

A Tecnologia Java.

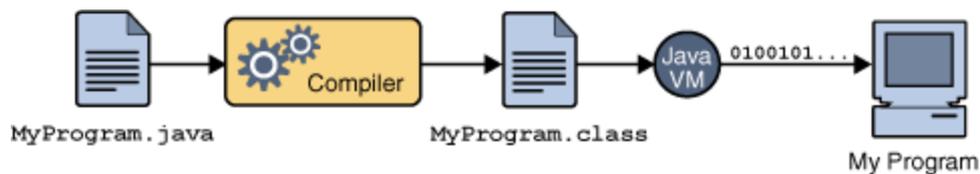
A tecnologia Java refere-se a ambas: a linguagem de programação e a plataforma.

A Linguagem de Programação Java.

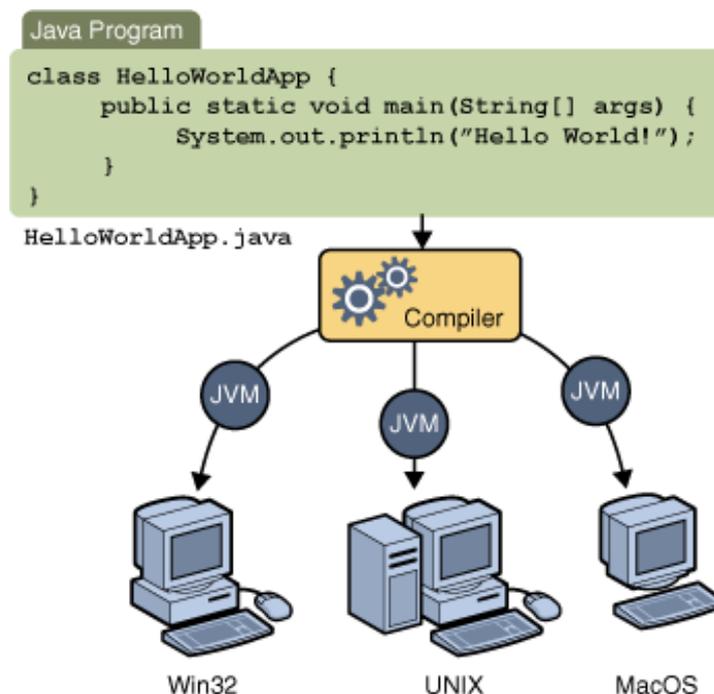
A linguagem de programação Java é uma linguagem de alto nível que pode ser caracterizada por todas os seguintes características:

- **Simples;**
- **Orientada a Objetos;**
- **Distribuída;**
- **Multithread;**
- **Dinâmica;**
- **Arquitetura Neutra;**
- **Portável;**
- **Alta Performance;**
- **Robusta;**
- **Segura.**

Na linguagem de programação Java, todos os códigos-fonte são primeiramente escritos em um arquivo de texto com a extensão `.java`. Esses arquivos-fonte são compilados nos arquivos `.class` pelo compilador `javac`. O arquivo `.class` não contém código que é nativo para seu processador; ele em seu lugar contém **bytecodes** – a linguagem de máquina da **Java Virtual Machine (Java VM)**. A ferramenta lançador java (**java launcher**) então roda sua aplicação com a instância da **Java Virtual Machine**.



Como a **Java VM** é disponível em diferentes sistemas operacionais, os mesmos arquivo `.class` são capazes de rodar em Microsoft Windows, Solaris Operating System (Solaris OS), Linux, ou Mac OS. Algumas máquinas virtuais, como a **JavaHotSpot virtual machine**, permitem marchas adicionais de tempo de execução para dar a suas aplicações uma performance melhorada. Ela inclui vários serviços como a procura por gargalos de performance e recompilação (para o código nativo) freqüentemente usada em seções do código.



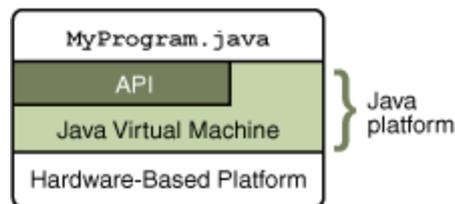
A Plataforma Java.

A **plataforma** é o ambiente de hardware ou software no qual os programas rodam. Nós mencionamos algumas das plataformas mais populares como o Microsoft Windows, Linux, Solaris OS, e Mac Os. Muitas plataformas podem ser descritas em uma combinação do sistema operacional e hardware subjacente. A plataforma Java difere de muitas outras plataformas nas quais a plataforma de software só roda em determinadas plataformas de hardware.

A plataforma Java tem dois componentes:

- **A Java Virtual Machine (Máquina Virtual Java);**
- **A Java Application Programming Interface (API);**

A **API** é uma grande coleção de componentes de software prontos para fornecer muitas capacidades convenientes. Ela é agrupada em bibliotecas de classes e interfaces; essas bibliotecas são conhecidas como **packages** (pacotes).



Como uma plataforma independente de desenvolvimento, a plataforma Java pode ser um pouco mais lenta que o código nativo. No entanto, avanços no compilador e a tecnologia virtual machine estão trazendo performance para o código fonte sem ameaçar a portabilidade.

//=====

O que a tecnologia Java pode fazer?

A linguagem de programação de alto nível Java é uma plataforma poderosa de software. Qualquer implementação completa da plataforma Java dá a você as seguintes características:

- **Ferramentas de Desenvolvimento:** As ferramentas de desenvolvimento fornecem qualquer coisa que você precisar para compilação, monitoramento, **debugging** e documentação de suas aplicações. Como um novo desenvolvedor, as ferramentas principais que você estará usando são o compilador **javac**, o lançador **java**, e a ferramenta de documentação **javadoc**.
- **Application Programming Interface (API):** A API fornece o núcleo da funcionalidade da linguagem de programação Java. Ela oferece um vasto conjunto de classes convenientes para uso em suas aplicações. Ela vai desde qualquer coisa sobre objetos básicos, até **networking** e segurança, geração **XML** e acesso a banco de dados, e mais. O núcleo API é muito extenso; para conseguir uma visão geral do que ela contém, consulte a **Java SE Development Kit 6 (JDK™ 6) documentation**.
- **Tecnologias de Desenvolvimento:** O software **JDK** fornece mecanismos padrão como o software **Java Web Start** e **Java Plug-In** para organizar suas aplicações para os usuários finais.
- **User Interface Toolkits:** Os kits de ferramentas **Swing** e **Java 2D** tornam possível criar sofisticadas **GUIs (Graphical User Interfaces)**.
- **Integration Libraries:** Bibliotecas de integração como a **Java IDL, API, JDBC™ API, Java Naming and Directory Interface™ (“J.N.D.I”) API, Java RMI, e Java Remote Method Invocation over Internet Inter-ORB Protocol Technology (Java RMI-IIOP Technology)** habilitam acesso a banco de dados e manipulação de objetos remotos.

//=====

Como a Tecnologia Java pode ajudar você?

- **Iniciar rapidamente:** Apesar da linguagem de programação Java ser uma poderosa linguagem de programação orientada a objetos, ela é fácil de aprender, especialmente para programadores que têm familiaridade com C ou C++.
- **Escreva menos códigos:** Comparações de medidas de programas (contagem de classes, contagem de métodos, etc) sugerem que um programa escrito em linguagem de programação Java pode ser quatro vezes menor que o mesmo programa escrito em C++.
- **Escreva códigos melhores:** A linguagem de programação Java encoraja boas práticas de programação, e automaticamente limpa o lixo de memória para ajudar você a evitar vazamento de memória.
- **Desenvolva programas mais rapidamente:** A linguagem de programação Java é mais simples que C++, e dessa maneira, seu tempo de desenvolvimento diminui pela metade quando escrevendo nele. Seus programas também requerem menos linhas de código.
- **Evita dependências de plataformas:** Você pode manter seus programas portáteis evitando o uso de bibliotecas escritas em outras plataformas.
- **Escreva uma vez, rode em qualquer lugar:** Pelo motivo de as aplicações escritas em linguagem de programação Java serem compiladas em uma máquina virtual independente de **bytecodes**, eles podem rodar consistentemente em qualquer plataforma Java.
- **Distribuição facilitada de software:** Com **Java Web Start software**, usuários estarão habilitados a usar suas aplicações com um simples clique do mouse. Uma versão automática checa no início para assegurar que os usuários têm sempre a última versão de seu software. Se uma **update** está disponível, o **Java Web Start** automaticamente atualizará seu software.

A Aplicação “Hello World!”.

Este capítulo fornece instruções detalhadas para compilar e rodar uma simples aplicação “Hello World!”. A primeira seção fornece informações para iniciar o **NetBeans IDE**, um ambiente integrado de desenvolvimento que simplifica grandemente o processo de desenvolvimento de software. A **IDE NetBeans** roda em todas as plataformas já citadas. O restante do capítulo fornece instruções específicas da plataforma para conseguir iniciar sem um ambiente integrado de desenvolvimento.

“Hello World!” para a IDE NetBeans.

Já é o momento para você escrever sua primeira aplicação. Estas instruções detalhadas são para usuários da **IDE NetBeans**. A **IDE NetBeans** roda na plataforma Java, que você pode usar em qualquer sistema operacional desde que tenha o **JDK 6** disponível.

Um Checklist.

Para escrever seu primeiro programa, você precisará:

1. O Java SE Development Kit 6 (JDK 6)

- Para Microsoft Windows, Solaris OS, e Linux: <http://java.sun.com/javase/6/download.jsp>
- Para Mac OS X: <http://connect.apple.com>

2. A IDE NetBeans

- Para todas as plataformas: <http://www.netbeans.info/downloads/index.php>

Criando sua primeira aplicação.

Sua primeira aplicação, **HelloWorldApp**, simplesmente mostrará a saudação “Hello World!”. Para criar esse programa, você precisará:

- **Criar um projeto IDE:** Quando você cria um projeto IDE, você cria um ambiente no qual você pode construir e rodar suas aplicações. Usando projetos IDE elimina configurações problemáticas normalmente associadas com o desenvolvimento em linha de comando. Você pode construir ou rodar suas aplicações escolhendo um simples item de menu dentro da IDE.
- **Adicionar código ao arquivo fonte gerado:** Um arquivo fonte contém código, escrito em linguagem de programação Java, que você e outros programadores podem entender. Como parte da criação de um projeto IDE, um esqueleto de arquivo fonte será automaticamente gerado. Você então modificará o arquivo fonte adicionando a mensagem “Hello World!”.
- **Compilar o arquivo fonte em um arquivo .class:** A IDE chama o compilador (**javac**) da linguagem de programação Java, que pega seu arquivo fonte e traduz esse texto em instruções que a Java virtual machine pode entender. As instruções escritas contidas dentro do arquivo são conhecidas como **bytecodes**.
- **Rodar o programa:** A IDE chama aplicação Java **launcher tool** (**java**), que usa a Java virtual machine para rodar seu programa.

```
//=====
```

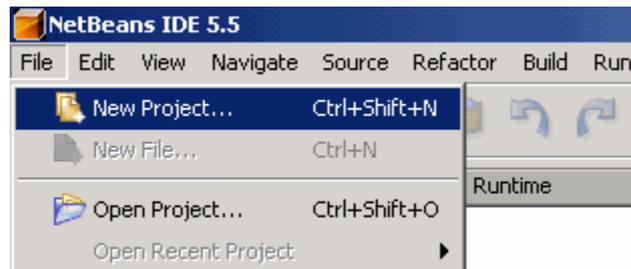
Criando um projeto IDE.

