exemplo explicando a funcionalidade -

### Exemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main( String args[] ) {
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
    }
}
```

```
System.out.println("Current INPUT is: "+INPUT);

System.out.println("lookingAt(): "+matcher.lookingAt());
System.out.println("matches(): "+matcher.matches());
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Current REGEX is: foo
Current INPUT is: fofooooooooooooooooooooo
lookingAt(): true
matches(): false
```

## Os métodos replaceFirst e replaceAll

Os métodos replaceFirst e replaceAll substituem o texto que corresponde a uma determinada expressão regular. Como seus nomes indicam, replaceFirst substitui a primeira ocorrência e replaceAll substitui todas as ocorrências.

Aqui está o exemplo explicando a funcionalidade -

### Exemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " + "All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);

        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
The cat says meow. All cats say meow.
```

## Os métodos appendReplacement e appendTail

A classe Matcher também fornece métodos appendReplacement e appendTail para substituição de texto.

Aqui está o exemplo explicando a funcionalidade -

### Exemplo



```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {

    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";
    public static void main(String[] args) {

        Pattern p = Pattern.compile(REGEX);

        // get a matcher object
        Matcher m = p.matcher(INPUT);
        StringBuffer sb = new StringBuffer();
        while(m.find()) {
            m.appendReplacement(sb, REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
-foo-foo-foo-
```

## Métodos de classe PatternSyntaxException

Um PatternSyntaxException é uma exceção não verificada que indica um erro de sintaxe em um padrão de expressão regular. A classe PatternSyntaxException fornece os seguintes métodos para ajudá-lo a determinar o que deu errado -

Sr. Não.	Método e Descrição
1	<b>public String getDescription ()</b> Recupera a descrição do erro.
2	<b>public int getIndex ()</b> Recupera o índice de erros.
3	<b>public String getPattern ()</b> Recupera o padrão de expressão regular incorreto.
4	<b>public String getMessage ()</b> Retorna uma cadeia de várias linhas contendo a descrição do erro de sintaxe e seu índice, o padrão de expressão regular errôneo e uma indicação visual do índice de erro dentro do padrão.

# Java - Métodos

Um método Java é uma coleção de instruções que são agrupadas para executar uma operação. Quando você chama o System.out. **O** método **println ()** , por exemplo, o sistema na verdade executa várias instruções para exibir uma mensagem no console.

Agora você aprenderá como criar seus próprios métodos com ou sem valores de retorno, invocar um método com ou sem parâmetros e aplicar abstração de método no design do programa.

## Criando Método

Considerando o exemplo a seguir para explicar a sintaxe de um método -

### Sintaxe

```
public static int methodName(int a, int b) {  
    // body  
}
```

Aqui,

**pública estática** - modificador

**int** - tipo de retorno

**methodName** - nome do método

**a, b** - parâmetros formais

**int a, int b** - lista de parâmetros

A definição do método consiste em um cabeçalho de método e um corpo de método. O mesmo é mostrado na seguinte sintaxe -

### Sintaxe

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

A sintaxe mostrada acima inclui -

**modificador** - Define o tipo de acesso do método e é opcional para uso.

**returnType** - O método pode retornar um valor.

**nameOfMethod** - Esse é o nome do método. A assinatura do método consiste no nome do método e na lista de parâmetros.

**Lista de Parâmetros** - A lista de parâmetros, é o tipo, ordem e número de parâmetros de um método. Estes são opcionais, o método pode conter zero parâmetros.

**corpo do método** - O corpo do método define o que o método faz com as instruções.

## Exemplo

Aqui está o código fonte do método acima definido chamado **min ()** . Esse método usa dois parâmetros num1 e num2 e retorna o máximo entre os dois -

```
/** the snippet returns the minimum between two numbers */  
  
public static int minFunction(int n1, int n2) {  
    int min;  
    if (n1 > n2)  
        min = n2;  
    else  
        min = n1;  
  
    return min;  
}
```

## Chamada de Método

Para usar um método, ele deve ser chamado. Existem duas maneiras pelas quais um método é chamado, ou seja, o método retorna um valor ou não retorna nada (nenhum valor de retorno).

O processo de chamada de método é simples. Quando um programa invoca um método, o controle do programa é transferido para o método chamado. Esse método chamado retorna o controle ao chamador em duas condições, quando -

- a instrução de retorno é executada.

- Atinge o fechamento final do método.

Os métodos retornando void são considerados como chamadas para uma instrução. Vamos considerar um exemplo -

```
System.out.println("This is tutorialspoint.com!");
```

O valor de retorno do método pode ser entendido pelo seguinte exemplo -

```
int result = sum(6, 9);
```

A seguir, o exemplo para demonstrar como definir um método e como chamá-lo -

## Exemplo

```
public class ExampleMinNumber {  
  
    public static void main(String[] args) {  
        int a = 11;  
        int b = 6;  
        int c = minFunction(a, b);  
        System.out.println("Minimum Value = " + c);  
    }  
  
    /** returns the minimum of two numbers */  
    public static int minFunction(int n1, int n2) {  
        int min;  
        if (n1 > n2)  
            min = n2;  
    }  
}
```

```
        else
            min = n1;

        return min;
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Minimum value = 6
```

## O vazio

A palavra-chave `void` nos permite criar métodos que não retornam um valor. Aqui, no exemplo a seguir, estamos considerando um método `void` *methodRankPoints*. Este método é um método vazio, que não retorna nenhum valor. Chamada para um método `void` deve ser uma instrução ie *methodRankPoints (255.7);*. É uma instrução Java que termina com um ponto-e-vírgula, conforme mostrado no exemplo a seguir.

### Exemplo

```
public class ExampleVoid {

    public static void main(String[] args) {
        methodRankPoints(255.7);
    }

    public static void methodRankPoints(double points) {
        if (points >= 202.5) {
            System.out.println("Rank:A1");
        }else if (points >= 122.4) {
            System.out.println("Rank:A2");
        }else {
            System.out.println("Rank:A3");
        }
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Rank:A1
```

## Passando Parâmetros por Valor

Enquanto trabalha no processo de chamada, os argumentos devem ser passados. Estes devem estar na mesma ordem que seus respectivos parâmetros na especificação do método. Os parâmetros podem ser passados por valor ou por referência.

Passar Parâmetros por Valor significa chamar um método com um parâmetro. Através disso, o valor do argumento é passado para o parâmetro.

### Exemplo

O programa a seguir mostra um exemplo de passagem de parâmetro por valor. Os valores

dos argumentos permanecem os mesmos, mesmo após a invocação do método.

```
public class swappingExample {  
  
    public static void main(String[] args) {  
        int a = 30;  
        int b = 45;  
        System.out.println("Before swapping, a = " + a + " and b = " + b);  
  
        // Invoke the swap method  
        swapFunction(a, b);  
        System.out.println("\n**Now, Before and After swapping values will be same here**");  
        System.out.println("After swapping, a = " + a + " and b is " + b);  
    }  
  
    public static void swapFunction(int a, int b) {  
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);  
  
        // Swap n1 with n2  
        int c = a;  
        a = b;  
        b = c;  
        System.out.println("After swapping(Inside), a = " + a + " b = " + b);  
    }  
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Before swapping, a = 30 and b = 45  
Before swapping(Inside), a = 30 b = 45  
After swapping(Inside), a = 45 b = 30  
  
**Now, Before and After swapping values will be same here**:  
After swapping, a = 30 and b is 45
```

## Sobrecarga de método

Quando uma classe possui dois ou mais métodos com o mesmo nome, mas com parâmetros diferentes, ela é conhecida como sobrecarga de método. É diferente de ignorar. Ao substituir, um método tem o mesmo nome de método, tipo, número de parâmetros, etc.

Vamos considerar o exemplo discutido anteriormente para encontrar números mínimos do tipo inteiro. Se, digamos que queremos encontrar o número mínimo de tipo duplo. Em seguida, o conceito de sobrecarga será introduzido para criar dois ou mais métodos com o mesmo nome, mas com parâmetros diferentes.

O exemplo a seguir explica o mesmo -

### Exemplo

```
public class ExampleOverloading {  
  
    public static void main(String[] args) {  
        int a = 11;  
        int b = 6;
```

```

double c = 7.3;
double d = 9.4;
int result1 = minFunction(a, b);

// same function name with different parameters
double result2 = minFunction(c, d);
System.out.println("Minimum Value = " + result1);
System.out.println("Minimum Value = " + result2);
}

// for integer
public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}

// for double
public static double minFunction(double n1, double n2) {
    double min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
}

```

Isso produzirá o seguinte resultado -

### Saída

```

Minimum Value = 6
Minimum Value = 7.3

```

Métodos de sobrecarga tornam o programa legível. Aqui, dois métodos são dados com o mesmo nome, mas com parâmetros diferentes. O número mínimo dos tipos inteiro e duplo é o resultado.

## Usando argumentos de linha de comando

Às vezes você vai querer passar algumas informações em um programa quando você executá-lo. Isso é feito passando os argumentos da linha de comando para main ().

Um argumento de linha de comando é a informação que segue diretamente o nome do programa na linha de comando quando é executado. Acessar os argumentos da linha de comandos dentro de um programa Java é muito fácil. Eles são armazenados como strings no array String passado para main ().

### Exemplo

O programa a seguir exibe todos os argumentos de linha de comando com os quais é chamado -

```

public class CommandLine {

```

```
public static void main(String args[]) {
    for(int i = 0; i<args.length; i++) {
        System.out.println("args[" + i + "]: " + args[i]);
    }
}
```

Tente executar este programa como mostrado aqui -

```
$java CommandLine this is a command line 200 -100
```

Isso produzirá o seguinte resultado -

### Saída

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

## Os construtores

Um construtor inicializa um objeto quando é criado. Ele tem o mesmo nome de sua classe e é sintaticamente semelhante a um método. No entanto, os construtores não possuem um tipo de retorno explícito.

Normalmente, você usará um construtor para fornecer valores iniciais às variáveis de instância definidas pela classe ou para executar qualquer outro procedimento de inicialização necessário para criar um objeto totalmente formado.

Todas as classes têm construtores, independentemente de você definir um ou não, porque o Java fornece automaticamente um construtor padrão que inicializa todas as variáveis de membro para zero. No entanto, depois de definir seu próprio construtor, o construtor padrão não será mais usado.

### Exemplo

Aqui está um exemplo simples que usa um construtor sem parâmetros -

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass() {
        x = 10;
    }
}
```

Você terá que chamar o construtor para inicializar objetos da seguinte maneira -

```
public class ConsDemo {
```

```
public static void main(String args[]) {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();
    System.out.println(t1.x + " " + t2.x);
}
}
```

### Saída

```
10 10
```

## Construtor Parametrizado

Na maioria das vezes, você precisará de um construtor que aceite um ou mais parâmetros. Os parâmetros são adicionados a um construtor da mesma forma que são adicionados a um método, apenas os declare dentro dos parênteses após o nome do construtor.

### Exemplo

Aqui está um exemplo simples que usa um construtor com um parâmetro -

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}
```

Você precisará chamar um construtor para inicializar objetos da seguinte maneira -

```
public class ConsDemo {

    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

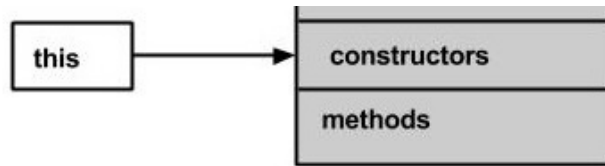
```
10 20
```

## A palavra-chave this

**esta** é uma palavra-chave em Java que é usada como uma referência ao objeto da classe atual, com um método de instância ou um construtor. Usando *isso*, você pode referenciar os membros de uma classe, como construtores, variáveis e métodos.

**Nota** - A palavra - chave *this* é usada apenas dentro de métodos de instâncias ou construtores





Em geral, a palavra - chave é usada para -

Diferencie as variáveis de instância das variáveis locais se elas tiverem nomes iguais, dentro de um construtor ou método.

```
class Student {  
    int age;  
    Student(int age) {  
        this.age = age;  
    }  
}
```

Chame um tipo de construtor (construtor parametrizado ou padrão) de outro em uma classe. É conhecido como invocação de construtor explícito.

```
class Student {  
    int age  
    Student() {  
        this(20);  
    }  
  
    Student(int age) {  
        this.age = age;  
    }  
}
```

## Exemplo

Aqui está um exemplo que usa essa palavra-chave para acessar os membros de uma classe. Copie e cole o programa a seguir em um arquivo com o nome **This\_Example.java** .

```
public class This_Example {  
    // Instance variable num  
    int num = 10;  
  
    This_Example() {  
        System.out.println("This is an example program on keyword this");  
    }  
  
    This_Example(int num) {  
        // Invoking the default constructor  
        this();  
  
        // Assigning the local variable num to the instance variable num  
        this.num = num;  
    }  
  
    public void greet() {  
        System.out.println("Hi Welcome to Tutorialspoint");  
    }  
  
    public void print() {  
        // Local variable num  
        int num = 20;
```

```

    // Printing the local variable
    System.out.println("value of local variable num is : "+num);

    // Printing the instance variable
    System.out.println("value of instance variable num is : "+this.num);

    // Invoking the greet method of a class
    this.greet();
}

public static void main(String[] args) {
    // Instantiating the class
    This_Example obj1 = new This_Example();

    // Invoking the print method
    obj1.print();

    // Passing a new value to the num variable through parametrized constructor
    This_Example obj2 = new This_Example(30);

    // Invoking the print method again
    obj2.print();
}
}

```

Isso produzirá o seguinte resultado -

### Saída

```

This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 10
Hi Welcome to Tutorialspoint
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 30
Hi Welcome to Tutorialspoint

```

## Argumentos Variáveis (var-args)

O JDK 1.5 permite que você passe um número variável de argumentos do mesmo tipo para um método. O parâmetro no método é declarado da seguinte maneira -

```
typeName... parameterName
```

Na declaração do método, você especifica o tipo seguido por reticências (...). Somente um parâmetro de comprimento variável pode ser especificado em um método e esse parâmetro deve ser o último parâmetro. Quaisquer parâmetros regulares devem precedê-lo.

### Exemplo

```

public class VarargsDemo {

    public static void main(String args[]) {
        // Call method with variable args
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }
}

```

```

    }

    public static void printMax( double... numbers) {
        if (numbers.length == 0) {
            System.out.println("No argument passed");
            return;
        }

        double result = numbers[0];

        for (int i = 1; i < numbers.length; i++)
            if (numbers[i] > result)
                result = numbers[i];
        System.out.println("The max value is " + result);
    }
}

```

Isso produzirá o seguinte resultado -

### Saída

```

The max value is 56.5
The max value is 3.0

```

## O método finalize ()

É possível definir um método que será chamado logo antes da destruição final de um objeto pelo coletor de lixo. Esse método é chamado **finalize ()** e pode ser usado para garantir que um objeto seja encerrado corretamente.

Por exemplo, você pode usar finalize () para certificar-se de que um arquivo aberto pertencente a esse objeto esteja fechado.

Para adicionar um finalizador a uma classe, você simplesmente define o método finalize (). O Java runtime chama esse método sempre que está prestes a reciclar um objeto dessa classe.

Dentro do método finalize (), você especificará as ações que devem ser executadas antes que um objeto seja destruído.

O método finalize () tem esta forma geral -

```

protected void finalize( ) {
    // finalization code here
}

```

Aqui, a palavra-chave protected é um especificador que impede o acesso a finalize () pelo código definido fora de sua classe.

Isto significa que você não pode saber quando ou mesmo se finalize () será executado. Por exemplo, se o seu programa terminar antes da coleta de lixo ocorrer, finalize () não será executado.

## Java - Arquivos e I / O

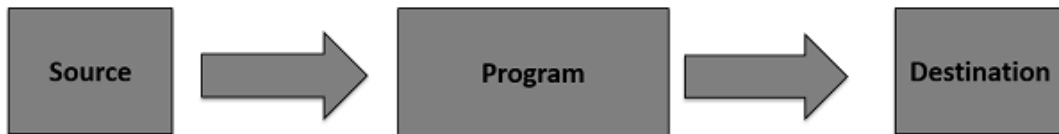
O pacote java.io contém quase todas as classes que você pode precisar para executar entrada e saída (E / S) em Java. Todos esses fluxos representam uma fonte de entrada e

um destino de saída. O fluxo no pacote java.io suporta muitos dados, como primitivos, objeto, caracteres localizados etc.

Um fluxo pode ser definido como uma sequência de dados. Existem dois tipos de Corrente fluxos -

**InputStream** - O InputStream é usado para ler dados de uma fonte.

**OutputStream** - O OutputStream é usado para gravar dados em um destino.



O Java fornece suporte forte, mas flexível, para E / S relacionadas a arquivos e redes, mas este tutorial aborda funcionalidades muito básicas relacionadas a fluxos e E / S. Vamos ver os exemplos mais usados um por um -

## Byte Streams

Os fluxos de bytes Java são usados para executar entrada e saída de bytes de 8 bits. Embora existam muitas classes relacionadas a fluxos de bytes, as classes mais usadas são **FileInputStream** e **FileOutputStream**. A seguir, um exemplo que faz uso dessas duas classes para copiar um arquivo de entrada em um arquivo de saída -

### Exemplo

```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Agora vamos ter um arquivo **input.txt** com o seguinte conteúdo -

```
This is test for copy file.
```

Como próximo passo, compile o programa acima e execute-o, o que resultará na criação do arquivo output.txt com o mesmo conteúdo que temos em input.txt. Então vamos colocar

o código acima no arquivo CopyFile.java e fazer o seguinte -

```
$javac CopyFile.java
$java CopyFile
```

## Fluxos de caracteres

Os fluxos de Java **Byte** são usados para executar entrada e saída de bytes de 8 bits, enquanto fluxos de **caracteres** Java são usados para executar entrada e saída para unicode de 16 bits. Embora existam muitas classes relacionadas a fluxos de caracteres, mas as classes mais usadas são, **FileReader** e **FileWriter** . Embora internamente FileReader usa FileInputStream e FileWriter usa FileOutputStream, mas aqui a principal diferença é que FileReader lê dois bytes de cada vez e FileWriter grava dois bytes de cada vez.

Podemos reescrever o exemplo acima, o que faz com que o uso dessas duas classes copie um arquivo de entrada (com caracteres unicode) em um arquivo de saída -

### Exemplo

```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Agora vamos ter um arquivo **input.txt** com o seguinte conteúdo -

```
This is test for copy file.
```

Como próximo passo, compile o programa acima e execute-o, o que resultará na criação do arquivo output.txt com o mesmo conteúdo que temos em input.txt. Então vamos colocar o código acima no arquivo CopyFile.java e fazer o seguinte -

```
$javac CopyFile.java
$java CopyFile
```

## Correntes Padrão

Todas as linguagens de programação fornecem suporte para E / S padrão, onde o programa do usuário pode receber entrada de um teclado e produzir uma saída na tela do computador. Se você está ciente de linguagens de programação C ou C ++, então você deve estar ciente de três dispositivos padrão STDIN, STDOUT e STDERR. Da mesma forma, o Java fornece os seguintes três fluxos padrão -

**Entrada padrão** - É usada para alimentar os dados para o programa do usuário e, geralmente, um teclado é usado como fluxo de entrada padrão e representado como **System.in** .

**Saída Padrão** - É usada para produzir os dados produzidos pelo programa do usuário e, geralmente, uma tela de computador é usada para o fluxo de saída padrão e representada como **System.out** .

**Erro Padrão** - É usado para gerar os dados de erro produzidos pelo programa do usuário e, geralmente, uma tela de computador é usada para o fluxo de erros padrão e representada como **System.err** .

A seguir está um programa simples, que cria **InputStreamReader** para ler o fluxo de entrada padrão até que o usuário digite um "q" -

### Exemplo

```
import java.io.*;
public class ReadConsole {

    public static void main(String args[]) throws IOException {
        InputStreamReader cin = null;

        try {
            cin = new InputStreamReader(System.in);
            System.out.println("Enter characters, 'q' to quit.");
            char c;
            do {
                c = (char) cin.read();
                System.out.print(c);
            } while(c != 'q');
        } finally {
            if (cin != null) {
                cin.close();
            }
        }
    }
}
```

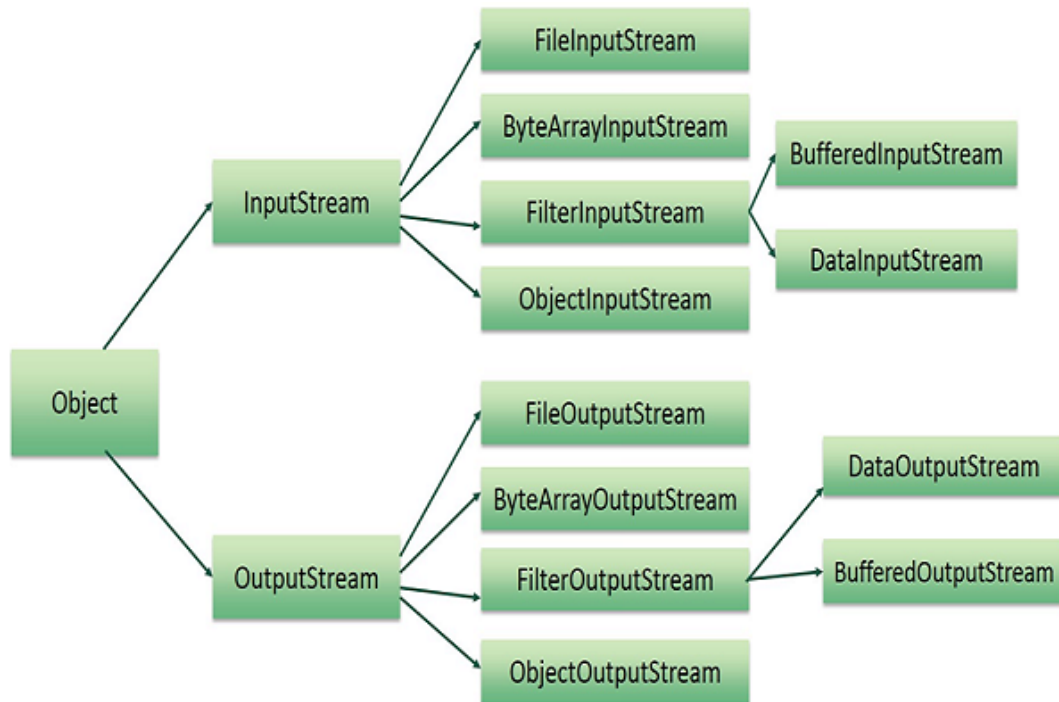
Vamos manter o código acima no arquivo ReadConsole.java e tentar compilá-lo e executá-lo conforme mostrado no programa a seguir. Este programa continua a ler e produzir o mesmo caractere até pressionar 'q' -

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
```

e  
q  
q

Conforme descrito anteriormente, um fluxo pode ser definido lendo e escrevendo arquivos como uma sequência de dados. O **InputStream** é usado para ler dados de uma fonte e o **OutputStream** é usado para gravar dados em um destino.