Para criar um projeto IDE:

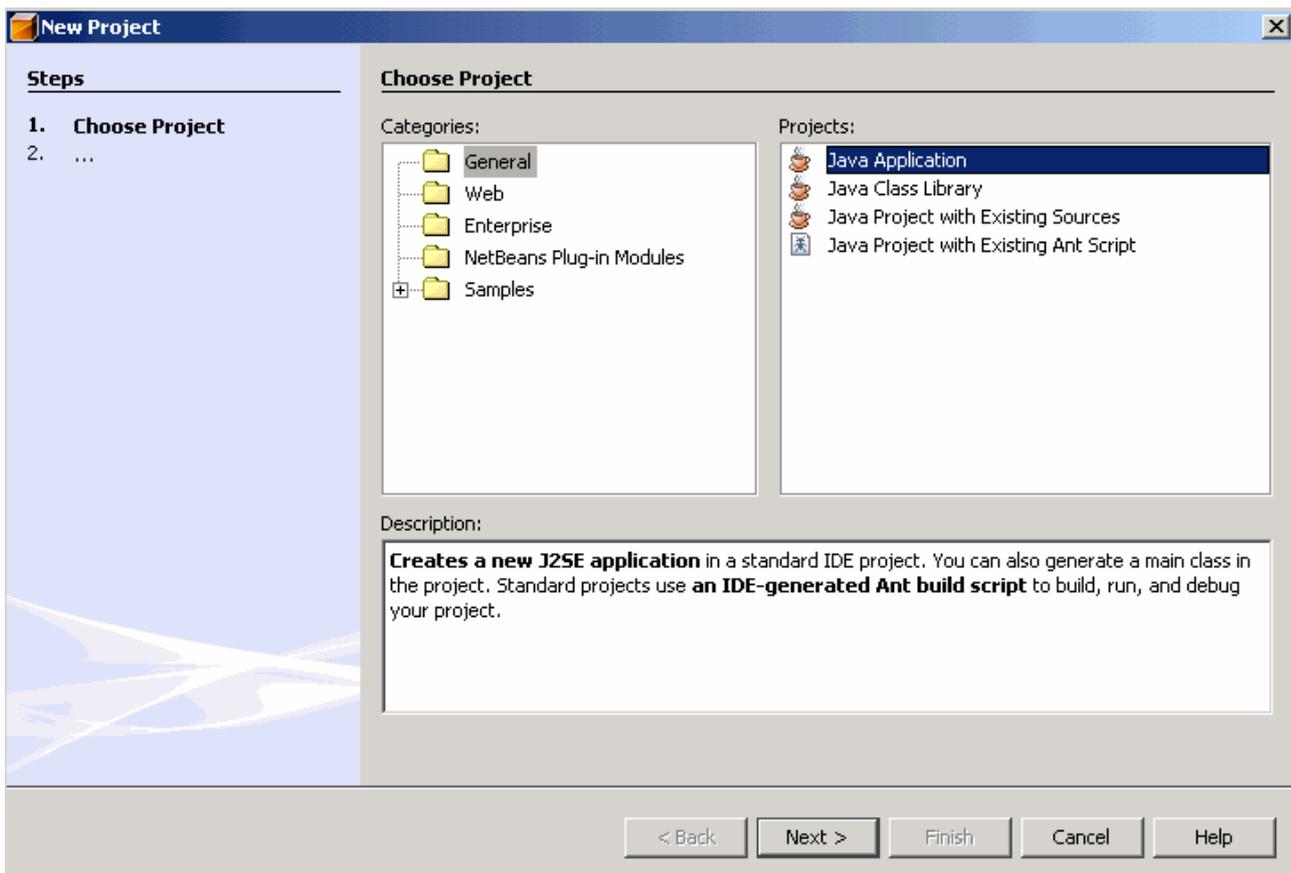
1. Inicie a IDE NetBeans.

- No sistema Microsoft Windows, você pode encontrar o item NetBeans IDE no menu Iniciar.
- No Solaris OS e sistemas Linux, você executa o script iniciador da IDE pela navegação do diretório `bin` e teclando `./netbeans`.
- No sistema Mac OS X, clique no ícone da aplicação NetBeans IDE.

2. Na IDE do NetBeans, escolha Arquivo | Novo Projeto.

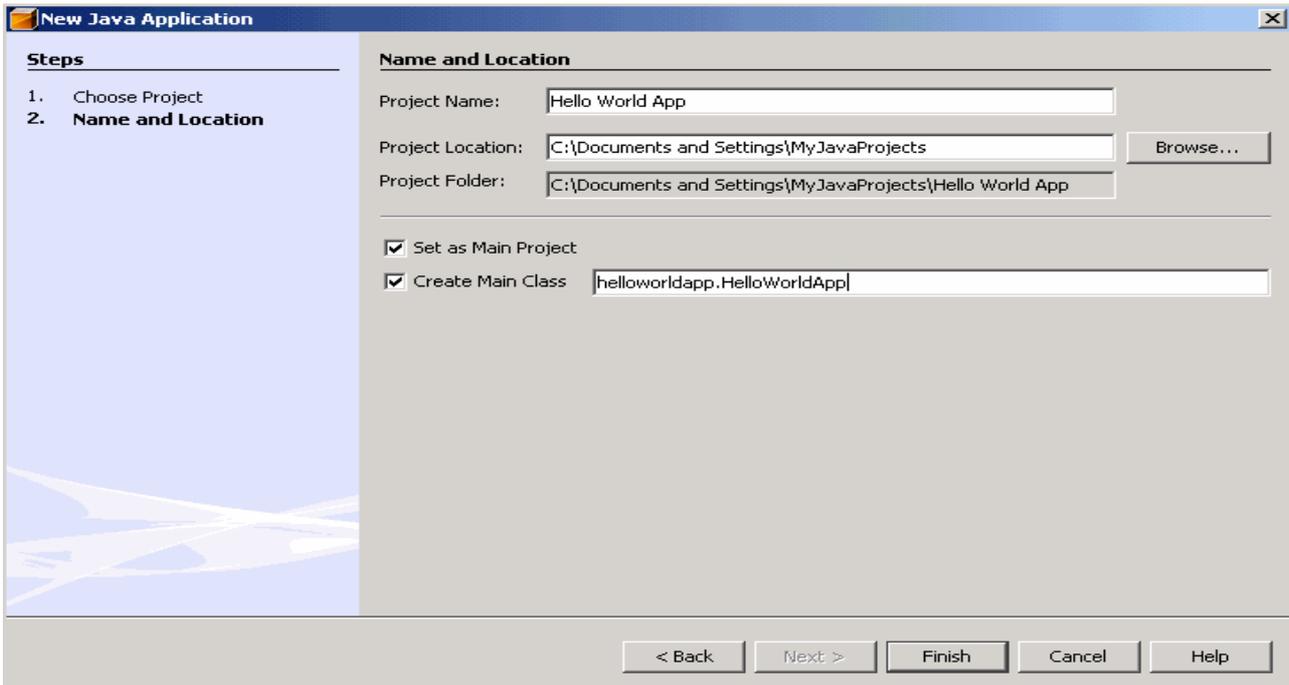


3. Na janela Novo Projeto, expanda a categoria Geral e selecione Aplicação Java:



4. Na página de Nome e Localização, faça o seguinte:

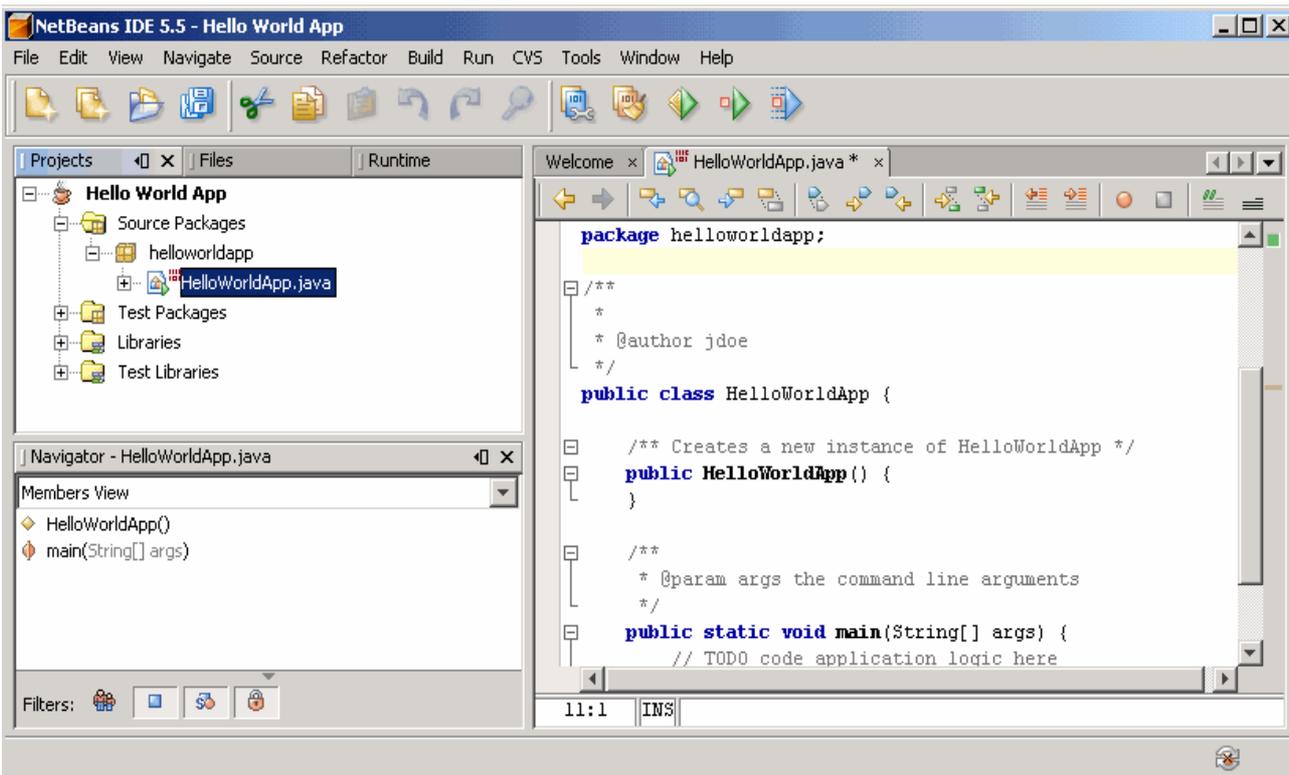
- No campo Nome do Projeto, digite **Hello World App**.
- No campo Criar Projeto Principal, digite **helloworldapp.HelloWorldApp**.
- Certifique-se que o checkbox Definir como Projeto Principal esteja marcado.



5. Clique Finalizar.

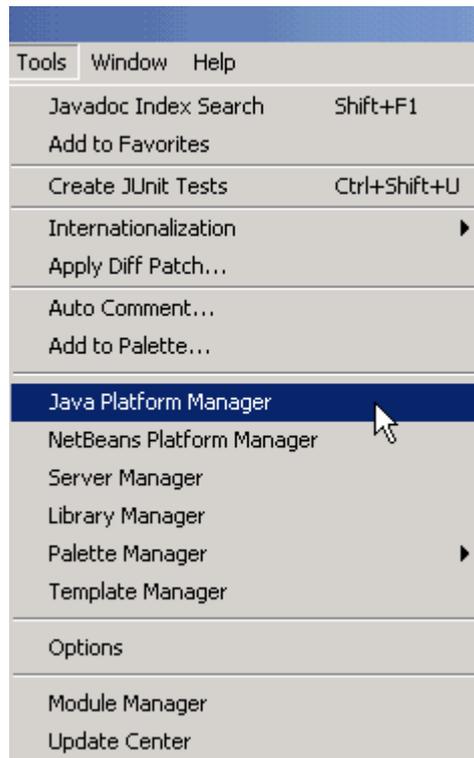
O projeto é criado e aberto na IDE. Você pode ver os seguintes componentes:

- A janela de **projetos**, que contém uma visão em árvore dos componentes do projeto, incluindo arquivos fonte, bibliotecas que seu código depende, e assim por diante.
- A janela do **Editor de Códigos** com o arquivo chamado **HelloWorldApp** aberto.
- A janela **Navegador**, que você pode usar para navegar rapidamente entre os elementos dentro da classe selecionada.

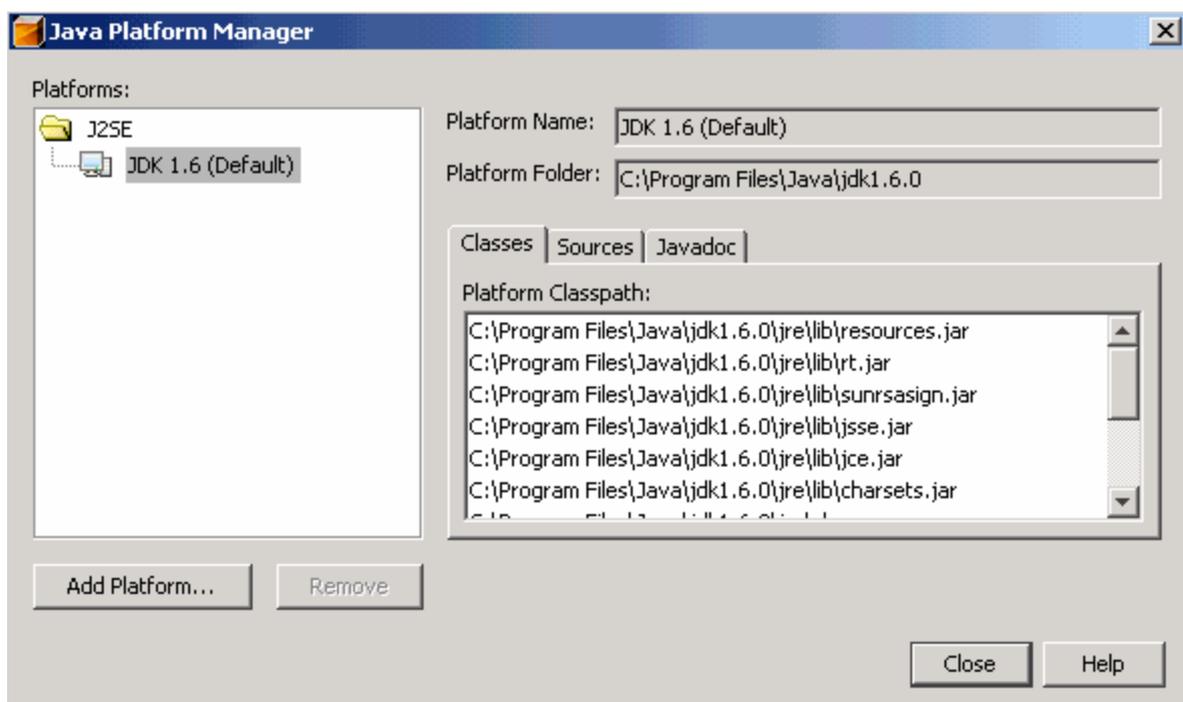


Adicionando o JDK 6 para a Plataforma (se necessário).

Pode ser necessário adicionar o JDK 6 para a lista da plataformas IDE disponível. Para fazer isso, escolha **Ferramentas | Gerenciador da Plataforma Java**, como mostra a figura:

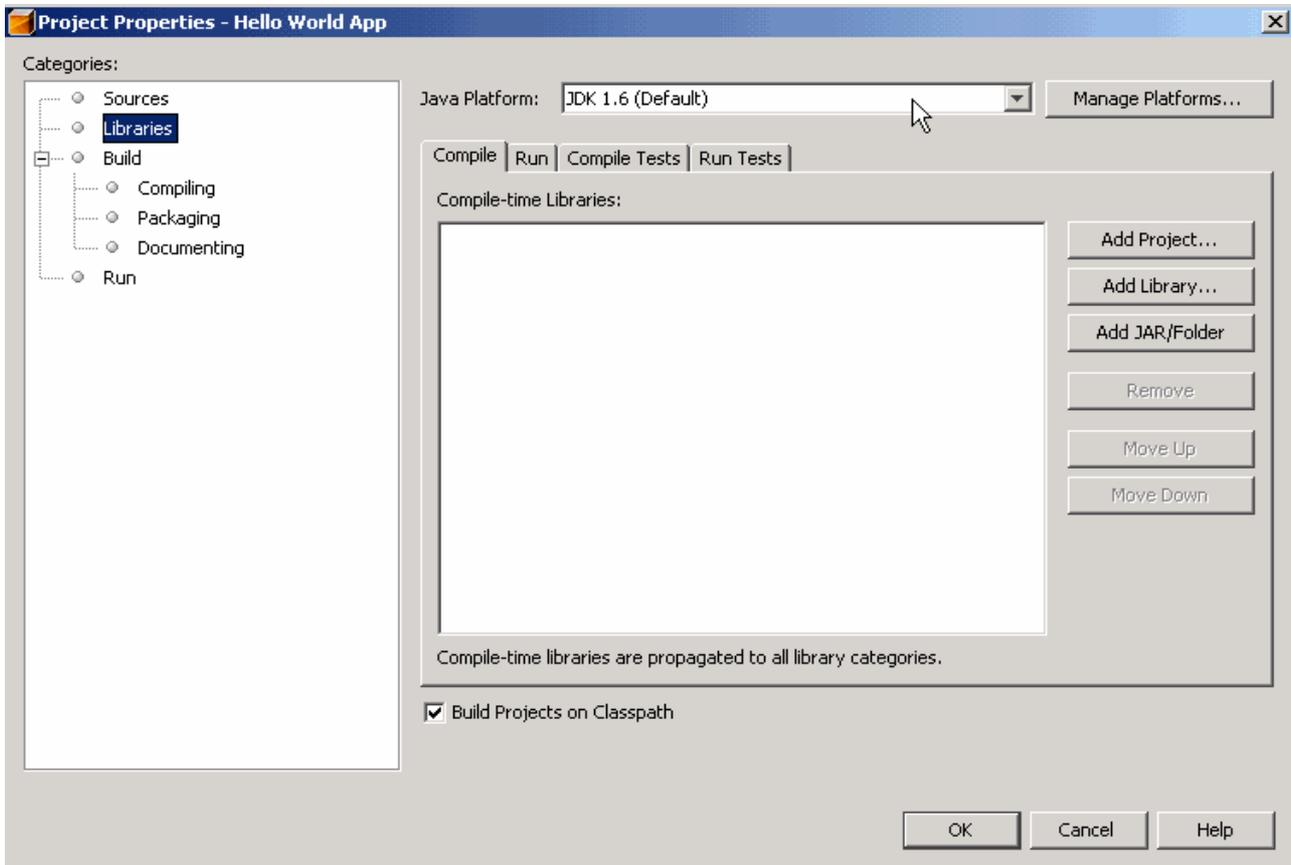


Se você não vê o JDK 6 (que às vezes aparece como 1.6 ou 1.6.0) na lista de plataformas instaladas, clique **“Adicionar Plataforma”**, navegue para seu diretório de instalação do JDK 6, clique **“Finalizar”**. Você verá agora a nova plataforma adicionada:



Para escolher esta JDK como padrão para todos os seus projetos, você pode rodar a IDE com o comando `--jdkhome` em linha de comando, ou introduzindo o path para o JDK na propriedade `netbeans_j2sdkhome` de seu arquivo `INSTALLATION_DIRECTORY/etc/netbeans.conf`.

Para especificar esta JDK somente para o projeto corrente, selecione **Hello World App** no painel **Projetos**, escolha **Arquivo | Propriedades de "Hello World App"**, clique em **Bibliotecas**, então selecione **JDK 6** abaixo do menu Java Plataforma. Você verá uma tela semelhante à seguinte:



Adicionando Código ao Arquivo Fonte Gerado.

Quando você cria seu projeto, você deixa o checkbox **Criar Classe Principal** selecionado na janela **Novo Projeto**. A IDE conseqüentemente terá criado um esqueleto da classe para você. Você pode adicionar a mensagem "Hello World!" para o esqueleto do código, reescrevendo a linha:

```
//TODO code application logic here
```

com a linha:

```
System.out.println("Hello World!"); //Mostra a string
```

Opcionalmente, você pode reescrever estas quatro linhas do código gerado:

```
/**
 *
 * @autor
 */
```

com estas linhas:

```
/**
 * Esta classe HelloWorldApp implementa uma aplicação que
 * simplesmente mostra a mensagem "Hello World" na saída padrão.
 */
```

Estas quatro linhas são um comentário de código e não afetam como o programa vai rodar.

Salve suas mudanças escolhendo **Arquivo | Salvar**.