Aqui está uma hierarquia de classes para lidar com fluxos de entrada e saída.



Os dois fluxos importantes são **FileInputStream** e **FileOutputStream**, que seriam discutidos neste tutorial.

## FileInputStream

Este fluxo é usado para ler dados dos arquivos. Objetos podem ser criados usando a palavra-chave **new** e existem vários tipos de construtores disponíveis.

O construtor a seguir usa um nome de arquivo como uma string para criar um objeto de fluxo de entrada para ler o arquivo -

```
InputStream f = new FileInputStream("C:/java/hello");
```

O construtor a seguir usa um objeto de arquivo para criar um objeto de fluxo de entrada para ler o arquivo. Primeiro, criamos um objeto de arquivo usando o método `File()` da seguinte maneira:

```
File f = new File("C:/java/hello");  
InputStream f = new FileInputStream(f);
```

Depois que você tiver o objeto *InputStream* em mãos, haverá uma lista de métodos auxiliares que podem ser usados para ler o fluxo ou fazer outras operações no fluxo.

Sr.	Método e Descrição
-----	--------------------

Não.	
1	<b>public void close () lança IOException {}</b> Este método fecha o fluxo de saída do arquivo. Libera quaisquer recursos do sistema associados ao arquivo. Lança uma IOException.
2	<b>void protected finalize () lança IOException {}</b> Este método limpa a conexão com o arquivo. Garante que o método de fechamento desse fluxo de saída de arquivo seja chamado quando não houver mais referências a esse fluxo. Lança uma IOException.
3	<b>public int read (int r) lança IOException {}</b> Este método lê o byte especificado de dados do InputStream. Retorna um int. Retorna o próximo byte de dados e -1 será retornado se for o final do arquivo.
4	<b>public int read (byte [] r) lança IOException {}</b> Este método lê bytes r.length do fluxo de entrada em uma matriz. Retorna o número total de bytes lidos. Se for o final do arquivo, -1 será retornado.
5	<b>public int available () lança IOException {}</b> Dá o número de bytes que podem ser lidos deste fluxo de entrada de arquivo. Retorna um int.

Existem outros fluxos de entrada importantes disponíveis, para mais detalhes você pode consultar os seguintes links -

ByteArrayInputStream

DataInputStream

## FileOutputStream

FileOutputStream é usado para criar um arquivo e gravar dados nele. O fluxo criaria um arquivo, se já não existir, antes de abri-lo para saída.

Aqui estão dois construtores que podem ser usados para criar um objeto FileOutputStream.

O construtor a seguir usa um nome de arquivo como uma string para criar um objeto de fluxo de entrada para gravar o arquivo -

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

O construtor a seguir usa um objeto de arquivo para criar um objeto de fluxo de saída para gravar o arquivo. Primeiro, criamos um objeto de arquivo usando o método File () da seguinte maneira:



```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Depois que você tiver o objeto *OutputStream* em mãos, haverá uma lista de métodos auxiliares, que podem ser usados para gravar no fluxo ou para executar outras operações no fluxo.

Sr. Não.	Método e Descrição
1	<b>public void close () lança IOException {}</b> Este método fecha o fluxo de saída do arquivo. Libera quaisquer recursos do sistema associados ao arquivo. Lança uma IOException.
2	<b>void protected finalize () lança IOException {}</b> Este método limpa a conexão com o arquivo. Garante que o método de fechamento desse fluxo de saída de arquivo seja chamado quando não houver mais referências a esse fluxo. Lança uma IOException.
3	<b>public void write (int w) lança IOException {}</b> Este método grava o byte especificado no fluxo de saída.
4	<b>public void write (byte [] w)</b> Grava bytes w.length da matriz de bytes mencionada para o OutputStream.

Existem outros fluxos de saída importantes disponíveis, para mais detalhes você pode consultar os seguintes links -

ByteArrayOutputStream

DataOutputStream

## Exemplo

A seguir, o exemplo para demonstrar InputStream e OutputStream -

```
import java.io.*;
public class FileStreamTest {

    public static void main(String args[]) {

        try {
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x = 0; x < bWrite.length ; x++) {
                os.write( bWrite[x] );    // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("test.txt");
```

```

int size = is.available();

for(int i = 0; i < size; i++) {
    System.out.print((char)is.read() + " ");
}
is.close();
} catch (IOException e) {
    System.out.print("Exception");
}
}
}

```

O código acima criaria o arquivo test.txt e escreveria números em formato binário. O mesmo seria a saída na tela stdout.

## Navegação de arquivos e E / S

Existem várias outras classes que estaríamos passando para conhecer os conceitos básicos de Navegação de Arquivos e E / S.

Classe de arquivo

Classe FileReader

Classe FileWriter

## Diretórios em Java

Um diretório é um arquivo que pode conter uma lista de outros arquivos e diretórios. Você usa o objeto **File** para criar diretórios, para listar os arquivos disponíveis em um diretório. Para obter detalhes completos, verifique uma lista de todos os métodos que você pode chamar no objeto Arquivo e quais estão relacionados aos diretórios.

### Criando diretórios

Existem dois métodos úteis do utilitário **File** , que podem ser usados para criar diretórios -

O método **mkdir ()** cria um diretório, retornando verdadeiro em sucesso e falso em falha. Falha indica que o caminho especificado no objeto File já existe ou que o diretório não pode ser criado porque o caminho inteiro ainda não existe.

O método **mkdirs ()** cria um diretório e todos os pais do diretório.

O exemplo a seguir cria o diretório "/ tmp / user / java / bin" -

### Exemplo

```

import java.io.File;
public class CreateDir {

    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);

        // Create directory now.
        d.mkdirs();
    }
}

```

Compile e execute o código acima para criar "/ tmp / user / java / bin".

**Nota** - O Java cuida automaticamente dos separadores de caminho no UNIX e Windows conforme as convenções. Se você usar uma barra (/) em uma versão do Java do Windows, o caminho ainda será resolvido corretamente.

## Listando Diretórios

Você pode usar o método **list ()** fornecido pelo objeto **File** para listar todos os arquivos e diretórios disponíveis em um diretório da seguinte maneira -

### Exemplo

```
import java.io.File;
public class ReadDir {

    public static void main(String[] args) {
        File file = null;
        String[] paths;

        try {
            // create new file object
            file = new File("/tmp");

            // array of files and directory
            paths = file.list();

            // for each name in the path array
            for(String path:paths) {
                // prints filename and directory name
                System.out.println(path);
            }
        } catch (Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

Isso produzirá o seguinte resultado com base nos diretórios e arquivos disponíveis em seu **diretório / tmp** -

### Saída

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

## Java - exceções

Uma exceção (ou evento excepcional) é um problema que surge durante a execução de um programa. Quando ocorre uma **exceção**, o fluxo normal do programa é interrompido e o programa / aplicativo é finalizado de forma anormal, o que não é recomendado, portanto, essas exceções devem ser tratadas.

Uma exceção pode ocorrer por diversos motivos. A seguir, alguns cenários em que ocorre uma exceção.

Um usuário inseriu um dado inválido.

Um arquivo que precisa ser aberto não pode ser encontrado.

Uma conexão de rede foi perdida no meio das comunicações ou a JVM ficou sem memória.

Algumas dessas exceções são causadas por erro do usuário, outras por erro do programador e outras por recursos físicos que falharam de alguma maneira.

Com base neles, temos três categorias de exceções. Você precisa entendê-los para saber como o tratamento de exceções funciona em Java.

**Exceções verificadas** - Uma exceção verificada é uma exceção que ocorre no momento da compilação. Essas exceções também são chamadas como exceções de tempo de compilação. Essas exceções não podem simplesmente ser ignoradas no momento da compilação, o programador deve tomar cuidado com essas exceções.

Por exemplo, se você usar a classe **FileReader** em seu programa para ler dados de um arquivo, se o arquivo especificado em seu construtor não existir, ocorrerá uma *FileNotFoundException* e o compilador solicitará que o programador manipule a exceção.

## Exemplo

```
import java.io.File;
import java.io.FileReader;

public class FileNotFound_Demo {

    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

Se você tentar compilar o programa acima, você receberá as seguintes exceções.

## Saída

```
C:\>javac FileNotFound_Demo.java
FileNotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
        FileReader fr = new FileReader(file);
        ^
1 error
```

**Nota** - Uma vez que os métodos **read ()** e **close ()** da classe *FileReader* lançam *IOException*, você pode observar que o compilador notifica para manipular *IOException*, junto com *FileNotFoundException*.

**Exceções não verificadas** - Uma exceção não verificada é uma exceção que ocorre no momento da execução. Eles também são chamados de **exceções de tempo de execução**. Isso inclui erros de programação, como erros lógicos ou uso indevido de uma API. Exceções de tempo de execução são ignoradas no

momento da compilação.

Por exemplo, se você tiver declarado uma matriz de tamanho 5 em seu programa e tentar chamar o 6º elemento da matriz, ocorrerá uma exceção *ArrayIndexOutOfBoundsException*.

## Exemplo

```
public class Unchecked_Demo {  
  
    public static void main(String args[]) {  
        int num[] = {1, 2, 3, 4};  
        System.out.println(num[5]);  
    }  
}
```

Se você compilar e executar o programa acima, você receberá a seguinte exceção.

## Saída

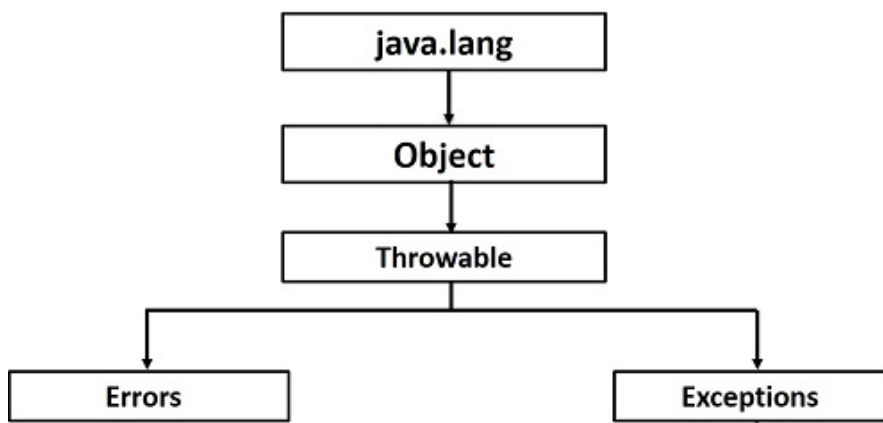
```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5  
    at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

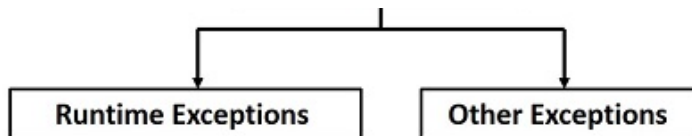
**Erros** - Estas não são exceções, mas problemas que surgem além do controle do usuário ou do programador. Erros são normalmente ignorados em seu código porque você raramente pode fazer algo sobre um erro. Por exemplo, se ocorrer um estouro de pilha, ocorrerá um erro. Eles também são ignorados no momento da compilação.

Todas as classes de exceção são subtipos da classe *Hierarquia de exceções* `java.lang.Exception`. A classe de exceção é uma subclasse da classe `Throwable`. Além da classe de exceção, há outra subclasse chamada `Error`, que é derivada da classe `Throwable`.

Erros são condições anormais que acontecem em caso de falhas graves, estas não são tratadas pelos programas Java. Erros são gerados para indicar erros gerados pelo ambiente de tempo de execução. Exemplo: a JVM está sem memória. Normalmente, os programas não podem se recuperar de erros.

A classe `Exception` possui duas subclasses principais: `IOException` class e `RuntimeException` Class.





A seguir, uma lista das exceções internas mais comuns verificadas e não verificadas do Java .

## Métodos de Exceções

A seguir, a lista de métodos importantes disponíveis na classe Throwable.

Sr. Não.	Método e Descrição
1	<b>public String getMessage ()</b> Retorna uma mensagem detalhada sobre a exceção ocorrida. Esta mensagem é inicializada no construtor Throwable.
2	<b>public Throwable getCause ()</b> Retorna a causa da exceção, conforme representado por um objeto Throwable.
3	<b>public String toString ()</b> Retorna o nome da classe concatenada com o resultado de getMessage ().
4	<b>public void printStackTrace ()</b> Imprime o resultado de toString () junto com o rastreamento de pilha para System.err, o fluxo de saída de erro.
5	<b>public StackTraceElement [] getStackTrace ()</b> Retorna uma matriz contendo cada elemento no rastreamento de pilha. O elemento no índice 0 representa o topo da pilha de chamadas e o último elemento na matriz representa o método na parte inferior da pilha de chamadas.
6	<b>public Throwable fillInStackTrace ()</b> Preenche o rastreamento de pilha desse objeto Throwable com o rastreamento de pilha atual, adicionando a qualquer informação anterior no rastreamento de pilha.

## Captura de exceções

Um método captura uma exceção usando uma combinação das palavras-chave **try** e **catch** . Um bloco try / catch é colocado em torno do código que pode gerar uma exceção.

O código dentro de um bloco try / catch é chamado de código protegido, e a sintaxe para usar o try / catch é semelhante à seguinte -

## Sintaxe

```
try {  
    // Protected code  
} catch (ExceptionName e1) {  
    // Catch block  
}
```

O código que é propenso a exceções é colocado no bloco try. Quando ocorre uma exceção, essa exceção é tratada pelo bloco catch associado a ela. Cada bloco try deve ser imediatamente seguido por um bloco catch ou por último bloco.

Uma instrução catch envolve declarar o tipo de exceção que você está tentando capturar. Se ocorrer uma exceção no código protegido, o bloco catch (ou blocos) que segue a tentativa é verificado. Se o tipo de exceção que ocorreu for listado em um bloco catch, a exceção será passada para o bloco catch da mesma maneira que um argumento é passado para um parâmetro de método.

## Exemplo

O seguinte é um array declarado com 2 elementos. Em seguida, o código tenta acessar o terceiro elemento da matriz que lança uma exceção.

```
// File Name : ExcepTest.java  
import java.io.*;  
  
public class ExcepTest {  
  
    public static void main(String args[]) {  
        try {  
            int a[] = new int[2];  
            System.out.println("Access element three :" + a[3]);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception thrown :" + e);  
        }  
        System.out.println("Out of the block");  
    }  
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3  
Out of the block
```

## Múltiplos Blocos de Captura

Um bloco try pode ser seguido por vários blocos catch. A sintaxe para vários blocos catch se parece com o seguinte -

## Sintaxe

```
try {
```

```
// Protected code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
} catch (ExceptionType3 e3) {
    // Catch block
}
```

As declarações anteriores demonstram três blocos de captura, mas você pode ter qualquer número deles após uma única tentativa. Se ocorrer uma exceção no código protegido, a exceção será lançada no primeiro bloco catch na lista. Se o tipo de dados da exceção lançada corresponder a `ExceptionType1`, ele será capturado lá. Se não, a exceção passa para a segunda instrução catch. Isso continua até que a exceção seja capturada ou passe por todas as capturas, caso em que o método atual interrompe a execução e a exceção é lançada no método anterior na pilha de chamadas.

## Exemplo

Aqui está o segmento de código mostrando como usar várias instruções try / catch.

```
try {
    file = new FileInputStream(fileName);
    x = (byte) file.read();
} catch (IOException i) {
    i.printStackTrace();
    return -1;
} catch (FileNotFoundException f) // Not valid! {
    f.printStackTrace();
    return -1;
}
```

## Capturando vários tipos de exceções

Desde o Java 7, você pode manipular mais de uma exceção usando um único bloco catch, esse recurso simplifica o código. Aqui está como você faria isso -

```
catch (IOException|FileNotFoundException ex) {
    logger.log(ex);
    throw ex;
```

## As palavras-chave Throws / Throw

Se um método não manipular uma exceção verificada, o método deve declará-lo usando a palavra-chave **throws**. A palavra-chave throws aparece no final da assinatura de um método.

Você pode lançar uma exceção, uma recém-instanciada ou uma exceção que você acabou de capturar, usando a palavra-chave **throw**.

Tente entender a diferença entre throws e throw keywords, *throws* é usado para adiar o tratamento de uma exceção verificada e *throw* é usado para invocar uma exceção explicitamente.



O seguinte método declara que ele lança um RemoteException -

## Exemplo

```
import java.io.*;
public class className {

    public void deposit(double amount) throws RemoteException {
        // Method implementation
        throw new RemoteException();
    }
    // Remainder of class definition
}
```

Um método pode declarar que ele lança mais de uma exceção, caso em que as exceções são declaradas em uma lista separada por vírgulas. Por exemplo, o método a seguir declara que ele lança um RemoteException e um InsufficientFundsException -

## Exemplo

```
import java.io.*;
public class className {

    public void withdraw(double amount) throws RemoteException,
        InsufficientFundsException {
        // Method implementation
    }
    // Remainder of class definition
}
```

## O bloco finalmente

O bloco finally segue um bloco try ou um bloco catch. Um bloco final de código sempre é executado, independentemente da ocorrência de uma exceção.

O uso de um bloco finally permite que você execute qualquer declaração de tipo de limpeza que você deseja executar, não importando o que aconteça no código protegido.

Um bloco finally aparece no final dos blocos catch e tem a seguinte sintaxe -

## Sintaxe

```
try {
    // Protected code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
} catch (ExceptionType3 e3) {
    // Catch block
}finally {
    // The finally block always executes.
}
```

## Exemplo

```

public class ExcepTest {

    public static void main(String args[]) {
        int a[] = new int[2];
        try {
            System.out.println("Access element three :" + a[3]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception thrown  :" + e);
        } finally {
            a[0] = 6;
            System.out.println("First element value: " + a[0]);
            System.out.println("The finally statement is executed");
        }
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed

```

Observe o seguinte -

Uma cláusula catch não pode existir sem uma instrução try.

Não é obrigatório ter cláusulas quando um bloco try / catch estiver presente.

O bloco try não pode estar presente sem cláusula catch ou cláusula finally.

Qualquer código não pode estar presente entre os blocos try, catch, finally.

## O try-with-resources

Geralmente, quando usamos quaisquer recursos como fluxos, conexões, etc., temos que fechá-los explicitamente usando finally block. No programa a seguir, estamos lendo dados de um arquivo usando o **FileReader** e estamos fechando usando finally block.

## Exemplo

```

import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

    public static void main(String args[]) {
        FileReader fr = null;
        try {
            File file = new File("file.txt");
            fr = new FileReader(file); char [] a = new char[50];
            fr.read(a); // reads the content to the array
            for(char c : a)
                System.out.print(c); // prints the characters one by one
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fr.close();
            }
        }
    }
}

```

```

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

**try-with-resources** , também chamado de **gerenciamento automático de recursos** , é um novo mecanismo de tratamento de exceções introduzido no Java 7, que fecha automaticamente os recursos usados no bloco try catch.

Para usar essa declaração, basta declarar os recursos necessários dentro dos parênteses e o recurso criado será fechado automaticamente no final do bloco. A seguir está a sintaxe da instrução try-with-resources.

## Sintaxe

```

try(FileReader fr = new FileReader("file path")) {
    // use the resource
} catch () {
    // body of catch
}
}

```

A seguir está o programa que lê os dados em um arquivo usando a instrução try-with-resources.

## Exemplo

```

import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

    public static void main(String args[]) {
        try(FileReader fr = new FileReader("E://file.txt")) {
            char [] a = new char[50];
            fr.read(a);    // reads the content to the array
            for(char c : a)
                System.out.print(c);    // prints the characters one by one
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Os pontos a seguir devem ser mantidos em mente ao trabalhar com a declaração try-with-resources.

- Para usar uma classe com a instrução try-with-resources, ela deve implementar a interface **AutoCloseable** e o método **close ()** dela é chamado automaticamente no tempo de execução.