O arquivo ficará parecido com isto:

```

/*
 * HelloWorldApp.java
 *
 * Created em September 29, 2007, 3:58 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package helloworldapp;

/**
 * Esta classe HelloWorldApp implementa uma aplicação que
 * simplesmente mostra a mensagem "Hello World" na saída padrão.
 */

public class HelloWorldApp {

    /** Creates a new instance of HelloWorldApp */
    public HelloWorldApp() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Mostra a string.
    }

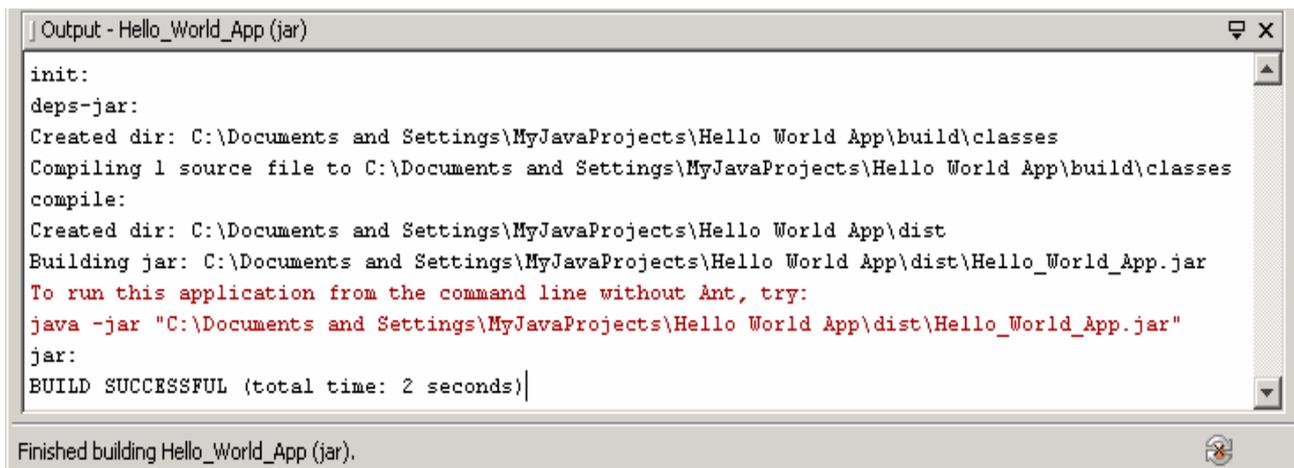
}

```

Compilando o Arquivo Fonte em um Arquivo .class.

Para compilar seu arquivo fonte, escolha **Construir | Construir Projeto Principal** no menu principal da IDE.

A janela **Saída** abre e mostra uma saída parecida com a que você vê na figura:



```

] Output - Hello_World_App (jar)
init:
deps-jar:
Created dir: C:\Documents and Settings\MyJavaProjects\Hello World App\build\classes
Compiling 1 source file to C:\Documents and Settings\MyJavaProjects\Hello World App\build\classes
compile:
Created dir: C:\Documents and Settings\MyJavaProjects\Hello World App\dist
Building jar: C:\Documents and Settings\MyJavaProjects\Hello World App\dist\Hello_World_App.jar
To run this application from the command line without Ant, try:
java -jar "C:\Documents and Settings\MyJavaProjects\Hello World App\dist\Hello_World_App.jar"
jar:
BUILD SUCCESSFUL (total time: 2 seconds)

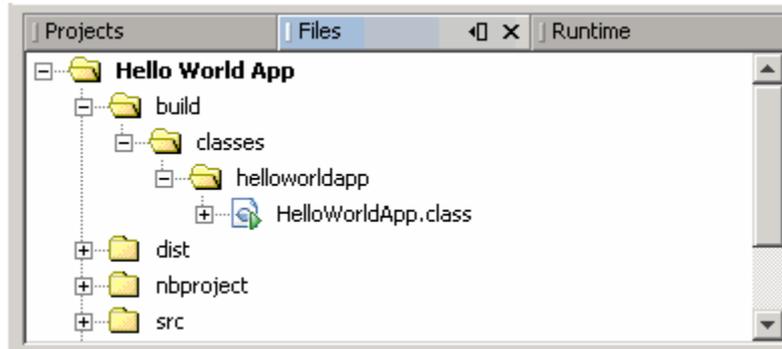
```

Finished building Hello_World_App (jar).

Se a saída do construtor concluir com a declaração **EXECUTADO COM SUCESSO**, congratulações! Você compilou com sucesso seu programa!

Se a saída do construtor concluir com a declaração **FALHA NA EXECUÇÃO**, você provavelmente teve um erro de sintaxe em seu código. Erros são reportados na janela Saída como um texto *hyper-linked*. Você pode dar um duplo-clique no *hyper-link* para navegar para a fonte de um erro. Você pode então consertar o erro e escolher novamente **Construir | Construir Projeto Principal**.

Quando você constrói o projeto, o arquivo bytecode `HelloWorldApp.class` é gerado. Você pode ver onde o novo arquivo foi gerado abrindo a janela **Arquivos** e expandindo o nó `Hello World App/build/classes/helloworldapp` como mostra a seguinte figura:

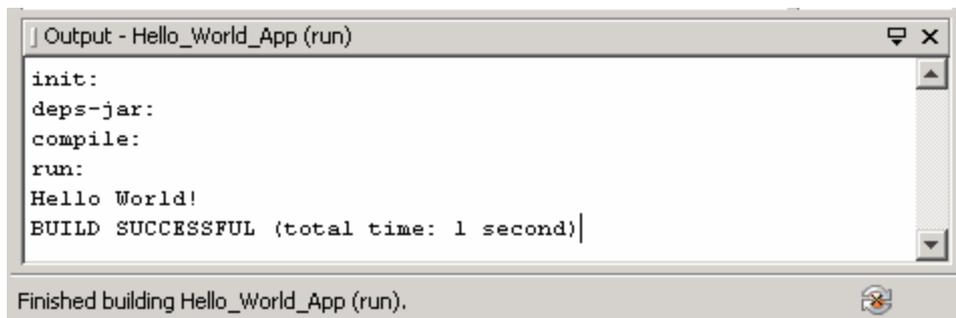


Agora que você já construiu o projeto, você pode rodar o programa.

Rodando o Programa.

No menu da IDE, escolha **Executar | Executar Projeto Principal**.

A próxima figura mostra o que você verá agora:



O programa mostra “Hello World!” na janela Saída (juntamente com outras saídas do script construtor).

Congratulações! Seu programa funciona!

Continuando o tutorial com a IDE NetBeans.

Nas próximas páginas do tutorial, será explicado o código em uma aplicação simples. Depois disso, as lições vão aprofundar-se nas características centrais da linguagem e fornecer muitos outros exemplos. Apesar de o restante do tutorial não dar instruções específicas a respeito do uso da IDE NetBeans, você pode facilmente usar a IDE para escrever e rodar os códigos de exemplo. A seguir estão algumas dicas no uso da IDE e explicações de alguns comportamentos da IDE que você provavelmente verá:

- Uma vez que você tenha criado um projeto em um IDE, você pode adicionar arquivos para o projeto usando a janela **Novo Arquivo**. Escolha **Arquivo | Novo Arquivo**, e então selecione um modelo na janela, como um modelo de **Arquivo Vazio Java**.
- Você pode compilar e rodar um arquivo individual usando os comandos **Compilar Arquivo (F9)** e **Executar Arquivo (Shift-F6)** da IDE. Se você usa o comando **Executar Projeto Principal**, a IDE executará o arquivo que a IDE associa como a classe principal do projeto principal. No entanto, se você criar uma classe adicional em seu projeto **HelloWorldApp** e então tentar executar este arquivo com o comando **Executar Projeto Principal**, a IDE vai rodar o arquivo **HelloWorldApp** em seu lugar.
- Você pode querer criar um projeto separado na IDE para exemplificar aplicações que incluem mais de um arquivo fonte.
- Se você estiver teclando na IDE, um box de complementação de código aparecerá periodicamente. Você pode simplesmente ignorar o box de complementação de código e continuar digitando, ou você pode selecionar uma das expressões sugeridas. Se você preferir não ter o box de complementação de código aparecendo automaticamente, você pode desativar essa característica. Escolha **Ferramentas | Opções**, clique na aba **Editor**, e limpe o checkbox **Janela Pop-Up de Auto-Completar Automática**.
- Se você tentar renomear o nó de um arquivo fonte na janela **Projetos**, a IDE abre para você a caixa de diálogo **Renomear** para conduzir você nas opções de renomeamento da classe e atualização do código ao qual essa classe se refere. Clique **Próximo** para mostrar a janela **Refatorar**, que contém uma árvore com as mudanças a serem feitas. Então clique em **Refatorar** para aplicar as alterações. Esta seqüência de cliques mostra-se desnecessária se você tiver somente uma classe simples em seu projeto, mas é muito útil quando suas mudanças afetam outras partes de seu código em projetos grandes.
- Para mais informações e um guia completo da IDE NetBeans, veja o **NetBeans IDE Docs and Support page** ou explore a documentação disponível no menu Help da IDE.

//=====

“Hello World!” para Microsoft Windows.

As seguintes instruções são para usuários do Windows XP Professional, Windows XP Home, Windows Server 2003, Windows 2000 Professional e Windows Vista.

Um Checklist.

Para escrever seu primeiro programa você precisará:

1. O Java SE Development Kit 6 (JDK 6)

- Você pode fazer o download em <http://java.sun.com/javase/downloads/index.jsp> . As instruções para instalação estão em <http://java.sun.com/javase/6/webnotes/install/index.html>

2. Um editor de textos.

- Neste exemplo, pode poderá usar o **Notepad**, um simples editor de textos incluído nas plataformas Windows. Você pode adaptar facilmente estas instruções se você usa uma editor de textos diferente.

Criando sua Primeira Aplicação.

Sua primeira aplicação, **HelloWorldApp**, simplesmente mostrará a saudação “Hello World!”. Para criar este programa, você precisará:

- **Criar um arquivo fonte:** Um arquivo fonte contém um código, escrito em linguagem de programação Java, que você e outros programadores podem compreender. Você pode usar qualquer editor de textos para criar e editar códigos fonte.
- **Compilar o arquivo fonte em um arquivo .class:** O compilador da linguagem de programação Java (**javac**) pega seu arquivo fonte e traduz este texto em instruções que a Java virtual machine pode compreender. Estas instruções contidas dentro do arquivo são conhecidas como **bytecodes**.
- **Rodar o programa:** A ferramenta **launcher tool (java)** usa a Java virtual machine para rodar sua aplicação.

Criando um Arquivo Fonte.