- Você pode declarar mais de uma classe na instrução try-with-resources.

- Enquanto você declara várias classes no bloco try da instrução try-with-resources, essas classes são fechadas em ordem inversa.

Exceto a declaração de recursos dentro do parêntese, tudo é igual ao bloco try / catch normal de um bloco try.

O recurso declarado no try é instanciado logo antes do início do try-block.

O recurso declarado no bloco try é implicitamente declarado como final.

## Exceções definidas pelo usuário

Você pode criar suas próprias exceções em Java. Tenha os seguintes pontos em mente ao escrever suas próprias classes de exceção -

Todas as exceções devem ser filhos de Throwable.

Se você deseja gravar uma exceção verificada que é aplicada automaticamente pela Handle ou Declare Rule, é necessário estender a classe Exception.

Se você deseja gravar uma exceção de tempo de execução, é necessário estender a classe RuntimeException.

Podemos definir nossa própria classe de exceção como abaixo -

```
class MyException extends Exception {  
}
```

Você só precisa estender a classe **Exceção** predefinida para criar sua própria Exceção. Estas são consideradas exceções verificadas. A seguinte classe **InsufficientFundsException** é uma exceção definida pelo usuário que estende a classe Exception, tornando-a uma exceção verificada. Uma classe de exceção é como qualquer outra classe, contendo campos e métodos úteis.

### Exemplo

```
// File Name InsufficientFundsException.java  
import java.io.*;  
  
public class InsufficientFundsException extends Exception {  
    private double amount;  
  
    public InsufficientFundsException(double amount) {  
        this.amount = amount;  
    }  
  
    public double getAmount() {  
        return amount;  
    }  
}
```

Para demonstrar usando nossa exceção definida pelo usuário, a seguinte classe CheckingAccount contém um método withdraw () que lança uma InsufficientFundsException.

```
// File Name CheckingAccount.java  
import java.io.*;  
  
public class CheckingAccount {  
    private double balance;
```

```

private int number;

public CheckingAccount(int number) {
    this.number = number;
}

public void deposit(double amount) {
    balance += amount;
}

public void withdraw(double amount) throws InsufficientFundsException {
    if(amount <= balance) {
        balance -= amount;
    }else {
        double needs = amount - balance;
        throw new InsufficientFundsException(needs);
    }
}

public double getBalance() {
    return balance;
}

public int getNumber() {
    return number;
}
}

```

O programa BankDemo a seguir demonstra invocar os métodos deposit () e withdraw () de CheckingAccount.

```

// File Name BankDemo.java
public class BankDemo {

    public static void main(String [] args) {
        CheckingAccount c = new CheckingAccount(101);
        System.out.println("Depositing $500...");
        c.deposit(500.00);

        try {
            System.out.println("\nWithdrawing $100...");
            c.withdraw(100.00);
            System.out.println("\nWithdrawing $600...");
            c.withdraw(600.00);
        } catch (InsufficientFundsException e) {
            System.out.println("Sorry, but you are short $" + e.getAmount());
            e.printStackTrace();
        }
    }
}

```

Compile todos os três arquivos acima e execute o BankDemo. Isso produzirá o seguinte resultado -

## Saída

```
Depositing $500...
```

```
Withdrawing $100...
```

```
Withdrawing $600...
```

```
Sorry, but you are short $200.0
```

```
InsufficientFundsException
    at CheckingAccount.withdraw(CheckingAccount.java:25)
    at BankDemo.main(BankDemo.java:13)
```

## Exceções Comuns

Em Java, é possível definir duas categorias de exceções e erros.

**Exceções da JVM** - Estas são exceções / erros que são exclusiva ou logicamente acionados pela JVM. Exemplos: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`.

**Exceções programáticas** - essas exceções são lançadas explicitamente pelo aplicativo ou pelos programadores da API. Exemplos: `IllegalArgumentException`, `IllegalStateException`.

## Java - Classes internas

Neste capítulo, discutiremos classes internas de Java.

### Classes aninhadas

Em Java, assim como os métodos, as variáveis de uma classe também podem ter outra classe como seu membro. Escrever uma classe dentro de outra é permitido em Java. A classe escrita dentro é chamada de **classe aninhada**, e a classe que contém a classe interna é chamada de **classe externa**.

#### Sintaxe

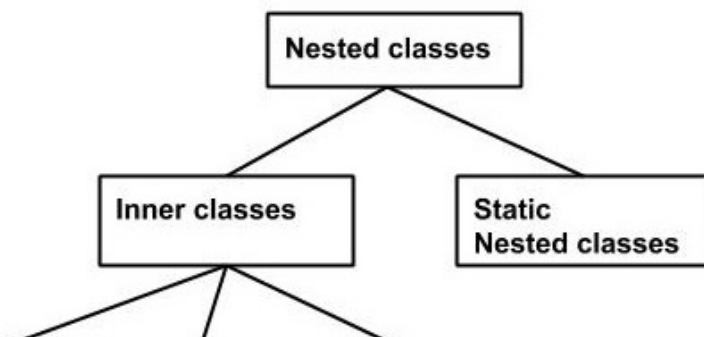
A seguir está a sintaxe para escrever uma classe aninhada. Aqui, a classe **Outer\_Demo** é a classe externa e a classe **Inner\_Demo** é a classe aninhada.

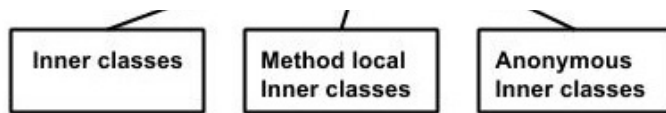
```
class Outer_Demo {
    class Inner_Demo {
    }
}
```

As classes aninhadas são divididas em dois tipos -

**Classes aninhadas não estáticas** - são os membros não estáticos de uma classe.

**Classes aninhadas estáticas** - são os membros estáticos de uma classe.





## Classes internas (classes aninhadas não estáticas)

Classes internas são um mecanismo de segurança em Java. Sabemos que uma classe não pode ser associada ao modificador de acesso **private**, mas se tivermos a classe como membro de outra classe, a classe interna poderá se tornar privada. E isso também é usado para acessar os membros particulares de uma classe.

As classes internas são de três tipos, dependendo de como e onde você as define. Eles são

Classe Interna

Classe Interna Método-local

Classe Interna Anônima

### Classe Interna

Criar uma classe interna é bem simples. Você só precisa escrever uma aula dentro de uma classe. Ao contrário de uma classe, uma classe interna pode ser privada e, quando você declara uma classe interna como privada, ela não pode ser acessada de um objeto fora da classe.

A seguir está o programa para criar uma classe interna e acessá-la. No exemplo dado, tornamos a classe interna privada e acessamos a classe por meio de um método.

### Exemplo

```
class Outer_Demo {
    int num;

    // inner class
    private class Inner_Demo {
        public void print() {
            System.out.println("This is an inner class");
        }
    }

    // Accessing the inner class from the method within
    void display_Inner() {
        Inner_Demo inner = new Inner_Demo();
        inner.print();
    }
}

public class My_class {

    public static void main(String args[]) {
        // Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();

        // Accessing the display_Inner() method.
        outer.display_Inner();
    }
}
```

Aqui você pode observar que **Outer\_Demo** é a classe externa, **Inner\_Demo** é a classe

interna, **display\_Inner ()** é o método dentro do qual estamos instanciando a classe interna, e este método é invocado a partir do método **main** .

Se você compilar e executar o programa acima, você obterá o seguinte resultado -

### Saída

```
This is an inner class.
```

## Acessando os membros privados

Como mencionado anteriormente, as classes internas também são usadas para acessar os membros privados de uma classe. Suponha que uma classe esteja tendo membros privados para acessá-los. Escreva uma classe interna nela, retorne os membros privados de um método dentro da classe interna, digamos, **getValue ()** e, finalmente, de outra classe (da qual você deseja acessar os membros privados) chame o método **getValue ()** do interior classe.

Para instanciar a classe interna, inicialmente você deve instanciar a classe externa. Depois disso, usando o objeto da classe externa, segue a maneira pela qual você pode instanciar a classe interna.

```
Outer_Demo outer = new Outer_Demo();  
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

O programa a seguir mostra como acessar os membros particulares de uma classe usando a classe interna.

### Exemplo

```
class Outer_Demo {  
    // private variable of the outer class  
    private int num = 175;  
  
    // inner class  
    public class Inner_Demo {  
        public int getNum() {  
            System.out.println("This is the getnum method of the inner class");  
            return num;  
        }  
    }  
}  
  
public class My_class2 {  
  
    public static void main(String args[]) {  
        // Instantiating the outer class  
        Outer_Demo outer = new Outer_Demo();  
  
        // Instantiating the inner class  
        Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();  
        System.out.println(inner.getNum());  
    }  
}
```

Se você compilar e executar o programa acima, você obterá o seguinte resultado -

### Saída



```
This is the getnum method of the inner class: 175
```

## Classe Interna Método-local

Em Java, podemos escrever uma classe dentro de um método e isso será um tipo local. Como as variáveis locais, o escopo da classe interna é restrito no método.

Uma classe interna local de método pode ser instanciada somente dentro do método onde a classe interna é definida. O programa a seguir mostra como usar uma classe interna local do método.

### Exemplo

```
public class Outerclass {
    // instance method of the outer class
    void my_Method() {
        int num = 23;

        // method-local inner class
        class MethodInner_Demo {
            public void print() {
                System.out.println("This is method inner class "+num);
            }
        } // end of inner class

        // Accessing the inner class
        MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }

    public static void main(String args[]) {
        Outerclass outer = new Outerclass();
        outer.my_Method();
    }
}
```

Se você compilar e executar o programa acima, você obterá o seguinte resultado -

### Saída

```
This is method inner class 23
```

## Classe Interna Anônima

Uma classe interna declarada sem um nome de classe é conhecida como uma **classe interna anônima**. No caso de classes internas anônimas, nós as declaramos e as instanciamos ao mesmo tempo. Geralmente, eles são usados sempre que você precisa substituir o método de uma classe ou interface. A sintaxe de uma classe interna anônima é a seguinte -

### Sintaxe

```
AnonymousInner an_inner = new AnonymousInner() {
    public void my_method() {
        .....
        .....
    }
}
```

```
};
```

O programa a seguir mostra como substituir o método de uma classe usando a classe interna anônima.

### Exemplo

```
abstract class AnonymousInner {
    public abstract void mymethod();
}

public class Outer_class {

    public static void main(String args[]) {
        AnonymousInner inner = new AnonymousInner() {
            public void mymethod() {
                System.out.println("This is an example of anonymous inner class");
            }
        };
        inner.mymethod();
    }
}
```

Se você compilar e executar o programa acima, você obterá o seguinte resultado -

### Saída

```
This is an example of anonymous inner class
```

Da mesma forma, você pode substituir os métodos da classe concreta, bem como a interface usando uma classe interna anônima.

## Classe Interna Anônima como Argumento

Geralmente, se um método aceita um objeto de uma interface, uma classe abstrata ou uma classe concreta, podemos implementar a interface, estender a classe abstrata e passar o objeto para o método. Se é uma classe, então podemos passá-la diretamente para o método.

Mas em todos os três casos, você pode passar uma classe interna anônima para o método. Aqui está a sintaxe de passar uma classe interna anônima como um argumento de método -

```
obj.my_Method(new My_Class() {
    public void Do() {
        .....
        .....
    }
});
```

O programa a seguir mostra como passar uma classe interna anônima como um argumento de método.

### Exemplo

```
// interface
interface Message {
```

```

    String greet();
}

public class My_class {
    // method which accepts the object of interface Message
    public void displayMessage(Message m) {
        System.out.println(m.greet() +
            ", This is an example of anonymous inner class as an argument");
    }

    public static void main(String args[]) {
        // Instantiating the class
        My_class obj = new My_class();

        // Passing an anonymous inner class as an argument
        obj.displayMessage(new Message() {
            public String greet() {
                return "Hello";
            }
        });
    }
}

```

Se você compilar e executar o programa acima, ele lhe dará o seguinte resultado -

### Saída

```
Hello, This is an example of anonymous inner class as an argument
```

## Classe aninhada estática

Uma classe interna estática é uma classe aninhada que é um membro estático da classe externa. Ele pode ser acessado sem instanciar a classe externa, usando outros membros estáticos. Assim como os membros estáticos, uma classe aninhada estática não tem acesso às variáveis da instância e aos métodos da classe externa. A sintaxe da classe aninhada estática é a seguinte -

### Sintaxe

```

class MyOuter {
    static class Nested_Demo {
    }
}

```

Instanciar uma classe aninhada estática é um pouco diferente de instanciar uma classe interna. O programa a seguir mostra como usar uma classe aninhada estática.

### Exemplo

```

public class Outer {
    static class Nested_Demo {
        public void my_method() {
            System.out.println("This is my nested class");
        }
    }

    public static void main(String args[]) {
        Outer.Nested_Demo nested = new Outer.Nested_Demo();
        nested.my_method();
    }
}

```

```
}
```

Se você compilar e executar o programa acima, você obterá o seguinte resultado -

### Saída

```
This is my nested class
```

## Java - herança

A herança pode ser definida como o processo em que uma classe adquire as propriedades (métodos e campos) de outra. Com o uso de herança, as informações são geridas em ordem hierárquica.

A classe que herda as propriedades de outra é conhecida como subclasse (classe derivada, classe filha) e a classe cujas propriedades são herdadas é conhecida como superclasse (classe base, classe pai).

### estende a palavra-chave

**extends** é a palavra-chave usada para herdar as propriedades de uma classe. A seguir está a sintaxe de extends keyword.

#### Sintaxe

```
class Super {  
    ....  
    ....  
}  
class Sub extends Super {  
    ....  
    ....  
}
```

A seguir, um exemplo demonstrando a herança do Java. Neste exemplo, Código de amostra você pode observar duas classes, a saber, Calculation e My\_Calculation.

Usando a palavra-chave extends, o My\_Calculation herda os métodos addition () e Subtraction () da classe Calculation.

Copie e cole o seguinte programa em um arquivo com o nome My\_Calculation.java

#### Exemplo

```
class Calculation {  
    int z;  
  
    public void addition(int x, int y) {  
        z = x + y;  
        System.out.println("The sum of the given numbers:"+z);  
    }  
  
    public void Subtraction(int x, int y) {  
        z = x - y;  
        System.out.println("The difference between the given numbers:"+z);  
    }  
}
```

```

}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:"+z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}

```

Compile e execute o código acima conforme mostrado abaixo.

```

javac My_Calculation.java
java My_Calculation

```

Depois de executar o programa, ele produzirá o seguinte resultado -

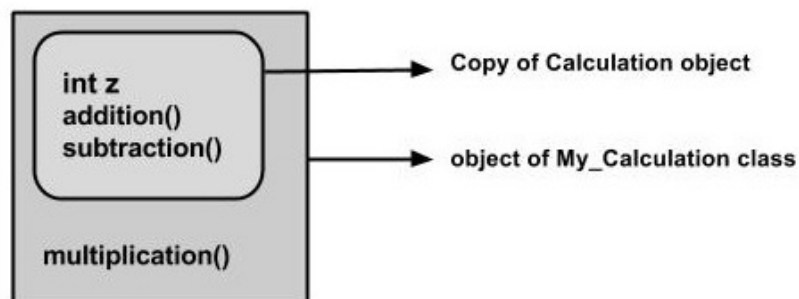
### Saída

```

The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200

```

No programa dado, quando um objeto para a classe **My\_Calculation** é criado, uma cópia do conteúdo da superclasse é feita dentro dele. É por isso que, usando o objeto da subclasse, você pode acessar os membros de uma superclasse.



A variável de referência Superclass pode conter o objeto de subclasse, mas usando essa variável você pode acessar apenas os membros da superclasse, portanto, para acessar os membros de ambas as classes, é recomendável sempre criar uma variável de referência para a subclasse.

Se você considerar o programa acima, você pode instanciar a classe como indicado abaixo. Mas usando a variável de referência da superclasse ( neste caso, **cal** ) você não pode chamar o método multiplation ( ) , que pertence à subclasse My\_Calculation.

```

Calculation demo = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);

```

**Nota** - Uma subclasse herda todos os membros (campos, métodos e classes aninhadas) de

sua superclasse. Os construtores não são membros, portanto, eles não são herdados por subclasses, mas o construtor da superclasse pode ser chamado a partir da subclasse.

## A palavra-chave super

A palavra-chave **super** é semelhante a **essa** palavra - chave. A seguir estão os cenários em que a palavra-chave super é usada.

É usado para **diferenciar os membros** da superclasse dos membros da subclasse, se eles tiverem os mesmos nomes.

É usado para **chamar o** construtor da **superclasse** da subclasse.

### Diferenciando os membros

Se uma classe está herdando as propriedades de outra classe. E se os membros da superclasse tiverem os mesmos nomes da subclasse, para diferenciar essas variáveis, usamos super keyword como mostrado abaixo.

```
super.variable  
super.method();
```

### Código de amostra

Esta seção fornece um programa que demonstra o uso da palavra - chave **super** .

No programa dado, você tem duas classes, *Sub\_class* e *Super\_class* , ambas possuem um método chamado display () com diferentes implementações, e uma variável chamada num com diferentes valores. Estamos invocando o método display () de ambas as classes e imprimindo o valor da variável num das duas classes. Aqui você pode observar que usamos a palavra-chave super para diferenciar os membros da superclasse da subclasse.

Copie e cole o programa em um arquivo com o nome Sub\_class.java.

### Exemplo

```
class Super_class {  
    int num = 20;  
  
    // display method of superclass  
    public void display() {  
        System.out.println("This is the display method of superclass");  
    }  
}  
  
public class Sub_class extends Super_class {  
    int num = 10;  
  
    // display method of sub class  
    public void display() {  
        System.out.println("This is the display method of subclass");  
    }  
  
    public void my_method() {  
        // Instantiating subclass  
        Sub_class sub = new Sub_class();  
  
        // Invoking the display() method of sub class
```

```

    sub.display();

    // Invoking the display() method of superclass
    super.display();

    // printing the value of variable num of subclass
    System.out.println("value of the variable named num in sub class:"+ sub.num);

    // printing the value of variable num of superclass
    System.out.println("value of the variable named num in super class:"+ super.num);
}

public static void main(String args[]) {
    Sub_class obj = new Sub_class();
    obj.my_method();
}
}

```

Compile e execute o código acima usando a seguinte sintaxe.

```

javac Super_Demo
java Super

```

Na execução do programa, você obterá o seguinte resultado -

### Saída

```

This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20

```

## Invocando o Construtor Superclasse

Se uma classe está herdando as propriedades de outra classe, a subclasse adquire automaticamente o construtor padrão da superclasse. Mas se você quiser chamar um construtor parametrizado da superclasse, você precisa usar a palavra-chave super como mostrado abaixo.

```

super(values);

```

## Código de amostra

O programa dado nesta seção demonstra como usar a palavra-chave super para invocar o construtor parametrizado da superclasse. Este programa contém uma superclasse e uma subclasse, onde a superclasse contém um construtor parametrizado que aceita um valor inteiro, e usamos a palavra-chave super para invocar o construtor parametrizado da superclasse.

Copie e cole o seguinte programa em um arquivo com o nome Subclass.java

### Exemplo

```

class Superclass {
    int age;

    Superclass(int age) {
        this.age = age;
    }
}

```

```

    }

    public void getAge() {
        System.out.println("The value of the variable named age in super class is: " +age);
    }
}

public class Subclass extends Superclass {
    Subclass(int age) {
        super(age);
    }

    public static void main(String argd[]) {
        Subclass s = new Subclass(24);
        s.getAge();
    }
}

```

Compile e execute o código acima usando a seguinte sintaxe.

```

javac Subclass
java Subclass

```

Na execução do programa, você obterá o seguinte resultado -

### Saída

```
The value of the variable named age in super class is: 24
```

## Relação IS-A

IS-A é uma maneira de dizer: esse objeto é um tipo desse objeto. Vamos ver como a palavra-chave **extends** é usada para obter herança.

```

public class Animal {
}

classe pública Mammal estende Animal {
}

public class Reptile extends Animal {
}

classe pública Dog estende Mamífero {
}

```

Agora, com base no exemplo acima, em termos orientados a objeto, os seguintes são verdadeiros:

Animal é a superclasse da classe Mamífero.

Animal é a superclasse da classe Reptile.

Mamífero e réptil são subclasses da classe Animal.

Dog é a subclasse das classes Mamífero e Animal.

Agora, se considerarmos o relacionamento IS-A, podemos dizer -

Mamífero IS-A Animal

Réptil IS-A Animal



Cachorro IS-A Mammal

Portanto: Dog IS-A Animal também

Com o uso da palavra-chave `extends`, as subclasses poderão herdar todas as propriedades da superclasse, exceto as propriedades privadas da superclasse.

Podemos garantir que o Mammal é na verdade um Animal com o uso do operador da instância.

### Exemplo

```
class Animal {  
}  
  
class Mammal extends Animal {  
}  
  
class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

Isso produzirá o seguinte resultado -

### Saída

```
true  
true  
true
```

Como temos um bom entendimento da palavra-chave **`extends`**, vamos ver como a palavra-chave **`implements`** é usada para obter o relacionamento IS-A.

Geralmente, a palavra-chave **`implements`** é usada com classes para herdar as propriedades de uma interface. Interfaces nunca podem ser estendidas por uma classe.