Para criar um arquivo fonte, você tem duas opções:

- Você pode salvar o arquivo **HelloWorldApp.java** em seu computador e evitar uma grande quantidade de digitação. Então, você pode ir para a linha de comando, para compilar o arquivo fonte em um arquivo **.class**, o que será explicado nas próximas páginas;
- Ou, você pode usar as seguintes instruções:

Primeiro, inicie seu editor. Você pode iniciar o editor **Notepad** do menu **Iniciar**, selecionando **Programas > Acessórios > Notepad**. Em um novo documento, digite o seguinte código:

```
/**
 * A classe HelloWorldApp implementa uma aplicação que
 * simplesmente mostra "Hello World!" na saída padrão.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Mostra a string.
    }
}
```

Salve o código em um arquivo com o nome `HelloWorldApp.java`. Para fazer isto no **Notepad**, primeiro escolha **Arquivo > Salvar** como no item de menu. Então, na caixa de diálogo **Salvar como**:

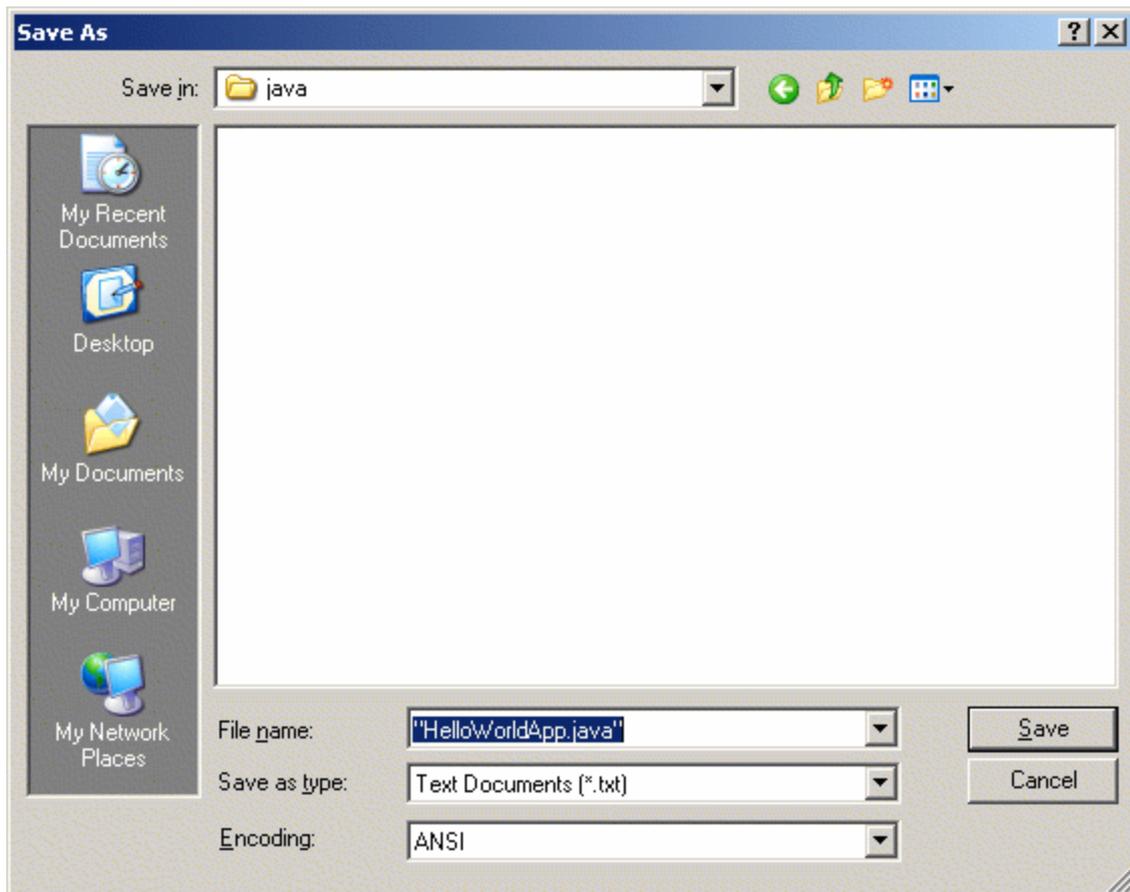
1. Usando a caixa de diálogo **Salvar como**, especifique o diretório onde você quer salvar seu arquivo. Neste exemplo, o diretório é `java` no drive C.

2. No campo **Nome do Arquivo**, digite `"HelloWorldApp.java"`, incluindo as aspas duplas.

3. No combo box **Salvar com o tipo**, escolha **Documentos de Texto (*.txt)**.

4. No combo box **Codificação**, marque a codificação como **ANSI**.

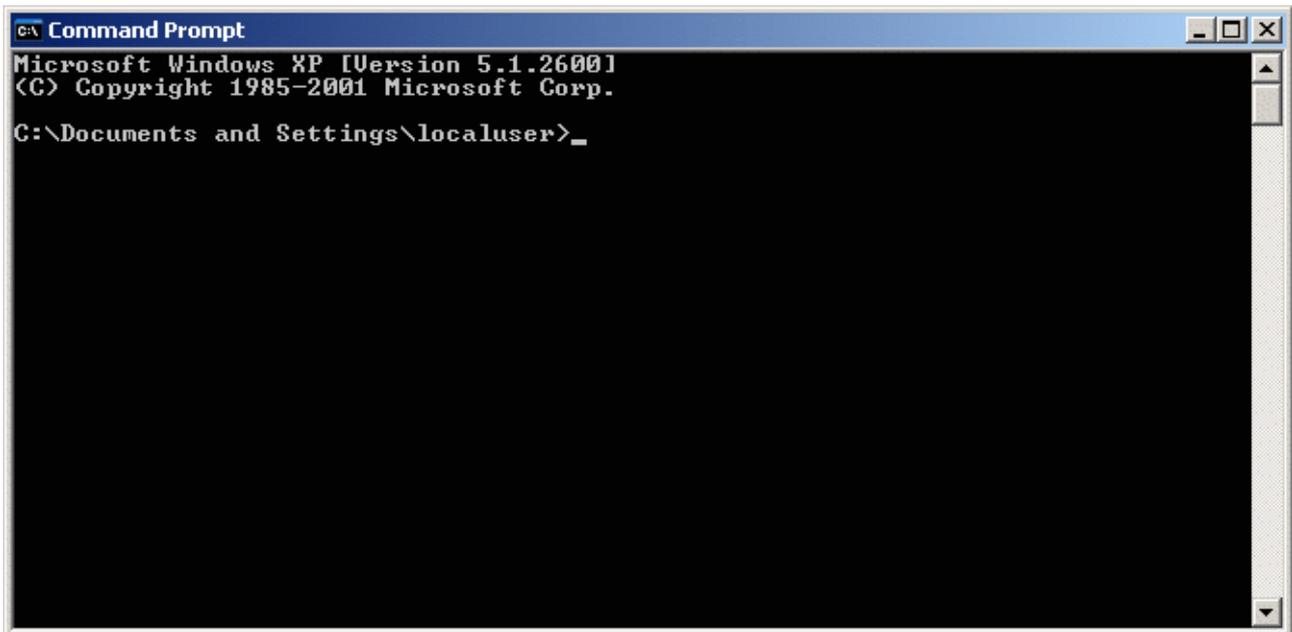
Quando você terminar, a caixa de diálogo mostrará algo como isto:



Agora clique em **Salvar**, e saia do Notepad.

Compilando o Arquivo Fonte em um Arquivo `.class`.

Abra um *shell*, ou janela “*command*”. Você pode fazer isso do menu **Iniciar** escolhendo **Prompt de Comando** (Windows XP), ou escolhendo **Run...** e então digitando `cmd`. A janela shell que aparece é semelhante à seguinte figura:



```
C:\ Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\localuser>_
```

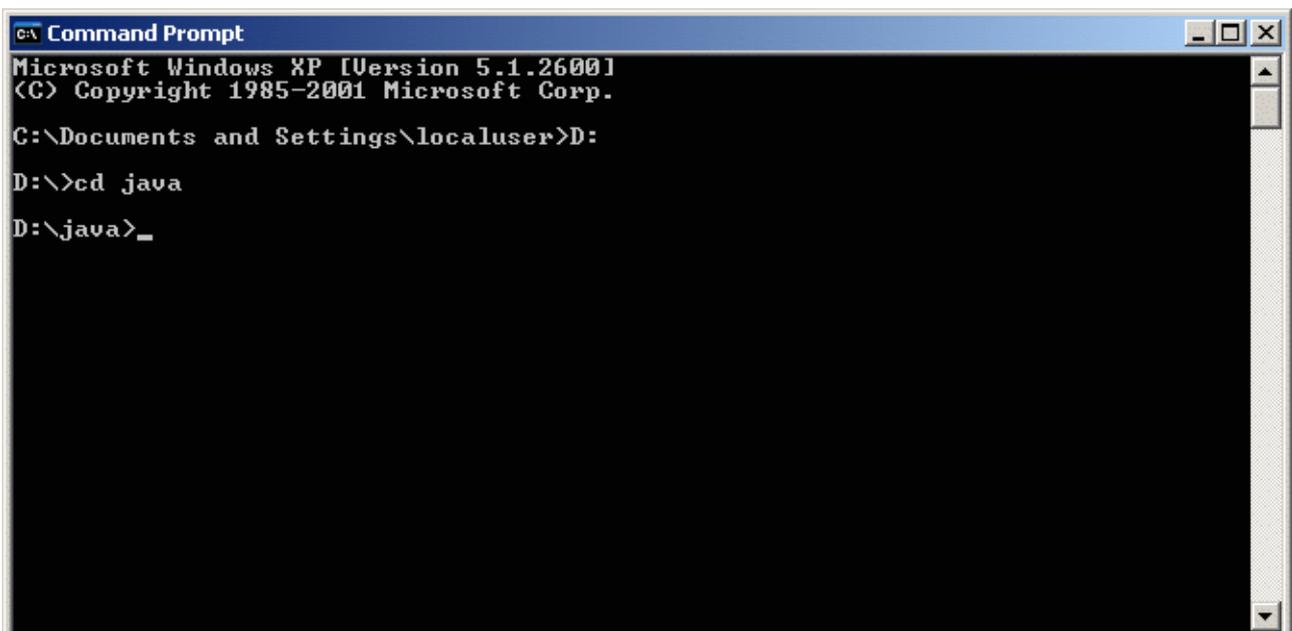
O *prompt* mostra o diretório corrente. Quando você abre o *prompt*, seu diretório corrente é geralmente seu diretório home para o Windows XP (como mostra a figura).

Para compilar seu arquivo fonte, mude o diretório corrente para o diretório onde seu arquivo está localizado. Por exemplo, se seu diretório fonte é `java` no drive `C`, digite os seguintes comandos no *prompt* e pressione **Enter**:

```
cd C:\java
```

Agora o *prompt* se mostrará mudado para `C:\java>`.

Nota: Para mudar para um diretório em um drive diferente, você precisa digitar um comando extra: o nome do drive. Por exemplo, para escolher o diretório `java` no drive, você precisa digitar `D:` e dar um **Enter**, como mostra a figura:



```
C:\ Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\localuser>D:
D:\>cd java
D:\java>_
```

Se você digitar `dir` no *prompt* e então der um **Enter**, você poderá ver seus arquivos fonte, como na

seguinte figura:

```

C:\ Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\localuser>cd C:\java

C:\java>dir
Volume in drive C has no label.
Volume Serial Number is 242E-E457

Directory of C:\java

11/20/2005  08:43 PM    <DIR>          .
11/20/2005  08:43 PM    <DIR>          ..
11/20/2005  08:43 PM                284 HelloWorldApp.java
                1 File(s)                284 bytes
                2 Dir(s)    1,918,476,288 bytes free

C:\java>_

```

Agora você está pronto para compilar. No **prompt**, digite o seguinte comando e pressione **Enter**:

```
javac HelloWorldApp.java
```

O compilador irá gerar um arquivo **bytecode**, **HelloWorldApp.class**. No **prompt**, digite **dir** para ver o novo arquivo que foi gerado, como mostra a seguinte figura:

```

C:\ Command Prompt

C:\java>dir
Volume in drive C has no label.
Volume Serial Number is 242E-E457

Directory of C:\java

11/21/2005  12:36 PM    <DIR>          .
11/21/2005  12:36 PM    <DIR>          ..
11/21/2005  12:36 PM                432 HelloWorldApp.class
11/20/2005  08:43 PM                284 HelloWorldApp.java
                2 File(s)                716 bytes
                2 Dir(s)    1,479,315,456 bytes free

C:\java>

```

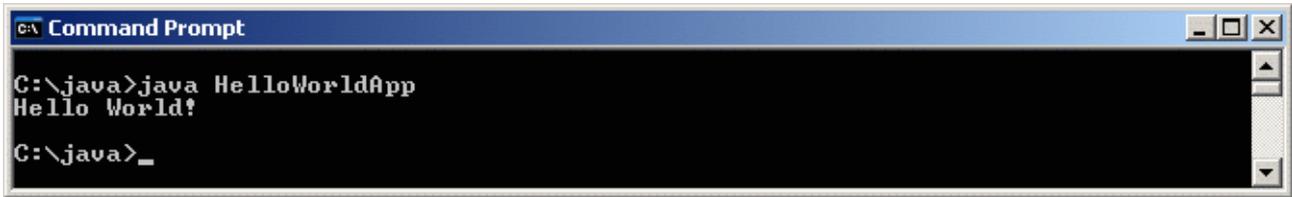
Agora que você tem um arquivo **.class**, você pode Executar seu programa.