### Exemplo

```
public interface Animal {  
}  
  
public class Mammal implements Animal {  
}  
  
public class Dog extends Mammal {  
}
```

## O exemplo da palavra-chave

Vamos usar o operador **instanceof** para verificar se o Mammal é realmente um Animal e se o cachorro é realmente um Animal.

### Exemplo

```
interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

    public static void main(String args[]) {
        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
true
true
true
```

## Relação HAS-A

Essas relações são baseadas principalmente no uso. Isso determina se uma determinada classe tem **uma** certa coisa. Esse relacionamento ajuda a reduzir a duplicação de código, bem como os erros.

Vamos olhar para um exemplo -

### Exemplo

```
public class Vehicle{}
public class Speed{}

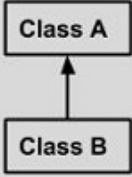
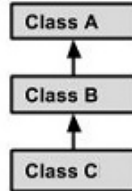
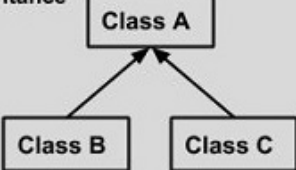
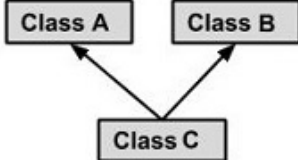
public class Van extends Vehicle {
    private Speed sp;
}
```

Isso mostra que a classe Van HAS-A Speed. Por ter uma classe separada para Speed, não precisamos colocar todo o código que pertence à velocidade dentro da classe Van, o que torna possível reutilizar a classe Speed em vários aplicativos.

No recurso Orientado a Objetos, os usuários não precisam se preocupar com qual objeto está fazendo o trabalho real. Para conseguir isso, a classe Van oculta os detalhes da implementação dos usuários da classe Van. Então, basicamente o que acontece é que os usuários pedem à classe Van para fazer uma determinada ação e a classe Van fará o trabalho sozinha ou perguntará a outra classe para executar a ação.

## Tipos de herança

Existem vários tipos de herança, conforme demonstrado abaixo.

<b>Single Inheritance</b>  <pre>graph BT; B[Class B] --&gt; A[Class A]</pre>	<pre>public class A {     ..... } public class B extends A {     ..... }</pre>
<b>Multi Level Inheritance</b>  <pre>graph BT; C[Class C] --&gt; B[Class B]; B --&gt; A[Class A]</pre>	<pre>public class A { .....} public class B extends A {.....} public class C extends B {.....}</pre>
<b>Hierarchical Inheritance</b>  <pre>graph BT; B[Class B] --&gt; A[Class A]; C[Class C] --&gt; A</pre>	<pre>public class A { .....} public class B extends A {.....} public class C extends A {.....}</pre>
<b>Multiple Inheritance</b>  <pre>graph BT; A[Class A] --&gt; C[Class C]; B[Class B] --&gt; C</pre>	<pre>public class A { .....} public class B {.....} public class C extends A,B {     ..... } // Java does not support mutiple Inheritance</pre>

Um fato muito importante a lembrar é que o Java não suporta múltiplas heranças. Isso significa que uma classe não pode estender mais de uma classe. Portanto, seguir é ilegal -

### Exemplo

```
public class extends Animal, Mammal{}
```

No entanto, uma classe pode implementar uma ou mais interfaces, o que ajudou o Java a se livrar da impossibilidade de herança múltipla.

## Java - substituindo

No capítulo anterior, falamos sobre superclasses e subclasses. Se uma classe herda um método de sua superclasse, então há uma chance de sobrescrever o método desde que ele não seja marcado como final.

O benefício da substituição é: capacidade de definir um comportamento específico para o tipo de subclasse, o que significa que uma subclasse pode implementar um método de classe pai com base em seu requisito.

Em termos orientados a objeto, substituir significa substituir a funcionalidade de um método existente.

### Exemplo

Vamos ver um exemplo.

```

class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and object
        Animal b = new Dog();       // Animal reference but Dog object

        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Animals can move
Dogs can walk and run

```

No exemplo acima, você pode ver que, embora **b** seja um tipo de `Animal`, ele executa o método `move` na classe `Dog`. A razão para isso é: No tempo de compilação, a verificação é feita no tipo de referência. No entanto, no tempo de execução, a JVM calcula o tipo de objeto e executaria o método que pertence a esse objeto específico.

Portanto, no exemplo acima, o programa irá compilar corretamente desde que a classe `Animal` tenha o método `move`. Em seguida, no tempo de execução, ele executa o método específico para esse objeto.

Considere o seguinte exemplo -

## Exemplo

```

class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
    public void bark() {
        System.out.println("Dogs can bark");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and object
    }
}

```

```
Animal b = new Dog(); // Animal reference but Dog object

a.move(); // runs the method in Animal class
b.move(); // runs the method in Dog class
b.bark();
    }
}
```

Isso produzirá o seguinte resultado -

## Saída

```
TestDog.java:26: error: cannot find symbol
    b.bark();
      ^
symbol:   method bark()
location: variable b of type Animal
1 error
```

Este programa lançará um erro de tempo de compilação, já que o tipo de referência de b não possui um método chamado bark.

## Regras para substituição do método

A lista de argumentos deve ser exatamente igual à do método substituído.

O tipo de retorno deve ser o mesmo ou um subtipo do tipo de retorno declarado no método original substituído na superclasse.

O nível de acesso não pode ser mais restritivo que o nível de acesso do método substituído. Por exemplo: Se o método da superclasse for declarado public, o método de sobreposição na subclasse não pode ser privado nem protegido.

Os métodos de instância só podem ser substituídos se forem herdados pela subclasse.

Um método declarado final não pode ser substituído.

Um método declarado estático não pode ser substituído, mas pode ser declarado novamente.

Se um método não puder ser herdado, ele não poderá ser substituído.

Uma subclasse dentro do mesmo pacote da superclasse da instância pode substituir qualquer método de superclasse que não seja declarado como privado ou final.

Uma subclasse em um pacote diferente só pode substituir os métodos não finais declarados como públicos ou protegidos.

Um método de substituição pode lançar qualquer exceção desmarcada, independentemente de o método substituído lançar exceções ou não. No entanto, o método de substituição não deve lançar exceções verificadas que são novas ou mais largas do que aquelas declaradas pelo método substituído. O método de substituição pode lançar exceções mais restritas ou menos do que o método substituído.

Construtores não podem ser substituídos.

## Usando a super palavra-chave

Ao invocar uma versão da superclasse de um método substituído, a palavra-chave **super** é usada.

### Exemplo

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        super.move(); // invokes the super class method
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {

    public static void main(String args[]) {
        Animal b = new Dog(); // Animal reference but Dog object
        b.move(); // runs the method in Dog class
    }
}
```

Isso produzirá o seguinte resultado -

### Saída

```
Animals can move
Dogs can walk and run
```

## Java - Polimorfismo

Polimorfismo é a capacidade de um objeto assumir muitas formas. O uso mais comum de polimorfismo na OOP ocorre quando uma referência de classe pai é usada para se referir a um objeto de classe filho.

Qualquer objeto Java que possa passar mais de um teste IS-A é considerado polimórfico. Em Java, todos os objetos Java são polimórficos, pois qualquer objeto passará no teste IS-A para seu próprio tipo e para a classe Object.

É importante saber que a única maneira possível de acessar um objeto é através de uma variável de referência. Uma variável de referência pode ser de apenas um tipo. Uma vez declarado, o tipo de uma variável de referência não pode ser alterado.

A variável de referência pode ser reatribuída a outros objetos, desde que não seja declarada final. O tipo da variável de referência determinaria os métodos que ela pode invocar no objeto.

Uma variável de referência pode referir-se a qualquer objeto de seu tipo declarado ou qualquer subtipo de seu tipo declarado. Uma variável de referência pode ser declarada

como uma classe ou tipo de interface.

## Exemplo

Vamos ver um exemplo.

```
public interface Vegetarian{}  
public class Animal{}  
public class Deer extends Animal implements Vegetarian{}
```

Agora, a classe Deer é considerada polimórfica, já que ela possui herança múltipla. A seguir, são verdadeiros para os exemplos acima -

Um cervo é um animal

Um cervo é um vegetariano

Um cervo é um cervo

Um objeto Deer IS-A

Quando aplicamos os fatos da variável de referência a uma referência de objeto Deer, as seguintes declarações são legais -

## Exemplo

```
Deer d = new Deer();  
Animal a = d;  
Vegetarian v = d;  
Object o = d;
```

Todas as variáveis de referência d, a, v, o referem-se ao mesmo objeto Deer no heap.

Nesta seção, mostrarei como o comportamento dos métodos substituídosMétodos Virtuais em Java permite que você aproveite o polimorfismo ao projetar suas classes.

Já discutimos a substituição de métodos, em que uma classe filha pode substituir um método em seu pai. Um método substituído é essencialmente oculto na classe pai e não é chamado a menos que a classe filha use a palavra-chave super dentro do método de substituição.

## Exemplo

```
/* File name : Employee.java */  
public class Employee {  
    private String name;  
    private String address;  
    private int number;  
  
    public Employee(String name, String address, int number) {  
        System.out.println("Constructing an Employee");  
        this.name = name;  
        this.address = address;  
        this.number = number;  
    }  
  
    public void mailCheck() {  
        System.out.println("Mailing a check to " + this.name + " " + this.address);  
    }  
}
```

```

    }

    public String toString() {
        return name + " " + address + " " + number;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String newAddress) {
        address = newAddress;
    }

    public int getNumber() {
        return number;
    }
}

```

Agora suponha que nós estendamos a classe Employee da seguinte maneira -

```

/* File name : Salary.java */
public class Salary extends Employee {
    private double salary; // Annual salary

    public Salary(String name, String address, int number, double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}

```

Agora, você estuda cuidadosamente o programa a seguir e tenta determinar sua saída -

```

/* File name : VirtualDemo.java */
public class VirtualDemo {

    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
    }
}

```



```
s.mailCheck();
System.out.println("\n Call mailCheck using Employee reference--");
e.mailCheck();
}
}
```

Isso produzirá o seguinte resultado -

## Saída

```
Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Aqui, instanciamos dois objetos Salary. Uma utilizando uma referência de salário **s** , e o outro usando uma referência empregado **e** .

Ao invocar *s.mailCheck ()* , o compilador vê mailCheck () na classe Salary no momento da compilação, e a JVM invoca mailCheck () na classe Salary em tempo de execução.

mailCheck () em **e** é bem diferente porque **e** é uma referência de Funcionário. Quando o compilador vê *e.mailCheck ()* , o compilador vê o método mailCheck () na classe Employee.

Aqui, em tempo de compilação, o compilador usou mailCheck () no Employee para validar essa declaração. No tempo de execução, no entanto, a JVM invoca mailCheck () na classe Salary.

Esse comportamento é chamado de chamada de método virtual e esses métodos são chamados de métodos virtuais. Um método substituído é invocado no tempo de execução, não importa o tipo de dados que a referência é usada no código-fonte em tempo de compilação.

## Java - abstração

Conforme o dicionário, a **abstração** é a qualidade de lidar com idéias e não com eventos. Por exemplo, quando você considera o caso de e-mail, detalhes complexos, como o que acontece assim que você envia um e-mail, o protocolo que o seu servidor de e-mail usa fica oculto ao usuário. Portanto, para enviar um e-mail, basta digitar o conteúdo, mencionar o endereço do destinatário e clicar em enviar.

Da mesma forma em programação orientada a objetos, abstração é um processo de ocultar os detalhes de implementação do usuário, apenas a funcionalidade será fornecida ao usuário. Em outras palavras, o usuário terá as informações sobre o que o objeto faz em vez de como ele faz isso.

Em Java, a abstração é obtida usando classes e interfaces abstratas.

# Classe abstrata

Uma classe que contém a palavra-chave **abstract** na sua declaração é conhecida como classe abstrata.

Classes abstratas podem ou não conter *métodos abstratos*, ou seja, métodos sem corpo (public void get ());)

Mas, se uma classe tiver pelo menos um método abstrato, a classe **deve** ser declarada abstrata.

Se uma classe é declarada abstrata, não pode ser instanciada.

Para usar uma classe abstrata, você precisa herdá-la de outra classe, fornecer implementações para os métodos abstratos nela.

Se você herdar uma classe abstrata, precisará fornecer implementações para todos os métodos abstratos nela contidos.

## Exemplo

Esta seção fornece um exemplo da classe abstrata. Para criar uma classe abstrata, apenas use a palavra-chave **abstract** antes da palavra-chave class, na declaração de classe.

```
/* File name : Employee.java */
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public double computePay() {
        System.out.println("Inside Employee computePay");
        return 0.0;
    }

    public void mailCheck() {
        System.out.println("Mailing a check to " + this.name + " " + this.address);
    }

    public String toString() {
        return name + " " + address + " " + number;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String newAddress) {
        address = newAddress;
    }
}
```

```
public int getNumber() {  
    return number;  
}  
}
```

Você pode observar que, exceto métodos abstratos, a classe Employee é igual à classe normal em Java. A classe agora é abstrata, mas ainda possui três campos, sete métodos e um construtor.

Agora você pode tentar instanciar a classe Employee da seguinte maneira -

```
/* File name : AbstractDemo.java */  
public class AbstractDemo {  
  
    public static void main(String [] args) {  
        /* Following is not allowed and would raise error */  
        Employee e = new Employee("George W.", "Houston, TX", 43);  
        System.out.println("\n Call mailCheck using Employee reference--");  
        e.mailCheck();  
    }  
}
```

Quando você compila a classe acima, ele apresenta o seguinte erro -

```
Employee.java:46: Employee is abstract; cannot be instantiated  
        Employee e = new Employee("George W.", "Houston, TX", 43);  
                        ^  
1 error
```

## Herdando a classe abstrata

Podemos herdar as propriedades da classe Employee como a classe concreta da seguinte maneira -

### Exemplo

```
/* File name : Salary.java */  
public class Salary extends Employee {  
    private double salary;    // Annual salary  
  
    public Salary(String name, String address, int number, double salary) {  
        super(name, address, number);  
        setSalary(salary);  
    }  
  
    public void mailCheck() {  
        System.out.println("Within mailCheck of Salary class ");  
        System.out.println("Mailing check to " + getName() + " with salary " + salary);  
    }  
  
    public double getSalary() {  
        return salary;  
    }  
  
    public void setSalary(double newSalary) {  
        if(newSalary >= 0.0) {  
            salary = newSalary;  
        }  
    }  
}
```

```

public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary/52;
}
}

```

Aqui, você não pode instanciar a classe Employee, mas pode instanciar a Classe Salarial e, usando essa instância, você pode acessar todos os três campos e sete métodos da classe Employee, conforme mostrado abaixo.

```

/* File name : AbstractDemo.java */
public class AbstractDemo {

    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

Isso produz o seguinte resultado -

## Saída

```

Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0

```

Se você quiser que uma classe contenha um método específico, mas Métodos abstratos deseja que a implementação real desse método seja determinada por classes filhas, você pode declarar o método na classe pai como um resumo.

palavra-chave **abstrata** é usada para declarar o método como abstrato.

Você precisa colocar a palavra-chave **abstract** antes do nome do método na declaração do método.

Um método abstrato contém uma assinatura de método, mas nenhum corpo de método.

Em vez de chaves, um método abstrato terá um ponto-e-vírgula (;) no final.

A seguir, um exemplo do método abstrato.

## Exemplo

```

public abstract class Employee {

```

```
private String name;  
private String address;  
private int number;  
  
public abstract double computePay();  
// Remainder of class definition  
}
```

Declarar um método como abstrato tem duas consequências -

A classe que o contém deve ser declarada como abstrata.

Qualquer classe que herda a classe atual deve sobrescrever o método abstrato ou declarar-se como abstrato.

**Nota** - Eventualmente, uma classe descendente tem que implementar o método abstrato; caso contrário, você teria uma hierarquia de classes abstratas que não podem ser instanciadas.

Suponha que a classe Salary herde a classe Employee, então deve implementar o método **computePay ()** como mostrado abaixo -

```
/* File name : Salary.java */  
public class Salary extends Employee {  
    private double salary;    // Annual salary  
  
    public double computePay() {  
        System.out.println("Computing salary pay for " + getName());  
        return salary/52;  
    }  
    // Remainder of class definition  
}
```

## Java - encapsulamento

**O encapsulamento** é um dos quatro conceitos fundamentais de OOP. Os outros três são herança, polimorfismo e abstração.

O encapsulamento em Java é um mecanismo de agrupamento dos dados (variáveis) e código que atua nos dados (métodos) juntos como uma única unidade. No encapsulamento, as variáveis de uma classe serão ocultadas de outras classes e podem ser acessadas apenas pelos métodos de sua classe atual. Portanto, também é conhecido como **ocultação de dados** .

Para conseguir o encapsulamento em Java -

Declare as variáveis de uma classe como privada.

Forneça métodos setter e getter públicos para modificar e visualizar os valores das variáveis.

### Exemplo

A seguir, um exemplo que demonstra como alcançar o encapsulamento em Java -

```
/* File name : EncapTest.java */  
public class EncapTest {
```

```

private String name;
private String idNum;
private int age;

public int getAge() {
    return age;
}

public String getName() {
    return name;
}

public String getIdNum() {
    return idNum;
}

public void setAge( int newAge) {
    age = newAge;
}

public void setName(String newName) {
    name = newName;
}

public void setIdNum( String newId) {
    idNum = newId;
}
}

```

Os métodos públicos setXXX () e getXXX () são os pontos de acesso das variáveis de instância da classe EncapTest. Normalmente, esses métodos são referidos como getters e setters. Portanto, qualquer classe que queira acessar as variáveis deve acessá-las por meio desses getters e setters.

As variáveis da classe EncapTest podem ser acessadas usando o seguinte programa -

```

/* File name : RunEncap.java */
public class RunEncap {

    public static void main(String args[]) {
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```
Name : James Age : 20
```

## Benefícios do Encapsulamento

Os campos de uma classe podem ser feitos somente para leitura ou somente para gravação.

Uma classe pode ter controle total sobre o que é armazenado em seus campos.

# Java - Interfaces

Uma interface é um tipo de referência em Java. É semelhante a classe. É uma coleção de métodos abstratos. Uma classe implementa uma interface, herdando assim os métodos abstratos da interface.

Juntamente com métodos abstratos, uma interface também pode conter constantes, métodos padrão, métodos estáticos e tipos aninhados. Os corpos de método existem apenas para métodos padrão e métodos estáticos.

Escrever uma interface é semelhante a escrever uma aula. Mas uma classe descreve os atributos e comportamentos de um objeto. E uma interface contém comportamentos que uma classe implementa.

A menos que a classe que implementa a interface seja abstrata, todos os métodos da interface precisam ser definidos na classe.

Uma interface é semelhante a uma classe das seguintes maneiras -

- Uma interface pode conter qualquer quantidade de métodos.

- Uma interface é gravada em um arquivo com uma extensão **.java** , com o nome da interface correspondente ao nome do arquivo.

- O código de bytes de uma interface aparece em um arquivo **.class** .

- Interfaces aparecem em pacotes e seu arquivo de bytecode correspondente deve estar em uma estrutura de diretório que corresponda ao nome do pacote.

No entanto, uma interface é diferente de uma classe de várias maneiras, incluindo -

- Você não pode instanciar uma interface.

- Uma interface não contém nenhum construtor.

- Todos os métodos em uma interface são abstratos.

- Uma interface não pode conter campos de instância. Os únicos campos que podem aparecer em uma interface devem ser declarados estáticos e finais.

- Uma interface não é estendida por uma classe; é implementado por uma classe.

- Uma interface pode estender várias interfaces.

## Declarando Interfaces

A palavra-chave da **interface** é usada para declarar uma interface. Aqui está um exemplo simples para declarar uma interface -

### Exemplo

A seguir, um exemplo de uma interface -

```
/* File name : NameOfInterface.java */  
import java.lang.*;  
// Any number of import statements
```

```
public interface NameOfInterface {  
    // Any number of final, static fields  
    // Any number of abstract method declarations\  
}
```

Interfaces tem as seguintes propriedades -

Uma interface é implicitamente abstrata. Você não precisa usar a palavra-chave **abstract** ao declarar uma interface.

Cada método em uma interface também é implicitamente abstrato, portanto, a palavra-chave abstrata não é necessária.

Métodos em uma interface são implicitamente públicos.

## Exemplo

```
/* File name : Animal.java */  
interface Animal {  
    public void eat();  
    public void travel();  
}
```

## Implementando Interfaces

Quando uma classe implementa uma interface, você pode pensar na classe como assinando um contrato, concordando em executar os comportamentos específicos da interface. Se uma classe não executa todos os comportamentos da interface, a classe deve se declarar como abstrata.

A classe usa o **implementos** palavra-chave para implementar uma interface. A palavra-chave implements aparece na declaração de classe após a parte de extensão da declaração.

## Exemplo

```
/* File name : MammalInt.java */  
public class MammalInt implements Animal {  
  
    public void eat() {  
        System.out.println("Mammal eats");  
    }  
  
    public void travel() {  
        System.out.println("Mammal travels");  
    }  
  
    public int noOfLegs() {  
        return 0;  
    }  
  
    public static void main(String args[]) {  
        MammalInt m = new MammalInt();  
        m.eat();  
        m.travel();  
    }  
}
```



Isso produzirá o seguinte resultado -

## Saída