Executando o Programa.

No mesmo diretório, digite o seguinte comando no *prompt*:

```
java HelloWorldApp
```

A próxima figura mostra o que você poderá ver agora:

A screenshot of a Windows Command Prompt window. The title bar reads "C:\ Command Prompt". The command prompt shows the following text:

```
C:\java>java HelloWorldApp  
Hello World!  
C:\java>_
```

Congratulações! Seu programa está funcionando!

//=====

“Hello World!” para Solaris OS e Linux.

Estas instruções detalhadas são para usuários do Solaris OS e Linux.

Um Checklist.

Para escrever seu primeiro programa você vai precisar:

1. O Java SE Development Kit 6 (JDK 6)

- Você pode fazer o download em <http://java.sun.com/javase/downloads/index.jsp> (certifique-se de fazer o download do **JDK** e não do **JRE**. Você pode consultar as instruções para instalação em <http://java.sun.com/javase/6/webnotes/install/index.html>

2. Um editor de textos

- Neste exemplo, nós usaremos o **Pico**, um editor disponível para muitas plataformas baseadas em UNIX. Você pode facilmente adaptar estas instruções se você usa um editor de textos diferente, como o **vi** ou **emacs**.

Esses dois itens são tudo o que você precisa para escrever sua primeira aplicação.

Criando sua Primeira Aplicação.

Sua primeira aplicação, **HelloWorldApp**, simplesmente mostrará a saudação “Hello World!”. Para criar este programa, você precisará:

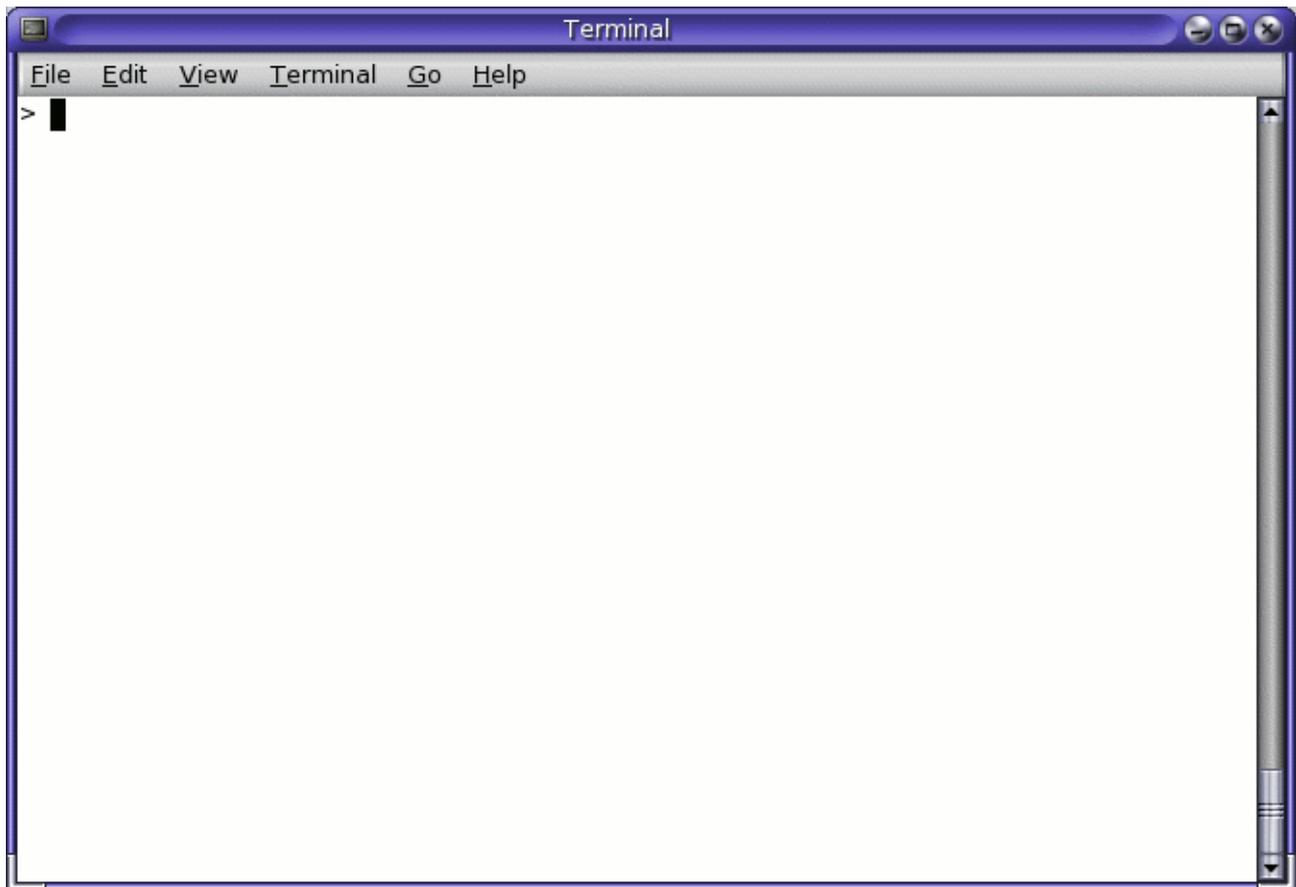
- **Criar um arquivo fonte:** Um arquivo fonte contém um código, escrito em linguagem de programação Java, que você e outros programadores podem compreender. Você pode usar qualquer editor de textos para criar e editar códigos fonte.
- **Compilar o arquivo fonte em um arquivo `.class`:** O compilador da linguagem de programação Java (**javac**) pega seu arquivo fonte e traduz este texto em instruções que a **Java Virtual Machine** pode compreender. Estas instruções contidas dentro do arquivo `.class` são conhecidas como **bytecodes**.
- **Rodar o programa:** A ferramenta **launcher tool** (**java**) usa a **Java Virtual Machine** para rodar sua aplicação.

Criando o Arquivo Fonte.

Para criar um arquivo fonte, você tem duas opções:

- Você pode salvar o arquivo `HelloWorldApp.java` em seu computador e evitar uma grande quantidade de digitação. Então, você pode ir para a linha de comando, para compilar o arquivo fonte em um arquivo `.class`, o que será explicado nas próximas páginas;
- Ou, você pode usar as seguintes instruções:

Primeiro, abra um *shell*, ou janela “*terminal*”:



Quando você abrir o *prompt*, seu diretório corrente geralmente será seu diretório *home*. Você pode mudar seu diretório corrente para seu diretório corrente a qualquer momento digitando `cd` no *prompt* e pressionando **Return**.

Os arquivos fonte que você criar deverão ser mantidos em um diretório separado. Você pode criar um diretório usando o comando `mkdir`. Por exemplo, para criar o diretório `java` em seu diretório *home*, use os seguintes comandos:

```
cd
mkdir java
```

Para mudar seu diretório corrente para o novo diretório, você precisa digitar:

```
cd java
```

Agora você pode dar início à criação de seu arquivo fonte.

Inicie o editor **Pico** digitando `pico` no prompt e pressionando **Return**. Se seu sistema responde com a mensagem `pico : command not found`, então o **Pico** está atualmente desabilitado. Consulte seu administrador do sistema para mais informações, ou use outro editor.

Quando você iniciar o **Pico**, ele mostrará em *buffer* novo e em branco. Esta é a área em que você digitará seu código.

Digite os seguintes comandos no novo *buffer*:

```
/**
 *
 * A classe HelloWorldApp implementa uma aplicação que
 * simplesmente mostra "HelloWorld!" na saída padrão.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Mostra a string.
    }
}
```

Salve o código em um arquivo com o nome de `HelloWorldApp.java`. No editor **Pico**, você faz isso digitando **Ctrl-O** e então, abaixo onde você vê o *prompt* `File Name to write:`, digitando o diretório em que você quer criar o arquivo, seguido por `HelloWorldApp.java`. Por exemplo, se você quer salvar `HelloWorldApp.java` no diretório `/home/jdoe/java`, então você deve digitar `/home/jdoe/java/HelloWorldApp.java` e pressionar **Return**.

Você pode digitar **Ctrl-X** para sair do **Pico**.

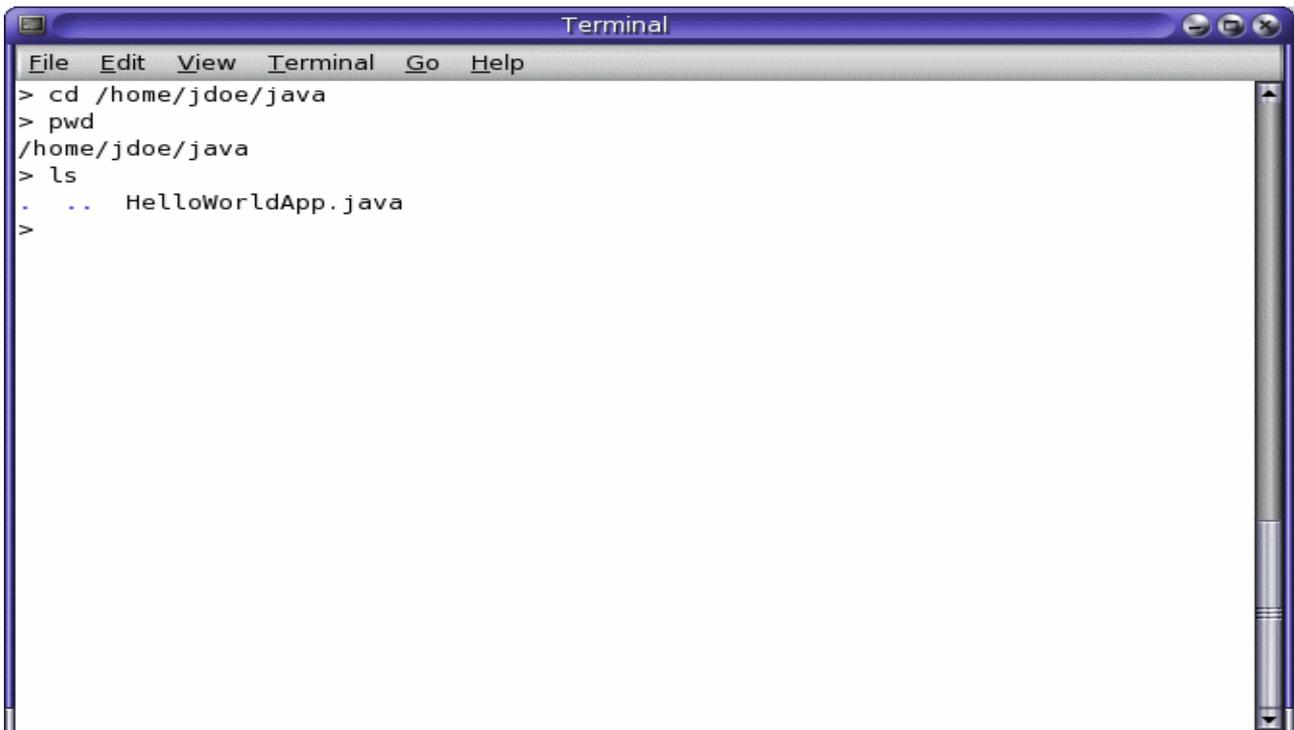
Compilando o Arquivo Fonte em um Arquivo `.class`.