```
Mammal eats  
Mammal travels
```

Ao substituir métodos definidos em interfaces, existem várias regras a serem seguidas -

Exceções verificadas não devem ser declaradas em métodos de implementação diferentes daqueles declarados pelo método de interface ou subclasses daqueles declarados pelo método de interface.

A assinatura do método de interface e o mesmo tipo ou subtipo de retorno devem ser mantidos ao substituir os métodos.

Uma classe de implementação em si pode ser abstrata e, em caso afirmativo, os métodos de interface não precisam ser implementados.

Quando interfaces de implementação, existem várias regras -

Uma classe pode implementar mais de uma interface por vez.

Uma classe pode estender apenas uma classe, mas implementar várias interfaces.

Uma interface pode estender outra interface, de maneira semelhante à que uma classe pode estender outra classe.

## Estendendo Interfaces

Uma interface pode estender outra interface da mesma maneira que uma classe pode estender outra classe. A palavra-chave **extends** é usada para estender uma interface e a interface filha herda os métodos da interface pai.

A seguinte interface de esportes é estendida por interfaces de hóquei e futebol.

## Exemplo

```
// Filename: Sports.java  
public interface Sports {  
    public void setHomeTeam(String name);  
    public void setVisitingTeam(String name);  
}  
  
// Filename: Football.java  
public interface Football extends Sports {  
    public void homeTeamScored(int points);  
    public void visitingTeamScored(int points);  
    public void endOfQuarter(int quarter);  
}  
  
// Filename: Hockey.java  
public interface Hockey extends Sports {  
    public void homeGoalScored();  
    public void visitingGoalScored();  
    public void endOfPeriod(int period);  
    public void overtimePeriod(int ot);  
}
```

A interface do Hockey tem quatro métodos, mas herda dois do Sports; Assim, uma classe que implementa o hóquei precisa implementar todos os seis métodos. Da mesma forma, uma classe que implementa o futebol precisa definir os três métodos do futebol e os dois métodos do esporte.

## Estendendo Múltiplas Interfaces

Uma classe Java só pode estender uma classe pai. Herança múltipla não é permitida. Interfaces não são classes, no entanto, e uma interface pode estender mais de uma interface pai.

A palavra-chave `extends` é usada uma vez e as interfaces pai são declaradas em uma lista separada por vírgulas.

Por exemplo, se a interface do Hockey estendesse esportes e eventos, seria declarada como -

### Exemplo

```
public interface Hockey extends Sports, Event
```

## Interfaces de marcação

O uso mais comum de interfaces de extensão ocorre quando a interface pai não contém nenhum método. Por exemplo, a interface `MouseListener` no pacote `java.awt.event` estendido `java.util.EventListener`, que é definido como -

### Exemplo

```
package java.util;  
public interface EventListener  
{}
```

Uma interface sem métodos é chamada de interface de **marcação**. Existem dois propósitos básicos de design de interfaces de tags -

**Cria um pai comum** - Assim como a interface `EventListener`, que é estendida por dezenas de outras interfaces na API Java, é possível usar uma interface de marcação para criar um pai comum entre um grupo de interfaces. Por exemplo, quando uma interface estende `EventListener`, a JVM sabe que essa interface específica será usada em um cenário de delegação de evento.

**Adiciona um tipo de dados a uma classe** - Essa situação é de onde vem o termo de marcação. Uma classe que implementa uma interface de marcação não precisa definir nenhum método (já que a interface não possui nenhum), mas a classe se torna um tipo de interface através do polimorfismo.

## Java - Pacotes

Os pacotes são usados em Java para evitar conflitos de nomes, controlar o acesso, facilitar a pesquisa / localização e uso de classes, interfaces, enumerações e anotações, etc.

Um **pacote** pode ser definido como um agrupamento de tipos relacionados (classes, interfaces, enumerações e anotações) fornecendo proteção de acesso e gerenciamento de namespace.

Alguns dos pacotes existentes em Java são -

**java.lang** - agrupa as classes fundamentais

**java.io** - classes para entrada, funções de saída são empacotadas neste pacote

Os programadores podem definir seus próprios pacotes para agrupar grupos de classes / interfaces, etc. É uma boa prática agrupar classes relacionadas implementadas por você para que um programador possa determinar facilmente que as classes, interfaces, enumerações e anotações estão relacionadas.

Como o pacote cria um novo namespace, não haverá conflitos de nome com nomes em outros pacotes. Usando pacotes, é mais fácil fornecer controle de acesso e também é mais fácil localizar as classes relacionadas.

## Criando um pacote

Ao criar um pacote, você deve escolher um nome para o pacote e incluir uma declaração de **pacote** junto com esse nome na parte superior de todo arquivo de origem que contenha as classes, interfaces, enumerações e tipos de anotação que deseja incluir no pacote.

A declaração do pacote deve ser a primeira linha no arquivo de origem. Pode haver apenas uma declaração de pacote em cada arquivo de origem e se aplica a todos os tipos no arquivo.

Se uma declaração de pacote não for usada, as classes, interfaces, enumerações e tipos de anotação serão colocados no pacote padrão atual.

Para compilar os programas Java com instruções package, você tem que usar a opção -d como mostrado abaixo.

```
javac -d Destination_folder file_name.java
```

Em seguida, uma pasta com o nome do pacote fornecido é criada no destino especificado e os arquivos de classe compilados serão colocados nessa pasta.

## Exemplo

Vejamos um exemplo que cria um pacote chamado **animais** . É uma boa prática usar nomes de pacotes com letras minúsculas para evitar conflitos com os nomes de classes e interfaces.

O exemplo de pacote a seguir contém a interface chamada *animals* -

```
/* File name : Animal.java */  
package animais;  
  
interface Animal {  
    public void eat();  
    public void travel();  
}
```

```
}
```

Agora, vamos implementar a interface acima no mesmo pacote de *animais* -

```
package animals;
/* File name : MammalInt.java */

public class MammalInt implements Animal {

    public void eat() {
        System.out.println("Mammal eats");
    }

    public void travel() {
        System.out.println("Mammal travels");
    }

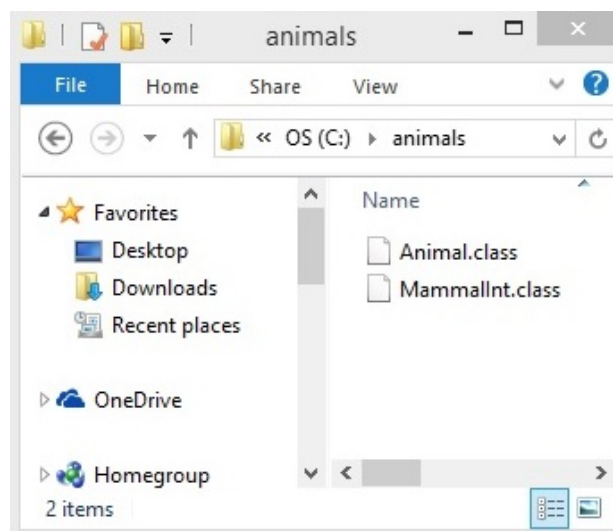
    public int noOfLegs() {
        return 0;
    }

    public static void main(String args[]) {
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

Agora compile os arquivos java como mostrado abaixo -

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Agora um pacote / pasta com o nome **animals** será criado no diretório atual e estes arquivos de classe serão colocados como mostrado abaixo.



Você pode executar o arquivo de classe dentro do pacote e obter o resultado como mostrado abaixo.

```
Mammal eats
Mammal travels
```

## A palavra-chave de importação

Se uma classe quiser usar outra classe no mesmo pacote, o nome do pacote não precisará ser usado. Classes no mesmo pacote se encontram sem nenhuma sintaxe especial.

## Exemplo

Aqui, uma classe chamada Boss é adicionada ao pacote de folha de pagamento que já contém Employee. O Boss pode então se referir à classe Employee sem usar o prefixo da folha de pagamento, conforme demonstrado pela seguinte classe Boss.

```
package payroll;
public class Boss {
    public void payEmployee(Employee e) {
        e.mailCheck();
    }
}
```

O que acontece se a classe Employee não estiver no pacote da folha de pagamento? A classe Boss deve então usar uma das seguintes técnicas para se referir a uma classe em um pacote diferente.

O nome totalmente qualificado da classe pode ser usado. Por exemplo -

```
payroll.Employee
```

O pacote pode ser importado usando a palavra-chave import e o curinga (\*). Por exemplo -

```
import payroll.*;
```

A classe em si pode ser importada usando a palavra-chave import. Por exemplo -

```
import payroll.Employee;
```

**Nota** - Um arquivo de classe pode conter qualquer número de instruções de importação. As instruções de importação devem aparecer após a declaração do pacote e antes da declaração da classe.

## A estrutura de diretórios dos pacotes

Dois resultados principais ocorrem quando uma classe é colocada em um pacote -

O nome do pacote se torna parte do nome da classe, como acabamos de discutir na seção anterior.

O nome do pacote deve corresponder à estrutura de diretórios onde o bytecode correspondente reside.

Aqui está uma maneira simples de gerenciar seus arquivos em Java -

Coloque o código-fonte para uma classe, interface, enumeração ou tipo de anotação em um arquivo de texto cujo nome é o nome simples do tipo e cuja extensão é **.java** .

Por exemplo -

```
// File Name : Car.java
package vehicle;

public class Car {
    // Class implementation.
}
```

Agora, coloque o arquivo de origem em um diretório cujo nome reflita o nome do pacote ao qual a classe pertence -

```
....\vehicle\Car.java
```

Agora, o nome da classe e o nome de caminho qualificados seriam os seguintes -

Nome da classe → vehicle.Car

Nome do caminho → vehicle \ Car.java (no windows)

Em geral, uma empresa usa seu nome de domínio da Internet invertida para seus nomes de pacotes.

**Exemplo** - O nome de domínio da Internet de uma empresa é apple.com e, em seguida, todos os nomes dos pacotes começam com com.apple. Cada componente do nome do pacote corresponde a um subdiretório.

**Exemplo** - A empresa tinha um pacote com.apple.computers que continha um arquivo de origem Dell.java, ele estaria contido em uma série de subdiretórios como esse -

```
....\com\apple\computers\Dell.java
```

No momento da compilação, o compilador cria um arquivo de saída diferente para cada classe, interface e enumeração definidas nele. O nome base do arquivo de saída é o nome do tipo e sua extensão é **.class** .

Por exemplo -

```
// File Name: Dell.java
package com.apple.computers;

public class Dell {
}

class Ups {
}
```

Agora, compile este arquivo da seguinte maneira usando a opção -d -

```
$javac -d . Dell.java
```

Os arquivos serão compilados da seguinte maneira -

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

Você pode importar todas as classes ou interfaces definidas em `\ com \ apple \ computers` \ como segue -

```
import com.apple.computers.*;
```

Como os arquivos de origem .java, os arquivos .class compilados devem estar em uma série de diretórios que refletem o nome do pacote. No entanto, o caminho para os arquivos .class não precisa ser o mesmo que o caminho para os arquivos de origem .java. Você pode organizar seus diretórios de origem e de classe separadamente, como -

```
<path-one>\sources\com\apple\computers\Dell.java  
  
<path-two>\classes\com\apple\computers\Dell.class
```

Ao fazer isso, é possível dar acesso ao diretório de classes para outros programadores sem revelar suas fontes. Você também precisa gerenciar os arquivos de origem e de classe dessa maneira para que o compilador e a Java Virtual Machine (JVM) possam encontrar todos os tipos usados pelo programa.

O caminho completo para o diretório de classes, <path-two> \ classes, é chamado de caminho de classe e é configurado com a variável de sistema CLASSPATH. Tanto o compilador quanto a JVM constroem o caminho para seus arquivos .class incluindo o nome do pacote no caminho da classe.

Dizer <caminho-dois> \ classes é o caminho da classe e o nome do pacote é com.apple.computers, então o compilador e a JVM procurarão arquivos .class em <caminho-dois> \ classes \ com \ apple \ computers.

Um caminho de classe pode incluir vários caminhos. Vários caminhos devem ser separados por um ponto-e-vírgula (Windows) ou dois-pontos (Unix). Por padrão, o compilador e a JVM pesquisam o diretório atual e o arquivo JAR contendo as classes da plataforma Java para que esses diretórios sejam automaticamente no caminho da classe.

## Definir variável de sistema CLASSPATH

Para exibir a variável CLASSPATH atual, use os seguintes comandos no Windows e no UNIX (Bourne shell) -

No Windows → C: \> defina CLASSPATH

No UNIX →% echo \$ CLASSPATH

Para excluir o conteúdo atual da variável CLASSPATH, use -

No Windows → C: \> defina CLASSPATH =

No UNIX →% não configurado CLASSPATH; exportar CLASSPATH

Para definir a variável CLASSPATH -

No Windows → defina CLASSPATH = C: \ usuários \ jack \ java \ classes

No UNIX →% CLASSPATH = / home / jack / java / classes; exportar CLASSPATH

## Java - Estruturas de Dados

As estruturas de dados fornecidas pelo pacote do utilitário Java são muito poderosas e executam uma ampla variedade de funções. Estas estruturas de dados consistem na

seguinte interface e classes -

Enumeração

BitSet

Vetor

Pilha

Dicionário

Hashtable

Propriedades

Todas essas classes agora são legadas e o Java-2 introduziu uma nova estrutura chamada Collections Framework, que será discutida no próximo capítulo. -

## A enumeração

A interface de enumeração não é em si uma estrutura de dados, mas é muito importante dentro do contexto de outras estruturas de dados. A interface de enumeração define um meio para recuperar elementos sucessivos de uma estrutura de dados.

Por exemplo, Enumeração define um método chamado `nextElement` que é usado para obter o próximo elemento em uma estrutura de dados que contém vários elementos.

Para ter mais detalhes sobre essa interface, marque [The Enumeration](#) .

## O BitSet

A classe `BitSet` implementa um grupo de bits ou sinalizadores que podem ser definidos e apagados individualmente.

Essa classe é muito útil nos casos em que você precisa manter um conjunto de valores booleanos; você apenas atribui um bit a cada valor e define ou desmarca conforme apropriado.

Para mais detalhes sobre esta classe, verifique o [BitSet](#) .

## O vetor

A classe `Vector` é semelhante a um array Java tradicional, exceto pelo fato de poder crescer conforme necessário para acomodar novos elementos.

Como uma matriz, os elementos de um objeto `Vector` podem ser acessados por meio de um índice no vetor.

O bom de usar a classe `Vector` é que você não precisa se preocupar em configurá-la para um tamanho específico na criação; ela encolhe e cresce automaticamente quando necessário.

Para mais detalhes sobre essa classe, marque [The Vector](#) .

A classe `Stack` implementa uma pilha de elementos last-in-first-out (LIFO).

A pilha



Você pode pensar em uma pilha literalmente como uma pilha vertical de objetos; quando você adiciona um novo elemento, ele fica empilhado sobre os outros.

Quando você puxa um elemento da pilha, ele sai do topo. Em outras palavras, o último elemento adicionado à pilha é o primeiro a ser desativado.

Para mais detalhes sobre essa classe, marque a pilha [aqui](#).

## O dicionário

A classe Dictionary é uma classe abstrata que define uma estrutura de dados para mapeamento de chaves para valores.

Isso é útil nos casos em que você deseja acessar dados por meio de uma chave específica, em vez de um índice inteiro.

Como a classe Dictionary é abstrata, ela fornece apenas a estrutura para uma estrutura de dados mapeada por chave, em vez de uma implementação específica.

Para mais detalhes sobre essa classe, verifique o dicionário [aqui](#).

## A Hashtable

A classe Hashtable fornece um meio de organizar dados com base em alguma estrutura de chave definida pelo usuário.

Por exemplo, em uma tabela de hash da lista de endereços, você pode armazenar e classificar dados com base em uma chave, como CEP, em vez de no nome de uma pessoa.

O significado específico de chaves em relação a tabelas hash depende totalmente do uso da tabela de hash e dos dados nela contidos.

Para mais detalhes sobre essa classe, marque A Hashtable [aqui](#).

## As propriedades

Propriedades é uma subclasse de Hashtable. É usado para manter listas de valores nos quais a chave é uma String e o valor também é uma String.

A classe Properties é usada por muitas outras classes Java. Por exemplo, é o tipo de objeto retornado por System.getProperties () ao obter valores ambientais.

Para mais detalhes sobre essa classe, marque as propriedades [aqui](#).

# Java - estrutura de coleções

Antes do Java 2, o Java fornecia classes ad hoc, como **Dictionary**, **Vector**, **Stack** e **Properties**, para armazenar e manipular grupos de objetos. Embora essas classes fossem bastante úteis, elas não tinham um tema central e unificador. Assim, a maneira como você usou o Vector foi diferente da maneira como você usou o recurso Propriedades.

A estrutura de coleções foi projetada para atender vários objetivos, como -

O framework tinha que ser de alto desempenho. As implementações para as

coleções fundamentais (arrays dinâmicos, listas vinculadas, árvores e hashtables) seriam altamente eficientes.

O framework tinha que permitir que diferentes tipos de coleções funcionassem de maneira similar e com alto grau de interoperabilidade.

O framework teve que ampliar e / ou adaptar uma coleção facilmente.

Para este fim, toda a estrutura de coleções é projetada em torno de um conjunto de interfaces padrão. Várias implementações padrão, como **LinkedList**, **HashSet** e **TreeSet**, dessas interfaces são fornecidas para você usar como estão e você também pode implementar sua própria coleção, se desejar.

Uma estrutura de coleções é uma arquitetura unificada para representar e manipular coleções. Todas as estruturas de coleções contêm o seguinte -

**Interfaces** - Estes são tipos de dados abstratos que representam coleções. Interfaces permitem que as coleções sejam manipuladas independentemente dos detalhes de sua representação. Em linguagens orientadas a objeto, as interfaces geralmente formam uma hierarquia.

**Implementações, isto é, Classes** - Estas são as implementações concretas das interfaces de coleção. Em essência, eles são estruturas de dados reutilizáveis.

**Algoritmos** - Estes são os métodos que realizam cálculos úteis, como pesquisa e classificação, em objetos que implementam interfaces de coleta. Os algoritmos são considerados polimórficos: isto é, o mesmo método pode ser usado em muitas implementações diferentes da interface de coleta apropriada.

Além das coleções, o framework define várias interfaces e classes de mapas. Mapas armazenam pares de chave / valor. Embora os mapas não sejam *coleções* no uso adequado do termo, eles são totalmente integrados às coleções.

## As Interfaces de Coleta

A estrutura de coleções define várias interfaces. Esta seção fornece uma visão geral de cada interface -

Sr. Não.	Interface e Descrição
1	<b>A interface de coleção</b> Isso permite que você trabalhe com grupos de objetos; está no topo da hierarquia de coleções.
2	<b>A interface da lista</b> Isso amplia a <b>coleção</b> e uma instância da lista armazena uma coleção ordenada de elementos.
3	<b>O conjunto</b>

	Isso amplia a coleção para manipular conjuntos, que devem conter elementos exclusivos.
4	<b>O SortedSet</b> Isso estende o conjunto para manipular conjuntos classificados.
5	<b>O mapa</b> Isso mapeia chaves exclusivas para valores.
6	<b>O Map.Entry</b> Isso descreve um elemento (um par chave / valor) em um mapa. Esta é uma classe interna do mapa.
7	<b>O SortedMap</b> Isso estende o mapa para que as chaves sejam mantidas em uma ordem crescente.
8	<b>A enumeração</b> Esta é uma interface legada que define os métodos pelos quais você pode enumerar (obter um de cada vez) os elementos em uma coleção de objetos. Essa interface herdada foi substituída pelo Iterator.

## As aulas de coleção

Java fornece um conjunto de classes de coleção padrão que implementam interfaces Collection. Algumas das classes fornecem implementações completas que podem ser usadas como estão e outras são da classe abstrata, fornecendo implementações esqueléticas que são usadas como pontos de partida para a criação de coleções concretas.

As classes de coleção padrão são resumidas na tabela a seguir -

Sr. Não.	Classe e Descrição
1	<b>AbstractColeção</b> Implementa a maior parte da interface da coleção.
2	<b>AbstractList</b> Estende AbstractCollection e implementa a maior parte da interface List.
3	<b>AbstractSequentialList</b> Estende o AbstractList para uso por uma coleção que usa acesso sequencial em vez de aleatório de seus elementos.

4	<b>LinkedList</b> Implementa uma lista vinculada estendendo <code>AbstractSequentialList</code> .
5	<b>ArrayList</b> Implementa um array dinâmico estendendo <code>AbstractList</code> .
6	<b>AbstractSet</b> Estende <code>AbstractCollection</code> e implementa a maior parte da interface <code>Set</code> .
7	<b>HashSet</b> Estende o <code>AbstractSet</code> para uso com uma tabela de hash.
8	<b>LinkedHashSet</b> Estende o <code>HashSet</code> para permitir iterações de ordem de inserção.
9	<b>TreeSet</b> Implementa um conjunto armazenado em uma árvore. Estende o <code>AbstractSet</code> .
10	<b>AbstractMap</b> Implementa a maior parte da interface do Mapa.
11	<b>HashMap</b> Estende o <code>AbstractMap</code> para usar uma tabela de hash.
12	<b>TreeMap</b> Estende o <code>AbstractMap</code> para usar uma árvore.
13	<b>WeakHashMap</b> Estende o <code>AbstractMap</code> para usar uma tabela de hash com chaves fracas.
14	<b>LinkedHashMap</b> Estende o <code>HashMap</code> para permitir iterações de ordem de inserção.
15	<b>IdentityHashMap</b> Estende o <code>AbstractMap</code> e usa a igualdade de referência ao comparar documentos.

As classes *AbstractCollection*, *AbstractSet*, *AbstractList*, *AbstractSequentialList* e *AbstractMap* fornecem implementações esqueléticas das interfaces de coleta principais, para minimizar o esforço necessário para implementá-las.

As seguintes classes legadas definidas por `java.util` foram discutidas no capítulo anterior -

Sr.	Classe e Descrição
-----	--------------------

Não.	
1	<b>Vetor</b> Isso implementa um array dinâmico. É semelhante ao ArrayList, mas com algumas diferenças.
2	<b>Pilha</b> Stack é uma subclasse de Vector que implementa uma pilha padrão de entrada e saída.
3	<b>Dicionário</b> Dictionary é uma classe abstrata que representa um repositório de armazenamento de chave / valor e opera de maneira muito semelhante ao Map.
4	<b>Hashtable</b> Hashtable era parte do java.util original e é uma implementação concreta de um Dicionário.
5	<b>Propriedades</b> Propriedades é uma subclasse de Hashtable. É usado para manter listas de valores nos quais a chave é uma String e o valor também é uma String.
6	<b>BitSet</b> Uma classe BitSet cria um tipo especial de matriz que contém valores de bits. Essa matriz pode aumentar em tamanho conforme necessário.

## Os algoritmos de coleção

A estrutura de coleções define vários algoritmos que podem ser aplicados a coleções e mapas. Esses algoritmos são definidos como métodos estáticos dentro da classe Collections.

Vários dos métodos podem lançar um **ClassCastException**, que ocorre quando é feita uma tentativa de comparar tipos incompatíveis, ou um **UnsupportedOperationException**, que ocorre quando é feita uma tentativa de modificar uma coleção não modificável.

As coleções definem três variáveis estáticas: EMPTY\_SET, EMPTY\_LIST e EMPTY\_MAP. Todos são imutáveis.

Sr. Não.	Algoritmo e Descrição
1	<b>Os algoritmos de coleção</b> Aqui está uma lista de toda a implementação do algoritmo.

## Como usar um iterador?

Muitas vezes, você vai querer percorrer os elementos de uma coleção. Por exemplo, você pode querer exibir cada elemento.

A maneira mais fácil de fazer isso é empregar um iterador, que é um objeto que implementa o `Iterator` ou a interface `ListIterator`.

O iterador permite percorrer uma coleção, obter ou remover elementos. `ListIterator` estende o `Iterator` para permitir a passagem bidirecional de uma lista e a modificação de elementos.

Sr. Não.	Método do Iterador e Descrição
1	<b>Usando o Iterador Java</b> Aqui está uma lista de todos os métodos com exemplos fornecidos pelas interfaces <code>Iterator</code> e <code>ListIterator</code> .

## Como usar um comparador?

Ambos `TreeSet` e `TreeMap` armazenam elementos em uma ordem classificada. No entanto, é o comparador que define precisamente o que significa *ordem ordenada*.

Essa interface nos permite classificar uma determinada coleção de várias maneiras diferentes. Além disso, essa interface pode ser usada para classificar qualquer instância de qualquer classe (até mesmo classes que não podemos modificar).

Sr. Não.	Método do Iterador e Descrição
1	<b>Usando o Java Comparator</b> Aqui está uma lista de todos os métodos com exemplos fornecidos pela Interface do Comparador.

A estrutura de coleções Java fornece ao programador acesso a estruturas de dados pré-empacotadas, bem como a algoritmos para manipulá-las.

Uma coleção é um objeto que pode conter referências a outros objetos. As interfaces de coleta declaram as operações que podem ser executadas em cada tipo de coleta.

As classes e interfaces da estrutura de coleções estão no pacote `java.util`.

## Java - genéricos

Seria bom se pudéssemos escrever um único método de classificação que pudesse ordenar os elementos em um array `Integer`, um array `String` ou um array de qualquer tipo que suportasse ordenação.

Java **genéricos** métodos e classes genéricas permitem aos programadores especificar,

com uma declaração de método único, um conjunto de métodos relacionados, ou com uma declaração de classe única, um conjunto de tipos relacionados, respectivamente.

Os genéricos também fornecem segurança de tipo em tempo de compilação que permite aos programadores capturar tipos inválidos em tempo de compilação.

Usando o conceito Java Generic, podemos escrever um método genérico para ordenar uma matriz de objetos e, em seguida, invocar o método genérico com matrizes Integer, matrizes duplas, matrizes String e assim por diante, para classificar os elementos da matriz.

## Métodos genéricos

Você pode escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes. Com base nos tipos de argumentos passados para o método genérico, o compilador manipula cada chamada de método apropriadamente. A seguir estão as regras para definir métodos genéricos -

Todas as declarações de métodos genéricos têm uma seção de parâmetro de tipo delimitada por colchetes angulares (<e>) que precede o tipo de retorno do método (<E> no próximo exemplo).

Cada seção de parâmetro de tipo contém um ou mais parâmetros de tipo separados por vírgulas. Um parâmetro de tipo, também conhecido como uma variável de tipo, é um identificador que especifica um nome de tipo genérico.

Os parâmetros de tipo podem ser usados para declarar o tipo de retorno e atuar como espaços reservados para os tipos de argumentos passados para o método genérico, que são conhecidos como argumentos de tipo reais.

O corpo de um método genérico é declarado como o de qualquer outro método. Observe que os parâmetros de tipo podem representar apenas tipos de referência, não tipos primitivos (como int, double e char).

## Exemplo

O exemplo a seguir ilustra como podemos imprimir uma matriz de tipos diferentes usando um único método genérico -

```
public class GenericMethodTest {
    // generic method printArray
    public static < E > void printArray( E[] inputArray ) {
        // Display array elements
        for(E element : inputArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

    public static void main(String args[]) {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
    }
}
```

```

System.out.println("Array integerArray contains:");
printArray(intArray);    // pass an Integer array

System.out.println("\nArray doubleArray contains:");
printArray(doubleArray); // pass a Double array

System.out.println("\nArray characterArray contains:");
printArray(charArray);   // pass a Character array
}
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O

```

## Parâmetros de Tipo Limitado

Pode haver momentos em que você desejará restringir os tipos de tipos que podem ser passados para um parâmetro de tipo. Por exemplo, um método que opera em números pode querer apenas aceitar instâncias de `Number` ou suas subclasses. Isto é o que os parâmetros de tipo limitados são para.

Para declarar um parâmetro de tipo limitado, liste o nome do parâmetro de tipo, seguido pela palavra-chave `extends`, seguida por seu limite superior.

## Exemplo

O exemplo a seguir ilustra como as extensões são usadas em um sentido geral para significar "estende" (como nas classes) ou "implementa" (como nas interfaces). Este exemplo é o método genérico para retornar o maior de três objetos comparáveis -

```

public class MaximumTest {
    // determines the Largest of three Comparable objects

    public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
        T max = x;    // assume x is initially the largest

        if(y.compareTo(max) > 0) {
            max = y;    // y is the largest so far
        }

        if(z.compareTo(max) > 0) {
            max = z;    // z is the largest now
        }

        return max;    // returns the largest object
    }

    public static void main(String args[]) {
        System.out.printf("Max of %d, %d and %d is %d\n\n",

```



```

        3, 4, 5, maximum( 3, 4, 5 ));

System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
    6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

System.out.printf("Max of %s, %s and %s is %s\n", "pear",
    "apple", "orange", maximum("pear", "apple", "orange"));
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear

```

Uma declaração de classe genérica parece uma declaração de classeClasses Genéricas não genérica, exceto que o nome da classe é seguido por uma seção de parâmetro de tipo.

Como nos métodos genéricos, a seção de parâmetro de tipo de uma classe genérica pode ter um ou mais parâmetros de tipo separados por vírgulas. Essas classes são conhecidas como classes parametrizadas ou tipos parametrizados porque aceitam um ou mais parâmetros.

## Exemplo

O exemplo a seguir ilustra como podemos definir uma classe genérica -

```

public class Box<T> {
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        Box<String> stringBox = new Box<String>();

        integerBox.add(new Integer(10));
        stringBox.add(new String("Hello World"));

        System.out.printf("Integer Value :%d\n\n", integerBox.get());
        System.out.printf("String Value :%s\n", stringBox.get());
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```
Integer Value :10
String Value :Hello World
```

## Java - serialização

Java fornece um mecanismo, chamado de serialização de objeto, em que um objeto pode ser representado como uma sequência de bytes que inclui os dados do objeto, bem como informações sobre o tipo do objeto e os tipos de dados armazenados no objeto.

Depois que um objeto serializado tiver sido gravado em um arquivo, ele poderá ser lido a partir do arquivo e desserializado, ou seja, as informações de tipo e os bytes que representam o objeto e seus dados poderão ser usados para recriar o objeto na memória.

O mais impressionante é que todo o processo é independente de JVM, o que significa que um objeto pode ser serializado em uma plataforma e desserializado em uma plataforma totalmente diferente.

Classes **ObjectInputStream** e **ObjectOutputStream** são fluxos de alto nível que contêm os métodos para serializar e desserializar um objeto.

A classe **ObjectOutputStream** contém muitos métodos de gravação para escrever vários tipos de dados, mas um método em particular se destaca -

```
public final void writeObject(Object x) throws IOException
```

O método acima serializa um objeto e o envia para o fluxo de saída. Da mesma forma, a classe **ObjectInputStream** contém o seguinte método para desserializar um objeto -

```
public final Object readObject() throws IOException, ClassNotFoundException
```

Esse método recupera o próximo objeto fora do fluxo e desserializa-lo. O valor de retorno é **Object**, portanto, será necessário convertê-lo em seu tipo de dados apropriado.

Para demonstrar como a serialização funciona em Java, vou usar a classe **Employee** que discutimos anteriormente no livro. Suponha que tenhamos a seguinte classe **Employee**, que implementa a interface **Serializable** -

### Exemplo

```
public class Employee implements java.io.Serializable {
    public String name;
    public String address;
    public transient int SSN;
    public int number;

    public void mailCheck() {
        System.out.println("Mailing a check to " + name + " " + address);
    }
}
```

Observe que, para uma classe ser serializada com sucesso, duas condições devem ser atendidas -

A classe deve implementar a interface **java.io.Serializable**.

Todos os campos da classe devem ser serializáveis. Se um campo não for

serializável, ele deve ser marcado como **transitório** .

Se você está curioso para saber se uma classe padrão de Java é serializável ou não, verifique a documentação para a classe. O teste é simples: se a classe implementa `java.io.Serializable`, ela é serializável; caso contrário, não é.

## Serializando um Objeto

A classe `ObjectOutputStream` é usada para serializar um objeto. O programa `SerializeDemo` a seguir instancia um objeto `Employee` e o serializa em um arquivo.

Quando o programa é executado, um arquivo chamado `employee.ser` é criado. O programa não gera nenhuma saída, mas estuda o código e tenta determinar o que o programa está fazendo.

**Nota** - Ao serializar um objeto para um arquivo, a convenção padrão em Java é fornecer ao arquivo uma extensão **.ser** .

### Exemplo

```
import java.io.*;
public class SerializeDemo {

    public static void main(String [] args) {
        Employee e = new Employee();
        e.name = "Reyan Ali";
        e.address = "Phokka Kuan, Ambehta Peer";
        e.SSN = 11122333;
        e.number = 101;

        try {
            FileOutputStream fileOut =
                new FileOutputStream("/tmp/employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
            System.out.printf("Serialized data is saved in /tmp/employee.ser");
        } catch (IOException i) {
            i.printStackTrace();
        }
    }
}
```

## Desserializando um Objeto

O seguinte programa `DeserializeDemo` desserializa o objeto `Employee` criado no programa `SerializeDemo`. Estude o programa e tente determinar sua saída -

### Exemplo

```
import java.io.*;
public class DeserializeDemo {

    public static void main(String [] args) {
        Employee e = null;
        try {
            FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
```

```

        ObjectInputStream in = new ObjectInputStream(fileIn);
        e = (Employee) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Employee class not found");
        c.printStackTrace();
        return;
    }

    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
}
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101

```

Aqui estão os pontos importantes a serem observados -

O bloco try / catch tenta capturar uma `ClassNotFoundException`, que é declarada pelo método `readObject ()`. Para que uma JVM possa desserializar um objeto, ele deve ser capaz de localizar o bytecode para a classe. Se a JVM não puder encontrar uma classe durante a desserialização de um objeto, ela lançará uma `ClassNotFoundException`.

Observe que o valor de retorno de `readObject ()` é convertido em uma referência de funcionário.

O valor do campo SSN era 11122333 quando o objeto era serializado, mas como o campo é temporário, esse valor não foi enviado para o fluxo de saída. O campo SSN do objeto Employer desserializado é 0.

## Java - Rede

O termo *programação de rede* refere-se a escrever programas que são executados em vários dispositivos (computadores), nos quais os dispositivos estão todos conectados entre si usando uma rede.

O pacote `java.net` das APIs do J2SE contém uma coleção de classes e interfaces que fornecem detalhes de comunicação de baixo nível, permitindo que você escreva programas que se concentram na solução do problema em questão.

O pacote `java.net` fornece suporte para os dois protocolos de rede comuns -

**TCP** - TCP significa Transmission Control Protocol, que permite a comunicação confiável entre dois aplicativos. O TCP é normalmente usado no Protocolo da Internet, conhecido como TCP / IP.

**UDP** - UDP significa User Datagram Protocol, um protocolo sem conexão que permite que pacotes de dados sejam transmitidos entre aplicativos.

Este capítulo dá uma boa compreensão sobre os dois assuntos a seguir -

**Socket Programming** - Este é o conceito mais utilizado em Networking e foi explicado em detalhes.

**Processamento de URL** - Isso seria coberto separadamente. Clique aqui para saber mais sobre Processamento de URL na linguagem Java.

## Programação de Soquete

Soquetes fornecem o mecanismo de comunicação entre dois computadores usando o TCP. Um programa cliente cria um soquete no final da comunicação e tenta conectar esse soquete a um servidor.

Quando a conexão é feita, o servidor cria um objeto de soquete no final da comunicação. O cliente e o servidor agora podem se comunicar gravando e lendo do soquete.

A classe `java.net.Socket` representa um soquete e a classe `java.net.ServerSocket` fornece um mecanismo para o programa do servidor escutar os clientes e estabelecer conexões com eles.

As etapas a seguir ocorrem ao estabelecer uma conexão TCP entre dois computadores usando soquetes -

O servidor instancia um objeto `ServerSocket`, indicando em qual comunicação de número de porta deve ocorrer.

O servidor chama o método `accept ()` da classe `ServerSocket`. Esse método aguarda até que um cliente se conecte ao servidor na porta especificada.

Depois que o servidor está aguardando, um cliente instancia um objeto `Socket`, especificando o nome do servidor e o número da porta a ser conectada.

O construtor da classe `Socket` tenta conectar o cliente ao servidor especificado e ao número da porta. Se a comunicação for estabelecida, o cliente agora tem um objeto `Socket` capaz de se comunicar com o servidor.

No lado do servidor, o método `accept ()` retorna uma referência a um novo soquete no servidor que está conectado ao soquete do cliente.

Depois que as conexões são estabelecidas, a comunicação pode ocorrer usando fluxos de E / S. Cada soquete possui um `OutputStream` e um `InputStream`. O `OutputStream` do cliente é conectado ao `InputStream` do servidor e o `InputStream` do cliente é conectado ao `OutputStream` do servidor.

O TCP é um protocolo de comunicação bidirecional, portanto, os dados podem ser enviados

em ambos os fluxos ao mesmo tempo. A seguir estão as classes úteis que fornecem um conjunto completo de métodos para implementar soquetes.

## Métodos de classe ServerSocket

A classe **java.net.ServerSocket** é usada por aplicativos de servidor para obter uma porta e escutar solicitações de clientes.

A classe ServerSocket tem quatro construtores -

Sr. Não.	Método e Descrição
1	<b>ServerSocket público (porta int) lança IOException</b> Tenta criar um soquete de servidor vinculado à porta especificada. Uma exceção ocorre se a porta já estiver vinculada por outro aplicativo.
2	<b>ServerSocket público (int port, int backlog) lança IOException</b> Semelhante ao construtor anterior, o parâmetro de lista de pendências especifica quantos clientes de entrada devem ser armazenados em uma fila de espera.
3	<b>ServerSocket público (porta int, backlog int, endereço InetAddress) lança IOException</b> Semelhante ao construtor anterior, o parâmetro InetAddress especifica o endereço IP local ao qual se ligar. O InetAddress é usado para servidores que podem ter vários endereços IP, permitindo que o servidor especifique quais dos seus endereços IP aceitarão solicitações de clientes.
4	<b>ServerSocket público () lança IOException</b> Cria um soquete do servidor não acoplado. Ao usar esse construtor, use o método bind () quando estiver pronto para ligar o soquete do servidor.

Se o construtor ServerSocket não lançar uma exceção, isso significa que seu aplicativo foi vinculado com êxito à porta especificada e está pronto para solicitações do cliente.

A seguir estão alguns dos métodos comuns da classe ServerSocket -

Sr. Não.	Método e Descrição
1	<b>public int getLocalPort ()</b> Retorna a porta na qual o soquete do servidor está escutando. Este método é útil se você passou em 0 como o número da porta em um construtor e deixou o servidor encontrar uma porta para você.

Quando o `ServerSocket` invoca `accept ()`, o método não retorna até que um cliente se conecte. Depois que um cliente se conecta, o `ServerSocket` cria um novo `Socket` em uma porta não especificada e retorna uma referência a esse novo `Socket`. Agora existe uma conexão TCP entre o cliente e o servidor, e a comunicação pode começar.

## Métodos de Classe de Soquete

A classe **`java.net.Socket`** representa o soquete que o cliente e o servidor usam para se comunicar uns com os outros. O cliente obtém um objeto `Socket`, instanciando um, enquanto o servidor obtém um objeto `Socket` do valor de retorno do método `accept ()`.

A classe `Socket` tem cinco construtores que um cliente usa para se conectar a um servidor -

Sr. Não.	Método e Descrição
1	<b><code>Socket público (host String, porta int) lança UnknownHostException, IOException.</code></b> Este método tenta se conectar ao servidor especificado na porta especificada. Se esse construtor não lançar uma exceção, a conexão será bem-sucedida e o cliente será conectado ao servidor.
2	<b><code>Socket público (host InetAddress, porta int) lança IOException</code></b> Esse método é idêntico ao construtor anterior, exceto que o host é denotado por um objeto <code>InetAddress</code> .
3	<b><code>Socket público (host String, porta int, InetAddress localAddress, int localPort) lança IOException.</code></b> Conecta-se ao host e porta especificados, criando um soquete no host local no endereço e porta especificados.
4	<b><code>Socket público (host InetAddress, int porta, InetAddress localAddress, int localPort) lança IOException.</code></b> Esse método é idêntico ao construtor anterior, exceto que o host é denotado por um objeto <code>InetAddress</code> em vez de uma <code>String</code> .
5	<b><code>soquete público ()</code></b> Cria um soquete desconectado. Use o método <code>connect ()</code> para conectar esse soquete a um servidor.

Quando o construtor `Socket` retorna, ele não simplesmente instancia um objeto `Socket`, mas na verdade tenta se conectar ao servidor e à porta especificados.

Alguns métodos de interesse na classe Socket estão listados aqui. Observe que o cliente e o servidor têm um objeto Socket, portanto, esses métodos podem ser chamados pelo cliente e pelo servidor.

Sr. Não.	Método e Descrição
1	<b>public void connect (host SocketAddress, int timeout) lança IOException</b> Este método conecta o soquete ao host especificado. Esse método é necessário somente quando você instancia o soquete usando o construtor sem argumento.
2	<b>public InetAddress getInetAddress ()</b> Este método retorna o endereço do outro computador ao qual este soquete está conectado.
3	<b>public int getPort ()</b> Retorna a porta à qual o soquete está ligado na máquina remota.
4	<b>public int getLocalPort ()</b> Retorna a porta na qual o soquete está ligado na máquina local.
5	<b>public SocketAddress getRemoteSocketAddress ()</b> Retorna o endereço do soquete remoto.
6	<b>public InputStream getInputStream () lança IOException</b> Retorna o fluxo de entrada do soquete. O fluxo de entrada é conectado ao fluxo de saída do soquete remoto.
7	<b>public OutputStream getOutputStream () lança IOException</b> Retorna o fluxo de saída do soquete. O fluxo de saída é conectado ao fluxo de entrada do soquete remoto.
8	<b>public void close () lança IOException</b> Fecha o soquete, o que faz com que esse objeto Soquete não seja mais capaz de se conectar novamente a qualquer servidor.

## Métodos de classe InetAddress

Esta classe representa um endereço IP (Internet Protocol). Aqui estão seguindo métodos úteis que você precisaria enquanto fazia programação de socket -



Sr. Não.	Método e Descrição
1	<b>InetAddress estático getByAddress (byte [] addr)</b> Retorna um objeto InetAddress dado o endereço IP bruto.
2	<b>InetAddress estático getByAddress (host de sequência de caracteres, byte [] addr)</b> Cria um InetAddress com base no nome do host e no endereço IP fornecidos.
3	<b>InetAddress estático getName (host de cadeia)</b> Determina o endereço IP de um host, dado o nome do host.
4	<b>String getHostAddress ()</b> Retorna a string do endereço IP na apresentação textual.
5	<b>String getHostName ()</b> Obtém o nome do host para esse endereço IP.
6	<b>InetAddress estático InetAddress getLocalHost ()</b> Retorna o host local.
7	<b>String toString ()</b> Converte esse endereço IP em uma string.