Abra sua janela *shell*. Para compilar seu arquivo fonte, mude seu diretório corrente para o diretório no qual seu arquivo está localizado. Por exemplo, se seu diretório fonte é `/home/jdoe/java`, digite o seguinte comando no *prompt* e pressione **Return**:

```
cd /home/jdoe/java
```

Se você digitar `pwd` no *prompt*, você verá o diretório corrente, o qual no exemplo foi mudado para `/home/jdoe/java`.

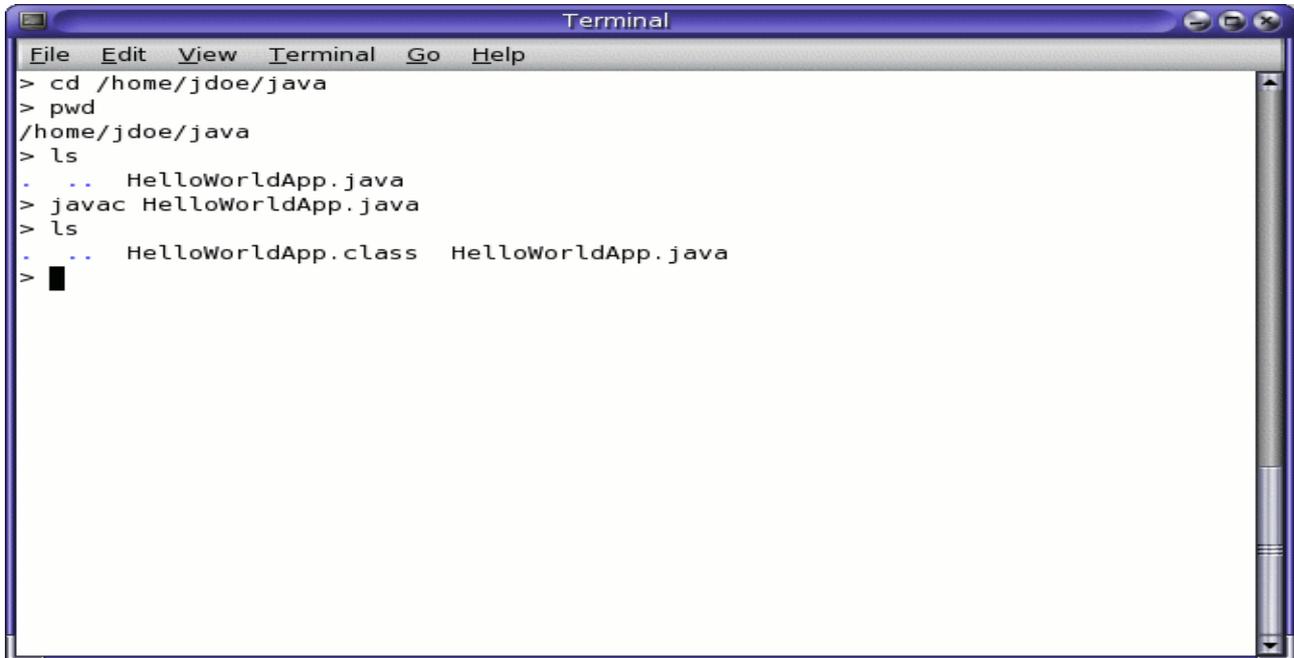
Se você digitar `ls` no *prompt*, você poderá ver seus arquivos.



Agora você está pronto para compilar o arquivo fonte. No **prompt**, digite o seguinte comando e pressione **Return**.

```
javac HelloWorldApp.java
```

O compilador vai gerar o arquivo **bytecode**, `HelloWorldApp.class`. No prompt, digite `ls` para ver o novo arquivo que foi gerado como na seguinte figura:



```
Terminal
File Edit View Terminal Go Help
> cd /home/jdoe/java
> pwd
/home/jdoe/java
> ls
.  ..  HelloWorldApp.java
> javac HelloWorldApp.java
> ls
.  ..  HelloWorldApp.class  HelloWorldApp.java
>
```

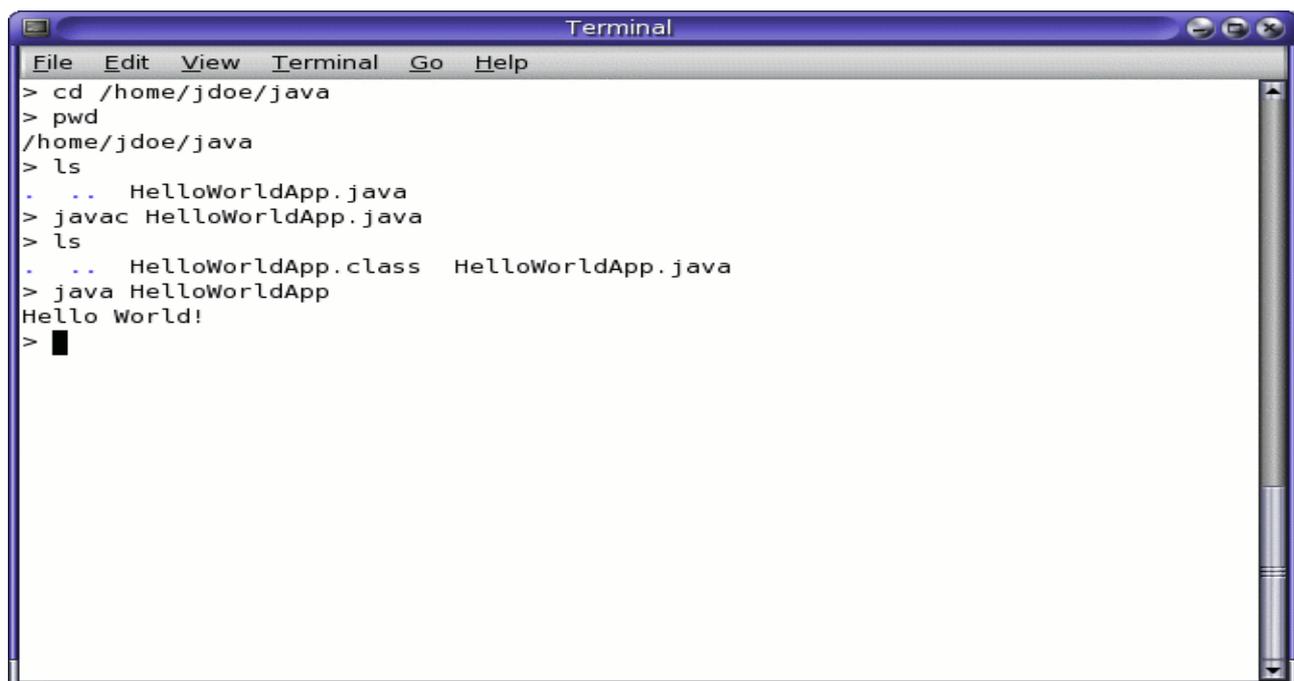
Agora que você tem um arquivo `.class`, pode rodar o programa.

Executando o Programa.

No mesmo diretório, digite no **prompt**:

```
java HelloWorldApp
```

A próxima figura mostra o que você verá em seguida:



```
Terminal
File Edit View Terminal Go Help
> cd /home/jdoe/java
> pwd
/home/jdoe/java
> ls
.  ..  HelloWorldApp.java
> javac HelloWorldApp.java
> ls
.  ..  HelloWorldApp.class  HelloWorldApp.java
> java HelloWorldApp
Hello World!
>
```

Parabéns! Seu programa funciona!

Uma Análise da Aplicação “Hello World!”.

Agora nós veremos como a aplicação “Hello World!” trabalha e analisar o código:

```
/**
 *
 * A classe HelloWorldApp implementa uma aplicação que
 * simplesmente mostra "Hello World!" na saída padrão.
 */

class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Mostra a string.
    }
}
```

A aplicação “Hello World!” consiste de três componentes primários: comentários de arquivo fonte, a definição da classe `HelloWorldApp`, e o método `main`. A seguinte explicação fornecerá a você uma compreensão básica do código, mas o entendimento profundo só virá depois que você terminar de ler o resto do tutorial.

Comentários de Arquivo Fonte.

O seguinte texto define os comentários da aplicação “Hello World!”:

```
/**
 *
 * A classe HelloWorldApp implementa uma aplicação que
 * simplesmente mostra "Hello World!" na saída padrão.
 */
```

Comentários são ignorados pelo compilador mas são úteis para outros programadores. A linguagem de programação Java suporta três tipos de comentários:

```
/* text */
    O compilador ignora qualquer coisa de /* até */.
```

```
/** documentation */
    Isto indica um comentário de documentação (doc comment). O compilador ignora este tipo de comentário, assim como ignora comentários que usam /* e */. A ferramenta javadoc usa comentários doc quando prepara automaticamente a geração de documentação.
```

```
// text
    O compilador ignora qualquer coisa desde // até o fim da linha.
```

A Definição da Classe `HelloWorldApp`.

O seguinte texto inicia o bloco de definição da classe para a aplicação “Hello World!”:

```
class HelloWorldApp {
```

Como mostrado acima, a forma mais básica de definição de uma classe é:

```
class name {
    . . .
}
```

A palavra reservada `class` inicia a definição da classe para uma classe nomeada `name`, e o código para cada classe aparece entre a abertura e o fechamento de chaves.

O Método `main`.

O seguinte método inicia a definição do método `main`:

```
public static void main(String[] args) {
```

Os modificadores `public` e `static` podem ser escritos em outra ordem (`public static` ou `static public`), mas a convenção é o uso de `public static` como mostrado acima. Você pode dar ao argumento o nome que quiser, mas muitos programadores escolhem “`args`” ou “`argv`”.