## Exemplo de cliente de soquete

O seguinte GreetingClient é um programa cliente que se conecta a um servidor usando um soquete, envia uma saudação e aguarda uma resposta.

### Exemplo

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {

    public static void main(String [] args) {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            System.out.println("Connecting to " + serverName + " on port " + port);
            Socket client = new Socket(serverName, port);

            System.out.println("Just connected to " + client.getRemoteSocketAddress());
        }
    }
}
```

```

        OutputStream outToServer = client.getOutputStream();
        DataOutputStream out = new DataOutputStream(outToServer);

        out.writeUTF("Hello from " + client.getLocalSocketAddress());
        InputStream inFromServer = client.getInputStream();
        DataInputStream in = new DataInputStream(inFromServer);

        System.out.println("Server says " + in.readUTF());
        client.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

## Exemplo de servidor de soquete

O seguinte programa GreetingServer é um exemplo de um aplicativo de servidor que usa a classe Socket para escutar clientes em um número de porta especificado por um argumento de linha de comando -

### Exemplo

```

// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
    private ServerSocket serverSocket;

    public GreetingServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(10000);
    }

    public void run() {
        while(true) {
            try {
                System.out.println("Waiting for client on port " +
                    serverSocket.getLocalPort() + "...");
                Socket server = serverSocket.accept();

                System.out.println("Just connected to " + server.getRemoteSocketAddress());
                DataInputStream in = new DataInputStream(server.getInputStream());

                System.out.println(in.readUTF());
                DataOutputStream out = new DataOutputStream(server.getOutputStream());
                out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress()
                    + "\nGoodbye!");
                server.close();

            } catch (SocketTimeoutException s) {
                System.out.println("Socket timed out!");
                break;
            } catch (IOException e) {
                e.printStackTrace();
                break;
            }
        }
    }

    public static void main(String [] args) {
        int port = Integer.parseInt(args[0]);
    }
}

```

```
try {
    Thread t = new GreetingServer(port);
    t.start();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Compile o cliente e o servidor e inicie o servidor da seguinte maneira -

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

Verifique o programa cliente da seguinte maneira -

## Saída

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

# Java - Envio de email

Enviar um e-mail usando seu aplicativo Java é simples o suficiente, mas, para começar, você deve ter o **JavaMail API** e o **Java Activation Framework (JAF)** instalados em sua máquina.

Você pode baixar a versão mais recente do JavaMail (Versão 1.2) do site padrão do Java.

Você pode baixar a versão mais recente do JAF (Versão 1.1.1) do site padrão do Java.

Faça o download e descompacte esses arquivos. Nos diretórios de nível superior recém-criados, você encontrará vários arquivos jar para ambos os aplicativos. Você precisa adicionar os arquivos **mail.jar** e **activation.jar** no seu CLASSPATH.

## Envie um e-mail simples

Aqui está um exemplo para enviar um simples e-mail da sua máquina. Supõe-se que o seu **host local** esteja conectado à Internet e seja capaz o suficiente para enviar um e-mail.

## Exemplo

```
// File Name SendEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail {

    public static void main(String [] args) {
```

```

// Recipient's email ID needs to be mentioned.
String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned
String from = "web@gmail.com";

// Assuming you are sending email from localhost
String host = "localhost";

// Get system properties
Properties properties = System.getProperties();

// Setup mail server
properties.setProperty("mail.smtp.host", host);

// Get the default Session object.
Session session = Session.getDefaultInstance(properties);

try {
    // Create a default MimeMessage object.
    MimeMessage message = new MimeMessage(session);

    // Set From: header field of the header.
    message.setFrom(new InternetAddress(from));

    // Set To: header field of the header.
    message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

    // Set Subject: header field
    message.setSubject("This is the Subject Line!");

    // Now set the actual message
    message.setText("This is actual message");

    // Send message
    Transport.send(message);
    System.out.println("Sent message successfully....");
} catch (MessagingException mex) {
    mex.printStackTrace();
}
}
}

```

Compile e execute este programa para enviar um simples e-mail -

## Saída

```

$ java SendEmail
Sent message successfully....

```

Se você quiser enviar um email para vários destinatários, os seguintes métodos serão usados para especificar vários IDs de email -

```

void addRecipients(Message.RecipientType type, Address[] addresses)
    throws MessagingException

```

Aqui está a descrição dos parâmetros -

**type** - Isso seria definido como TO, CC ou BCC. Aqui CC representa Carbon Copy e BCC representa Black Carbon Copy. Exemplo: *Message.RecipientType.TO*

**endereços** - esta é uma matriz de ID de e-mail. Você precisaria usar o método `InternetAddress ()` ao especificar IDs de email.

## Envie um e-mail em HTML

Aqui está um exemplo para enviar um e-mail em HTML da sua máquina. Aqui assume-se que o seu **localhost** está conectado à Internet e capaz o suficiente para enviar um e-mail.

Este exemplo é muito semelhante ao anterior, exceto que estamos usando o método `setContent ()` para definir o conteúdo cujo segundo argumento é "text / html" para especificar que o conteúdo HTML está incluído na mensagem.

Usando este exemplo, você pode enviar o conteúdo HTML que desejar.

### Exemplo

```
// File Name SendHTMLEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendHTMLEmail {

    public static void main(String [] args) {
        // Recipient's email ID needs to be mentioned.
        String to = "abcd@gmail.com";

        // Sender's email ID needs to be mentioned
        String from = "web@gmail.com";

        // Assuming you are sending email from localhost
        String host = "localhost";

        // Get system properties
        Properties properties = System.getProperties();

        // Setup mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default Session object.
        Session session = Session.getDefaultInstance(properties);

        try {
            // Create a default MimeMessage object.
            MimeMessage message = new MimeMessage(session);

            // Set From: header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To: header field of the header.
            message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

            // Set Subject: header field
            message.setSubject("This is the Subject Line!");

            // Send the actual HTML message, as big as you like
            message.setContent("<h1>This is actual message</h1>", "text/html");

            // Send message
            Transport.send(message);
            System.out.println("Sent message successfully....");
        } catch (MessagingException mex) {
            mex.printStackTrace();
        }
    }
}
```

```
}  
}
```

Compile e execute este programa para enviar um email em HTML -

## Saída

```
$ java SendHTMLEmail  
Sent message successfully....
```

## Enviar anexo em e-mail

Aqui está um exemplo para enviar um e-mail com anexo da sua máquina. Aqui assume-se que o seu **localhost** está conectado à internet e capaz o suficiente para enviar um e-mail.

## Exemplo

```
// File Name SendFileEmail.java  
  
import java.util.*;  
import javax.mail.*;  
import javax.mail.internet.*;  
import javax.activation.*;  
  
public class SendFileEmail {  
  
    public static void main(String [] args) {  
        // Recipient's email ID needs to be mentioned.  
        String to = "abcd@gmail.com";  
  
        // Sender's email ID needs to be mentioned  
        String from = "web@gmail.com";  
  
        // Assuming you are sending email from localhost  
        String host = "localhost";  
  
        // Get system properties  
        Properties properties = System.getProperties();  
  
        // Setup mail server  
        properties.setProperty("mail.smtp.host", host);  
  
        // Get the default Session object.  
        Session session = Session.getDefaultInstance(properties);  
  
        try {  
            // Create a default MimeMessage object.  
            MimeMessage message = new MimeMessage(session);  
  
            // Set From: header field of the header.  
            message.setFrom(new InternetAddress(from));  
  
            // Set To: header field of the header.  
            message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));  
  
            // Set Subject: header field  
            message.setSubject("This is the Subject Line!");  
  
            // Create the message part  
            BodyPart messageBodyPart = new MimeBodyPart();  
  
            // Fill the message  
            messageBodyPart.setText("This is message body");  

```

```

// Create a multipart message
Multipart multipart = new MimeMultipart();

// Set text message part
multipart.addBodyPart(messageBodyPart);

// Part two is attachment
messageBodyPart = new MimeBodyPart();
String filename = "file.txt";
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Send the complete message parts
message.setContent(multipart );

// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
} catch (MessagingException mex) {
    mex.printStackTrace();
}
}
}

```

Compile e execute este programa para enviar um email em HTML -

## Saída

```

$ java SendFileEmail
Sent message successfully....

```

## Peça de Autenticação do Usuário

Se for necessário fornecer ID de usuário e Senha ao servidor de e-mail para fins de autenticação, você poderá definir essas propriedades da seguinte maneira -

```

props.setProperty("mail.user", "myuser");
props.setProperty("mail.password", "mypwd");

```

O restante do mecanismo de envio de emails permaneceria conforme explicado acima.

## Java - Multithreading

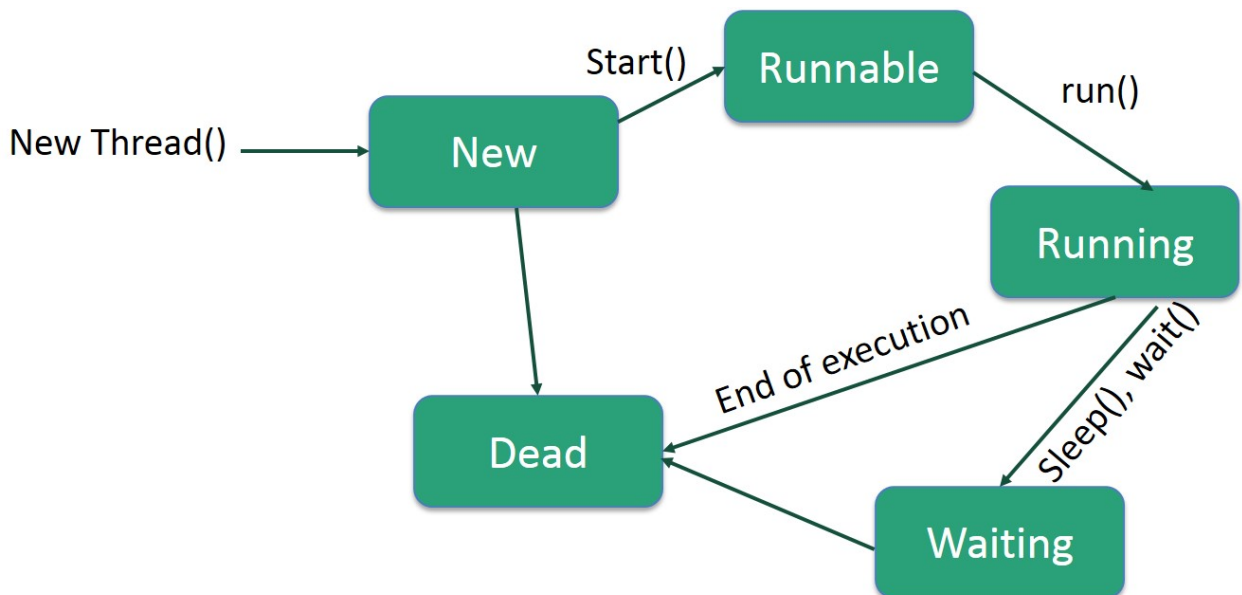
Java é uma *linguagem de programação multi-thread*, o que significa que podemos desenvolver um programa multi-thread usando Java. Um programa multi-threaded contém duas ou mais partes que podem ser executadas simultaneamente e cada parte pode manipular uma tarefa diferente ao mesmo tempo, otimizando o uso dos recursos disponíveis, especialmente quando o seu computador possui múltiplas CPUs.

Por definição, multitarefa é quando vários processos compartilham recursos de processamento comuns, como uma CPU. O multiencadeamento amplia a ideia de multitarefa em aplicativos nos quais você pode subdividir operações específicas em um único aplicativo em segmentos individuais. Cada um dos segmentos pode ser executado em paralelo. O sistema operacional divide o tempo de processamento não apenas entre aplicativos diferentes, mas também entre cada segmento dentro de um aplicativo.

O multi-threading permite que você escreva de uma maneira em que várias atividades podem prosseguir simultaneamente no mesmo programa.

## Ciclo de Vida de um Fio

Um thread passa por vários estágios em seu ciclo de vida. Por exemplo, um segmento é carregado, iniciado, executado e, em seguida, morre. O diagrama a seguir mostra o ciclo de vida completo de um encadeamento.



A seguir estão as etapas do ciclo de vida -

**Novo** - Um novo segmento inicia seu ciclo de vida no novo estado. Permanece nesse estado até que o programa inicie o thread. Também é referido como um **segmento nascido** .

**Runnable** - Depois que um thread recém-nascido é iniciado, o thread se torna executável. Considera-se que um encadeamento neste estado esteja executando sua tarefa.

**Aguardando** - Às vezes, um encadeamento faz a transição para o estado de espera enquanto o encadeamento aguarda que outro encadeamento realize uma tarefa. Um encadeamento transita de volta ao estado executável somente quando outro encadeamento sinaliza o encadeamento em espera para continuar a execução.

**Espera Temporizada** - Um encadeamento executável pode entrar no estado de espera temporizada por um intervalo de tempo especificado. Um encadeamento nesse estado faz a transição de volta para o estado executável quando esse intervalo de tempo expira ou quando o evento que está aguardando ocorre.

**Terminado (Morto)** - Um encadeamento executável entra no estado finalizado quando conclui sua tarefa ou termina.

## Prioridades de Thread



Cada encadeamento Java tem uma prioridade que ajuda o sistema operacional a determinar a ordem na qual os encadeamentos são planejados.

As prioridades de encadeamento Java estão no intervalo entre `MIN_PRIORITY` (uma constante de 1) e `MAX_PRIORITY` (uma constante de 10). Por padrão, cada thread recebe prioridade `NORM_PRIORITY` (uma constante de 5).

Threads com prioridade mais alta são mais importantes para um programa e devem ser alocados no tempo do processador antes dos threads de menor prioridade. No entanto, as prioridades de encadeamento não podem garantir a ordem na qual os encadeamentos são executados e são muito dependentes da plataforma.

## Criar um thread implementando uma interface executável

Se sua classe é destinada a ser executada como um thread, você pode conseguir isso implementando uma interface **Runnable**. Você precisará seguir três etapas básicas -

### Passo 1

Como primeiro passo, você precisa implementar um método `run()` fornecido por uma interface **Runnable**. Este método fornece um ponto de entrada para o encadeamento e você colocará sua lógica comercial completa dentro desse método. A seguir, uma sintaxe simples do método `run()` -

```
public void run( )
```

### Passo 2

Como segundo passo, você irá instanciar um objeto **Thread** usando o seguinte construtor -

```
Thread(Runnable threadObj, String threadName);
```

Onde, *threadObj* é uma instância de uma classe que implementa a interface **Runnable** e **threadName** é o nome dado ao novo thread.

### etapa 3

Quando um objeto `Thread` é criado, você pode iniciá-lo chamando o método **start()**, que executa uma chamada para o método `run()`. A seguir, uma sintaxe simples do método `start()` -

```
void start();
```

## Exemplo

Aqui está um exemplo que cria um novo segmento e começa a executá-lo -

```
class RunnableDemo implements Runnable {  
    private Thread t;  
    private String threadName;  
  
    RunnableDemo( String name) {  
        threadName = name;  
        System.out.println("Creating " + threadName );  
    }  
}
```

```

    }

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {

    public static void main(String args[]) {
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.

```

## Criar um thread estendendo uma classe de thread

A segunda maneira de criar um thread é criar uma nova classe que estenda a classe **Thread** usando as duas etapas simples a seguir. Essa abordagem fornece mais flexibilidade no tratamento de vários encadeamentos criados usando métodos disponíveis na classe Thread.

## Passo 1

Você precisará sobrescrever o método **run ()** disponível na classe Thread. Este método fornece um ponto de entrada para o encadeamento e você colocará sua lógica comercial completa dentro desse método. A seguir, uma sintaxe simples do método run () -

```
public void run( )
```

## Passo 2

Uma vez que o objeto Thread é criado, você pode iniciá-lo chamando o método **start ()** , que executa uma chamada ao método run (). A seguir, uma sintaxe simples do método start () -

```
void start( );
```

## Exemplo

Aqui está o programa anterior reescrito para estender o Thread -

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;

    ThreadDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {

    public static void main(String args[]) {
```

```

ThreadDemo T1 = new ThreadDemo( "Thread-1");
T1.start();

ThreadDemo T2 = new ThreadDemo( "Thread-2");
T2.start();
}
}

```

Isso produzirá o seguinte resultado -

## Saída

```

Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.

```

## Métodos de Thread

A seguir, a lista de métodos importantes disponíveis na classe Thread.

Sr. Não.	Método e Descrição
1	<b>public void start ()</b> Inicia o encadeamento em um caminho separado de execução e, em seguida, chama o método run () neste objeto Thread.
2	<b>corrida vazia pública ()</b> Se esse objeto Thread foi instanciado usando um destino Runnable separado, o método run () será chamado nesse objeto Runnable.
3	<b>public final void setName (nome da string)</b> Altera o nome do objeto Thread. Existe também um método getName () para recuperar o nome.
4	<b>public final void setPriority (prioridade int)</b>

	Define a prioridade deste objeto Thread. Os valores possíveis estão entre 1 e 10.
5	<b>public final void setDaemon (boolean on)</b> Um parâmetro true denota esse Thread como um thread do daemon.
6	<b>Junção void final pública (milissegundos longos)</b> O encadeamento atual invoca esse método em um segundo encadeamento, fazendo com que o encadeamento atual seja bloqueado até que o segundo encadeamento termine ou o número especificado de milissegundos seja aprovado.
7	<b>interrupção do vazio público ()</b> Interrompe este encadeamento, fazendo com que continue a execução se ele foi bloqueado por qualquer motivo.
8	<b>final público boolean isAlive ()</b> Retorna true se o encadeamento estiver ativo, o que é a qualquer momento após o encadeamento ter sido iniciado, mas antes de ser executado até a conclusão.

Os métodos anteriores são invocados em um determinado objeto Thread. Os seguintes métodos na classe Thread são estáticos. Invocar um dos métodos estáticos executa a operação no encadeamento atualmente em execução.

Sr. Não.	Método e Descrição
1	<b>rendimento vazio público estático ()</b> Faz com que o encadeamento atualmente em execução produza quaisquer outros encadeamentos da mesma prioridade que estão aguardando para serem planejados.
2	<b>sono estático público estático (longo milissegundo)</b> Faz com que o encadeamento atualmente em execução seja bloqueado por pelo menos o número especificado de milissegundos.
3	<b>retenções booleanas estáticas públicas (Objeto x)</b> Retorna true se o segmento atual contiver o bloqueio no objeto especificado.

4	<p><b>public static Thread currentThread ()</b></p> <p>Retorna uma referência ao encadeamento atualmente em execução, que é o encadeamento que invoca esse método.</p>
5	<p><b>public static void dumpStack ()</b></p> <p>Imprime o rastreamento de pilha para o encadeamento atualmente em execução, o que é útil ao depurar um aplicativo multithread.</p>

## Exemplo

O seguinte programa ThreadClassDemo demonstra alguns desses métodos da classe Thread. Considere uma classe **DisplayMessage** que implementa **Runnable** -

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable

public class DisplayMessage implements Runnable {
    private String message;

    public DisplayMessage(String message) {
        this.message = message;
    }

    public void run() {
        while(true) {
            System.out.println(message);
        }
    }
}
```

A seguir está outra classe que estende a classe Thread -

```
// File Name : GuessANumber.java
// Create a thread to extend Thread

public class GuessANumber extends Thread {
    private int number;
    public GuessANumber(int number) {
        this.number = number;
    }

    public void run() {
        int counter = 0;
        int guess = 0;
        do {
            guess = (int) (Math.random() * 100 + 1);
            System.out.println(this.getName() + " guesses " + guess);
            counter++;
        } while(guess != number);
        System.out.println("** Correct!" + this.getName() + " in " + counter + " guesses.**");
    }
}
```

A seguir é o programa principal, que faz uso das classes acima definidas -

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo {
```

```

public static void main(String [] args) {
    Runnable hello = new DisplayMessage("Hello");
    Thread thread1 = new Thread(hello);
    thread1.setDaemon(true);
    thread1.setName("hello");
    System.out.println("Starting hello thread...");
    thread1.start();

    Runnable bye = new DisplayMessage("Goodbye");
    Thread thread2 = new Thread(bye);
    thread2.setPriority(Thread.MIN_PRIORITY);
    thread2.setDaemon(true);
    System.out.println("Starting goodbye thread...");
    thread2.start();

    System.out.println("Starting thread3...");
    Thread thread3 = new GuessANumber(27);
    thread3.start();
    try {
        thread3.join();
    } catch (InterruptedException e) {
        System.out.println("Thread interrupted.");
    }
    System.out.println("Starting thread4...");
    Thread thread4 = new GuessANumber(75);

    thread4.start();
    System.out.println("main() is ending...");
}
}

```

Isso produzirá o seguinte resultado. Você pode tentar este exemplo várias vezes e obterá um resultado diferente a cada vez.

## Saída

```

Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.....

```

## Principais conceitos multithreading de Java

Ao fazer a programação Multithreading em Java, você precisaria ter os seguintes conceitos muito úteis -

O que é sincronização de threads?

Manipulando comunicação interthread

Manipulando o impasse do thread

Principais operações de thread

## Java - Noções básicas de applet

Um **applet** é um programa Java que é executado em um navegador da Web. Um applet pode ser um aplicativo Java totalmente funcional, pois possui toda a API Java à sua disposição.

Existem algumas diferenças importantes entre um applet e um aplicativo Java independente, incluindo o seguinte -

Um applet é uma classe Java que estende a classe `java.applet.Applet`.

Um método `main ()` não é chamado em um applet e uma classe de applet não define `main ()`.

Os applets são projetados para serem incorporados em uma página HTML.

Quando um usuário visualiza uma página HTML que contém um applet, o código do applet é baixado para a máquina do usuário.

Uma JVM é necessária para visualizar um applet. A JVM pode ser um plug-in do navegador da Web ou um ambiente de tempo de execução separado.

A JVM na máquina do usuário cria uma instância da classe de applet e invoca vários métodos durante a vida útil do applet.

Os applets possuem regras de segurança rígidas que são aplicadas pelo navegador da Web. A segurança de um applet é geralmente chamada de segurança do sandbox, comparando o applet a um filho que está jogando em um sandbox com várias regras que devem ser seguidas.

Outras classes que o applet precisa podem ser baixadas em um único arquivo Java Archive (JAR).

## Ciclo de Vida de um Applet

Quatro métodos na classe `Applet` fornecem a estrutura na qual você constrói qualquer applet sério -

**init** - Este método é destinado a qualquer inicialização necessária para o seu applet. Ele é chamado depois que as tags de parâmetro dentro da tag do applet foram processadas.

**start** - Este método é chamado automaticamente depois que o navegador chama o método `init`. Ele também é chamado sempre que o usuário retorna à página que contém o applet depois de ter ido para outras páginas.

**stop** - Esse método é chamado automaticamente quando o usuário sai da página em que o applet está. Pode, portanto, ser chamado repetidamente no mesmo



applet.

**destroy** - Esse método é chamado apenas quando o navegador é desligado normalmente. Como os applets são destinados a residir em uma página HTML, normalmente você não deve deixar recursos para trás depois que um usuário sair da página que contém o applet.

**paint** - Chamado imediatamente após o método start (), e também sempre que o applet precisar ser repintado no navegador. O método paint () é realmente herdado do java.awt.

## Um applet "Hello, World"

A seguir, um applet simples chamado HelloWorldApplet.java -

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
    public void paint (Graphics g) {
        g.drawString ("Hello World", 25, 50);
    }
}
```

Essas instruções de importação trazem as classes para o escopo de nossa classe de applet -

java.applet.Applet

java.awt.Graphics

Sem essas instruções de importação, o compilador Java não reconheceria as classes Applet e Graphics, às quais a classe de applet se refere.

## A classe de applet

Cada applet é uma extensão da *classe java.applet.Applet* . A classe de Applet base fornece métodos que uma classe de Applet derivada pode chamar para obter informações e serviços do contexto do navegador.

Estes incluem métodos que fazem o seguinte -

Obter parâmetros do applet

Obter o local de rede do arquivo HTML que contém o applet

Obter o local de rede do diretório de classes do applet

Imprimir uma mensagem de status no navegador

Buscar uma imagem

Buscar um clipe de áudio

Tocar um clipe de áudio

Redimensionar o applet

Além disso, a classe Applet fornece uma interface pela qual o visualizador ou navegador obtém informações sobre o applet e controla a execução do applet. O espectador pode -

- Solicitar informações sobre o autor, versão e direitos autorais do applet

- Solicite uma descrição dos parâmetros que o applet reconhece

- Inicialize o applet

- Destrua o applet

- Inicie a execução do applet

- Pare a execução do applet

A classe Applet fornece implementações padrão de cada um desses métodos. Essas implementações podem ser substituídas conforme necessário.

O applet "Hello, World" está completo como está. O único método substituído é o método de pintura.

## Invocando um applet

Um applet pode ser chamado pela incorporação de diretivas em um arquivo HTML e pela exibição do arquivo por meio de um visualizador de applet ou de um navegador ativado por Java.

A tag <applet> é a base para incorporar um applet em um arquivo HTML. A seguir, um exemplo que chama o applet "Hello, World" -

```
<html>
  <title>The Hello, World Applet</title>
  <hr>
  <applet code = "HelloWorldApplet.class" width = "320" height = "120">
    If your browser was Java-enabled, a "Hello, World"
    message would appear here.
  </applet>
  <hr>
</html>
```

**Nota** - Você pode consultar o [HTML Applet Tag](#) para entender mais sobre o applet de chamada do HTML.

O atributo de código da tag <applet> é obrigatório. Especifica a classe Applet a ser executada. Largura e altura também são necessárias para especificar o tamanho inicial do painel no qual um applet é executado. A diretiva de applet deve ser fechada com uma tag </ applet>.

Se um applet usa parâmetros, os valores podem ser passados para os parâmetros adicionando tags <param> entre <applet> e </ applet>. O navegador ignora texto e outras tags entre as tags do applet.

Navegadores não habilitados para Java não processam <applet> e </ applet>. Portanto, qualquer coisa que apareça entre as tags, não relacionada ao applet, é visível em navegadores não habilitados para Java.

O visualizador ou navegador procura o código Java compilado no local do documento. Para

especificar o contrário, use o atributo codebase da tag <applet> como mostrado -

```
<applet codebase = "https://amrood.com/applets" code = "HelloWorldApplet.class"
width = "320" height = "120">
```

Se um applet residir em um pacote diferente do padrão, o pacote de armazenamento deverá ser especificado no atributo code usando o caractere de ponto (.) Para separar componentes de pacote / classe. Por exemplo -

```
<applet code = "mypackage.subpackage.TestApplet.class"
width = "320" height = "120">
```

## Obtendo Parâmetros de Applet

O exemplo a seguir demonstra como fazer um applet responder aos parâmetros de configuração especificados no documento. Este applet exibe um padrão xadrez de preto e uma segunda cor.

A segunda cor e o tamanho de cada quadrado podem ser especificados como parâmetros para o applet no documento.

O CheckerApplet obtém seus parâmetros no método init (). Ele também pode obter seus parâmetros no método paint (). No entanto, obter os valores e salvar as configurações uma vez no início do applet, em vez de a cada atualização, é conveniente e eficiente.

O visualizador de applet ou navegador chama o método init () de cada applet que é executado. O visualizador chama init () uma vez, imediatamente após carregar o applet. (Applet.init () é implementado para não fazer nada.) Substitua a implementação padrão para inserir o código de inicialização personalizado.

O método Applet.getParameter () busca um parâmetro dado o nome do parâmetro (o valor de um parâmetro é sempre uma string). Se o valor for numérico ou outros dados que não sejam de caractere, a cadeia deverá ser analisada.

O seguinte é um esqueleto de CheckerApplet.java -

```
import java.applet.*;
import java.awt.*;

public class CheckerApplet extends Applet {
    int squareSize = 50; // initialized to default size
    public void init() {}
    private void parseSquareSize (String param) {}
    private Color parseColor (String param) {}
    public void paint (Graphics g) {}
}
```

Aqui estão os métodos init () e parseSquareSize () do CheckerApplet -

```
public void init () {
    String squareSizeParam = getParameter ("squareSize");
    parseSquareSize (squareSizeParam);

    String colorParam = getParameter ("color");
    Color fg = parseColor (colorParam);
}
```

```

        setBackground (Color.black);
        setForeground (fg);
    }

    private void parseSquareSize (String param) {
        if (param == null) return;
        try {
            squareSize = Integer.parseInt (param);
        } catch (Exception e) {
            // Let default value remain
        }
    }
}

```

O applet chama `parseSquareSize ()` para analisar o parâmetro `squareSize`. `parseSquareSize ()` chama o método de biblioteca `Integer.parseInt ()`, que analisa uma string e retorna um inteiro. `Integer.parseInt ()` lança uma exceção sempre que seu argumento é inválido.

Portanto, `parseSquareSize ()` captura exceções, em vez de permitir que o applet falhe na entrada incorreta.

O applet chama `parseColor ()` para analisar o parâmetro de cor em um valor `Color`. `parseColor ()` faz uma série de comparações de strings para combinar o valor do parâmetro com o nome de uma cor predefinida. Você precisa implementar esses métodos para fazer este applet funcionar.

## Especificando Parâmetros do Applet

A seguir, um exemplo de um arquivo HTML com um `CheckerApplet` incorporado a ele. O arquivo HTML especifica os dois parâmetros para o applet por meio da tag `<param>`.

```

<html>
<title>Checkerboard Applet</title>
<hr>
<applet code = "CheckerApplet.class" width = "480" height = "320">
    <param name = "color" value = "blue">
    <param name = "squareSize" value = "30">
</applet>
<hr>
</html>

```

**Nota** - Os nomes dos parâmetros não diferenciam maiúsculas de minúsculas.

## Conversão de aplicativo para applets

É fácil converter um aplicativo Java gráfico (ou seja, um aplicativo que usa o AWT e que você pode iniciar com o ativador de programas Java) em um applet que você pode incorporar em uma página da web.

A seguir estão as etapas específicas para converter um aplicativo em um applet.

Crie uma página HTML com a tag apropriada para carregar o código do applet.

Forneça uma subclasse da classe `JApplet`. Torne esta aula pública. Caso contrário, o applet não pode ser carregado.

Elimine o método principal na aplicação. Não construa uma janela de quadro para o

aplicativo. Seu aplicativo será exibido dentro do navegador.

Mova qualquer código de inicialização do construtor da janela de quadro para o método `init` do applet. Você não precisa construir explicitamente o objeto applet. O navegador instancia isso para você e chama o método `init`.

Remova a chamada para `setSize`; para applets, o dimensionamento é feito com os parâmetros `width` e `height` no arquivo HTML.

Remova a chamada para `setDefaultCloseOperation`. Um applet não pode ser fechado; termina quando o navegador sai.

Se o aplicativo chamar `setTitle`, elimine a chamada para o método. Os applets não podem ter barras de título. (Você pode, obviamente, intitular a própria página da Web, usando a tag de título HTML.)

Não chame `setVisible(true)`. O applet é exibido automaticamente.

Os applets herdam um grupo de métodos de manipulação de eventos da classe `Container`. A classe `Container` define vários métodos, como `processKeyEvent` e `processMouseEvent`, para manipular determinados tipos de eventos e, em seguida, um método catch-all chamado `processEvent`.

Para reagir a um evento, um applet deve substituir o método específico do evento apropriado.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet implements MouseListener {
    StringBuffer strBuffer;

    public void init() {
        addMouseListener(this);
        strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
        // Draw a Rectangle around the applet's display area.
```

```

        g.drawRect(0, 0,
        getWidth() - 1,
        getHeight() - 1);

        // display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }

    public void mouseEntered(MouseEvent event) {
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }
    public void mouseClicked(MouseEvent event) {
        addItem("mouse clicked! ");
    }
}

```

Agora, vamos chamar este applet da seguinte forma -

```

<html>
<title>Event Handling</title>
<hr>
<applet code = "ExampleEventHandling.class"
        width = "300" height = "300">
</applet>
<hr>
</html>

```

Inicialmente, o applet exibirá "inicializando o applet. Iniciando o applet". Então, quando você clicar dentro do retângulo, o "mouse clicado" também será exibido.

## Exibindo Imagens

Um applet pode exibir imagens do formato GIF, JPEG, BMP e outros. Para exibir uma imagem dentro do applet, use o método `drawImage ()` encontrado na classe `java.awt.Graphics`.

A seguir, um exemplo que ilustra todas as etapas para mostrar imagens.

```

import java.applet.*;
import java.awt.*;
import java.net.*;

public class ImageDemo extends Applet {
    private Image image;
    private AppletContext context;

    public void init() {
        context = this.getAppletContext();
        String imageURL = this.getParameter("image");
        if(imageURL == null) {
            imageURL = "java.jpg";
        }
        try {
            URL url = new URL(this.getDocumentBase(), imageURL);
            image = context.getImage(url);
        } catch (MalformedURLException e) {

```

```

        e.printStackTrace();
        // Display in browser status bar
        context.showStatus("Could not load image!");
    }
}

public void paint(Graphics g) {
    context.showStatus("Displaying image");
    g.drawImage(image, 0, 0, 200, 84, null);
    g.drawString("www.javalicense.com", 35, 100);
}
}

```

Agora, vamos chamar este applet da seguinte forma -

```

<html>
<title>The ImageDemo applet</title>
<hr>
<applet code = "ImageDemo.class" width = "300" height = "200">
  <param name = "image" value = "java.jpg">
</applet>
<hr>
</html>

```

## Tocando Áudio

Um applet pode reproduzir um arquivo de áudio representado pela interface AudioClip no pacote java.applet. A interface do AudioClip tem três métodos, incluindo -

**public void play ()** - Reproduz o clipe de áudio uma vez, desde o início.

**public void loop ()** - Faz com que o clipe de áudio seja repetido continuamente.

**public void stop ()** - Pára a reprodução do clipe de áudio.

Para obter um objeto AudioClip, você deve invocar o método getAudioClip () da classe Applet. O método getAudioClip () retorna imediatamente, independentemente de o URL resolver ou não um arquivo de áudio real. O arquivo de áudio não é baixado até que seja feita uma tentativa de reproduzir o clipe de áudio.

A seguir, um exemplo que ilustra todas as etapas para reproduzir um áudio

```

import java.applet.*;
import java.awt.*;
import java.net.*;

public class AudioDemo extends Applet {
    private AudioClip clip;
    private AppletContext context;

    public void init() {
        context = this.getAppletContext();
        String audioURL = this.getParameter("audio");
        if(audioURL == null) {
            audioURL = "default.au";
        }
        try {
            URL url = new URL(this.getDocumentBase(), audioURL);
            clip = context.getAudioClip(url);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}

```

```

        context.showStatus("Could not load audio file!");
    }
}

public void start() {
    if(clip != null) {
        clip.loop();
    }
}

public void stop() {
    if(clip != null) {
        clip.stop();
    }
}
}

```

Agora, vamos chamar este applet da seguinte forma -

```

<html>
<title>The ImageDemo applet</title>
<hr>
<applet code = "ImageDemo.class" width = "0" height = "0">
    <param name = "audio" value = "test.wav">
</applet>
<hr>
</html>

```

Você pode usar o test.wav no seu PC para testar o exemplo acima.

## Java - Comentários de documentação

A linguagem Java suporta três tipos de comentários -

Sr. Não.	Comentário e Descrição
1	<b><code>/ * text * /</code></b> O compilador ignora tudo de <code>/ *</code> para <code>* /</code> .
2	<b><code>//texto</code></b> O compilador ignora tudo, desde <code>//</code> até o final da linha.
3	<b><code>/ ** documentação * /</code></b> Este é um comentário de documentação e, em geral, é chamado de <b>comentário doc</b> . A ferramenta <b>javadoc do JDK</b> usa <i>comentários do documento</i> ao preparar a documentação gerada automaticamente.

Este capítulo é sobre como explicar o Javadoc. Vamos ver como podemos usar o Javadoc para gerar documentação útil para o código Java.

## O que é o Javadoc?

Javadoc é uma ferramenta que vem com o JDK e é usada para gerar a documentação do



código Java no formato HTML a partir do código-fonte Java, que requer documentação em um formato predefinido.

A seguir, um exemplo simples em que as linhas dentro de `/*...*/` são comentários de várias linhas Java. Da mesma forma, a linha que precede `//` é o comentário de linha única Java.

### Exemplo

```
/**
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 *
 * @author  Zara Ali
 * @version 1.0
 * @since   2014-03-31
 */
public class HelloWorld {

    public static void main(String[] args) {
        /* Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

Você pode incluir as tags HTML necessárias dentro da parte da descrição. Por exemplo, o exemplo a seguir faz uso de `<h1> .... </ h1>` para título e `<p>` foi usado para criar quebra de parágrafo -

### Exemplo

```
/**
 * <h1>Hello, World!</h1>
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 * <p>
 * Giving proper comments in your program makes it more
 * user friendly and it is assumed as a high quality code.
 *
 *
 * @author  Zara Ali
 * @version 1.0
 * @since   2014-03-31
 */
public class HelloWorld {

    public static void main(String[] args) {
        /* Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

## The javadoc Tags

A ferramenta javadoc reconhece os seguintes tags -

Tag	Descrição	Sintaxe
@autor	Adiciona o autor de uma turma.	@author nome-texto

<code>{@código}</code>	Exibe texto na fonte do código sem interpretar o texto como marcação HTML ou tags javadoc aninhadas.	<code>{@code text}</code>
<code>{@docRoot}</code>	Representa o caminho relativo para o diretório raiz do documento gerado a partir de qualquer página gerada.	<code>{@docRoot}</code>
<code>@descontinuada</code>	Adiciona um comentário indicando que essa API não deve mais ser usada.	<code>@prected deprecatedtext</code>
<code>@exceção</code>	Adiciona um subtítulo <b>Throws</b> à documentação gerada, com o nome da classe e o texto da descrição.	<code>@exception nome da classe descrição</code>
<code>{@inheritDoc}</code>	Herda um comentário da classe herdável <b>mais próxima</b> ou da interface implementável.	Herda um comentário da superclass imediata.
<code>{@ligação}</code>	Insere um link sequencial com o rótulo de texto visível que aponta para a documentação do pacote, classe ou nome de membro especificado de uma classe referenciada.	<code>{@link package.class # label de membro}</code>
<code>{@linkplain}</code>	Idêntico ao <code>{@link}</code> , exceto que o rótulo do link é exibido em texto simples do que a fonte do código.	<code>{@linkplain package.class # label de membro}</code>
<code>@param</code>	Adiciona um parâmetro com o nome do parâmetro especificado seguido pela descrição especificada na seção "Parâmetros".	<code>@param descrição do nome do parâmetro</code>
<code>@Retorna</code>	Adiciona uma seção "Retorna" com o texto da descrição.	<code>@retiro descrição</code>
<code>@Vejo</code>	Adiciona um título "Consulte também" com um link ou entrada de texto que aponta para referência.	<code>@ver referência</code>
<code>@serial</code>	Usado no comentário doc para um campo serializável padrão.	<code>@serial-description   include   excluir</code>
<code>@serialData</code>	Documenta os dados gravados pelos métodos <code>writeObject ()</code> ou <code>writeExternal ()</code> .	<code>@serialData data-description</code>
<code>@serialField</code>	Documenta um componente <code>ObjectStreamField</code> .	<code>@serialField campo-nome tipo de campo campo-descrição</code>
<code>@Desde a</code>	Adiciona um título "Since" com o texto <code>since</code> especificado à documentação gerada.	<code>@desde lançamento</code>
<code>@throws</code>	As tags <code>@throws</code> e <code>@exception</code> são sinônimos.	<code>@throws descrição do nome da classe</code>

{@valor}	Quando {@value} é usado no comentário doc de um campo estático, ele exibe o valor dessa constante.	{@value package.class # field}
@versão	Adiciona um subtítulo "Versão" com o texto da versão especificado aos documentos gerados quando a opção -version é usada.	@version version-text

O programa a seguir usa algumas das tags importantes disponíveis para comentários de documentação. Você pode usar outras tags com base em seus requisitos.

A documentação sobre a classe AddNum será produzida no arquivo HTML AddNum.html, mas ao mesmo tempo um arquivo mestre com um nome index.html também será criado.

```
import java.io.*;

/**
 * <h1>Add Two Numbers!</h1>
 * The AddNum program implements an application that
 * simply adds two given integer numbers and Prints
 * the output on the screen.
 * <p>
 * <b>Note:</b> Giving proper comments in your program makes it more
 * user friendly and it is assumed as a high quality code.
 *
 * @author Zara Ali
 * @version 1.0
 * @since 2014-03-31
 */
public class AddNum {
    /**
     * This method is used to add two integers. This is
     * a the simplest form of a class method, just to
     * show the usage of various javadoc Tags.
     * @param numA This is the first paramter to addNum method
     * @param numB This is the second parameter to addNum method
     * @return int This returns sum of numA and numB.
     */
    public int addNum(int numA, int numB) {
        return numA + numB;
    }

    /**
     * This is the main method which makes use of addNum method.
     * @param args Unused.
     * @return Nothing.
     * @exception IOException On input error.
     * @see IOException
     */

    public static void main(String args[]) throws IOException {
        AddNum obj = new AddNum();
        int sum = obj.addNum(10, 20);

        System.out.println("Sum of 10 and 20 is :" + sum);
    }
}
```

Agora, processe o arquivo AddNum.java acima usando o utilitário javadoc da seguinte maneira -

```
$ javadoc AddNum.java
Loading source file AddNum.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_51
Building tree for all the packages and classes...
Generating /AddNum.html...
AddNum.java:36: warning - @return tag cannot be used in method with void return type.
Generating /package-frame.html...
Generating /package-summary.html...
Generating /package-tree.html...
Generating /constant-values.html...
Building index for all the packages and classes...
Generating /overview-tree.html...
Generating /index-all.html...
Generating /deprecated-list.html...
Building index for all classes...
Generating /allclasses-frame.html...
Generating /allclasses-noframe.html...
Generating /index.html...
Generating /help-doc.html...
1 warning
$
```

Você pode verificar toda a documentação gerada aqui - AddNum . Se você estiver usando o JDK 1.7, o javadoc não gerará uma grande **stylesheet.css** , portanto, sugerimos que você faça o download e use uma folha de estilo padrão em <https://docs.oracle.com/javase/7/docs/api/stylesheet.css>



Página anterior

Próxima página



---

Anúncios