O método `main` é similar à função `main` em C e C++; ele é o ponto de entrada para sua aplicação e ele subseqüentemente invocará todos os outros métodos requeridos pelo seu programa.

O método `main` é o mecanismo direto no qual o sistema de *runtime* passa informações para sua aplicação. Cada *string* no *array* é chamado um **argumento de linha de comando**. Argumentos de linha de comando permitem aos usuários mudar as operações da aplicação sem recompilá-la. Por exemplo, um programa de ordenação permite ao usuário especificar que o dado seja ordenado em ordem decrescente com o argumento de linha de comando:

```
-descending
```

A aplicação “Hello World!” ignora estes argumentos de linha de comando, mas você deveria ser cauteloso com argumentos que existam.

Finalmente, a linha:

```
System.out.println("Hello World!");
```

usa a classe `System` da biblioteca de núcleo para mostrar a mensagem “Hello World!” na saída padrão.

Problemas no Compilador.

Mensagens de Erro Comuns no Sistema Microsoft Windows.

`'javac' is not recognized as an internal or external command, operable program or batch file`

Se você receber esse erro, o Windows não encontrou o compilador (`javac`).

Aqui está um caminho para dizer ao Windows onde encontrar `javac`. Supondo que você instalou o **JDK** em `C:\jdk6`. No *prompt* você deve digitar o seguinte comando e pressionar **Enter**:

```
C:\jdk6\bin\javac HelloWorldApp.java
```

Se você escolheu esta opção, você terá que preceder seus comandos `javac` e `java` com `C:\jdk6\bin` cada vez que você compilar ou executar um programa. Para evitar esta digitação extra, consulte a seção **Update the PATH variable** nas instruções de instalação do **JDK 6** em: <http://java.sun.com/javase/6/webnotes/install/jdk/install-windows.html#Environment>.

`Class names, 'HelloWorldApp', are only accept if annotation processing is explicitly requested`

Se você receber esse erro, você esqueceu de incluir o sufixo `.java` quando compilou o programa. Relembre, o comando é `javac HelloWorldApp.java` e não `javac HelloWorldApp`.

Mensagens de Erros Comuns nos Sistemas UNIX.

`javac: Command not found`

Se você receber este erro, UNIX não conseguiu encontrar o compilador `javac`.

Este é um caminho para dizer ao UNIX onde encontrar `javac`. Supondo que você instalou o **JDK** em `/usr/local/jdk6`. No *prompt* você deve digitar o seguinte comando e pressionar **Return**:

```
/usr/local/jak6/javac HelloWorldApp.java
```

Nota: Se você escolher esta opção, cada vez que você compilar ou executar um programa, você terá que preceder seus comandos `javac` e `java` com `/usr/local/jak6/`. Para evitar essa digitação extra, você deveria adicionar esta informação em sua variável **PATH**. Os caminhos para fazer isso variam dependendo em qual shell você está executando atualmente.

`Class names, 'HelloWorldApp', are only accept if annotation processing is explicitly requested`

Se você receber este erro, você esqueceu de incluir o sufixo `.java` quando compilou seu programa. Relembre, o comando é `javac HelloWorldApp.java` e não `javac HelloWorldApp`.

Erros de Sintaxe (Todas as Plataformas).

Se você digitou incorretamente uma parte de um programa, o compilador pode emitir um erro de sintaxe. A mensagem geralmente mostra o tipo de erro, o número da linha onde o erro foi detectado, o código naquela linha, e a posição do erro dentro do código. Aqui está um erro causado pela omissão do ponto-e-vírgula (;) no fim da declaração:

```
testing.java:14: ';' expected.
System.out.println("Input has " + count + " chars.")
                ^
1 error
```

Algumas vezes o compilador não descobre sua intenção e mostra uma mensagem de erro confusa ou múltiplas mensagens de erro se o erro desenrolar-se por várias linhas. Por exemplo, o seguinte código omite um ponto-e-vírgula (;) na linha "count++":

```
while (System.in.read() != -1)
    count++
System.out.println("Input has " + count + " chars.");
```

Quando estiver processando este código, o compilador emite duas mensagens de erro:

```
testing.java:13: Invalid type expression.
    count++
    ^
testing.java:14: Invalid declaration.
    System.out.println("Input has " + count + " chars.");
    ^
2 errors
```

O compilador emite duas mensagens de erro porque depois de processar `count++`, o estado do compilador indica que ele está no meio de uma expressão. Sem o ponto-e-vírgula, o compilador não tem como saber que a declaração está completa.

Se você ver qualquer erro no compilador, então seu programa não é compilado com sucesso, e o compilador não cria um arquivo `.class`. Verifique cuidadosamente o programa, conserte quaisquer erros que você detectar, e tente novamente.

Erros de Semântica.

Em adição à verificação se seu programa está sintaticamente correto, o compilador checa por outras incorreções. Por exemplo, o compilador adverte você cada vez que você usar uma variável que não foi inicializada:

```
testing.java:13: Variable count may not have been initialized.
    count++
    ^
testing.java:14: Variable count may not have been initialized.
    System.out.println("Input has " + count + "chars.");
    ^
2 errors
```

Mais uma vez, seu programa não foi compilado com sucesso, e o compilador não criou o arquivo `.class`. Corrija o erro e tente novamente.

Problemas de Runtime.

Mensagens de Erro no Sistema Operacional Windows.

```
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp
```

Se você receber esse erro, `java` não encontrou seu arquivo bytecode, `HelloWorldApp.class`.

Um dos lugares `java` prováveis para encontrar seu arquivo `.class` é seu diretório corrente. Então se seu arquivo `.class` está em `C:\java`, você poderia mudar seu diretório corrente para lá. Para mudar seu diretório, digite o seguinte comando no **prompt** e pressione **Enter**:

```
cd c:\java
```

O prompt será modificado para `C:\java>`. Se você digitar `dir` no **prompt**, você poderá ver seus arquivos `.java` e `.class`. Agora digite `java HelloWorldApp` novamente.

Se continuar tendo problemas, você poderia mudar sua variável `CLASSPATH`. Para ver se isto é necessário, tente acessar o classpath com o seguinte comando:

```
set CLASSPATH=
```

Agora digite `java HelloWorldApp` novamente. Se seu programa rodar agora, você terá que mudar sua variável `CLASSPATH`. Para acessar essa variável, consulte a seção Update the `PATH` variable nas instruções de instalação do **JDK 6** em <http://java.sun.com/javase/6/webnotes/install/jdk/install-windows.html#Environment>.

```
Exception in thread "main" java.lang.NoClassDefFoundError:
HelloWorldApp/class
```

Um engano comum cometido por programadores iniciantes é tentar rodar o lançador `java` no arquivo `.class` que foi criado pelo compilador. Por exemplo, você receberá esse erro se tentar executar seu programa com `java HelloWorldApp.class` ao invés de `java HelloWorldApp`. Relembre, o argumento é o nome da classe que você quer usar, não o nome do arquivo.

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

A **Java virtual machine** requer que a classe que você executa com ela tenha um método `main` para que possa iniciar a execução de sua aplicação.

Mensagens de Erro em Sistemas UNIX.

```
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp
```

Se você receber este erro, `java` não pode encontrar seu arquivo `bytecode`, `HelloWorldApp.class`.

Um dos lugares `java` prováveis para encontrar seu arquivo `bytecode` é seu diretório corrente. Então, por exemplo, se seu arquivo `bytecode` está em `/home/jdoe/java`, você poderia mudar seu diretório corrente para lá. Para mudar seu diretório, digite o seguinte comando no `prompt` e pressione **Return**:

```
cd /home/jdoe/java
```

Se você digitar `pwd` no `prompt`, você deveria ver `/home/jdoe/java`. Se você digitar `ls` no `prompt`, você poderia ver seus arquivos `.java` e `.class`. Agora entre `java HelloWorldApp` novamente.

Se você continuar tendo problemas, você pode ter que mudar sua variável de ambiente **CLASSPATH**. Para ver se isso é necessário, tente acessar o **CLASSPATH** com o seguinte comando:

```
unset CLASSPATH
```

Agora digite `java HelloWorldApp` novamente. Se seu programa rodar agora, você terá que mudar sua variável **CLASSPATH** da mesma maneira que a variável **PATH** acima.

```
Exception in thread "main" java.lang.NoClassDefError: HelloWorldApp/class
```

Um engano comum cometido por programadores iniciantes é tentar rodar o lançador `java` no arquivo `.class` que foi criado pelo compilador. Por exemplo, você receberá esse erro se tentar executar seu programa com `java HelloWorldApp.class` ao invés de `java HelloWorldApp`. Relembre, o argumento é o nome da classe que você quer usar, não o nome do arquivo.

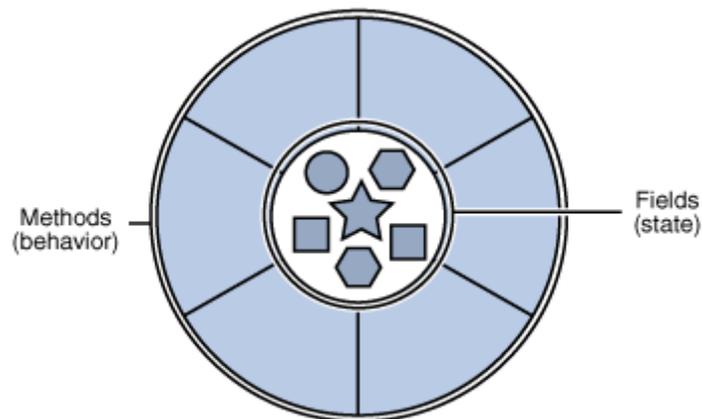
```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

A **Java virtual machine** requer que a classe que você executa com ela tenha um método `main` para que possa iniciar a execução de sua aplicação.

O que é um objeto?

O conceito de objetos é fundamental para compreender a tecnologia de orientação a objeto. Olhe à sua volta agora e você encontrará muitos exemplos de objetos do mundo real: seu cachorro, sua mesa, sua televisão, sua bicicleta.

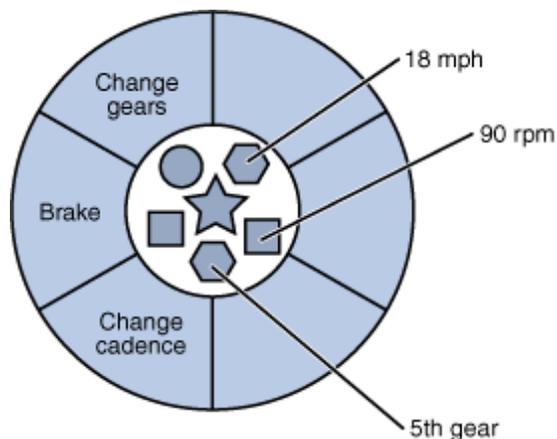
Os objetos do mundo real compartilham duas características: todos eles têm um **estado** e um **comportamento**. Cachorros têm um estado (nome, cor, raça) e comportamento (forma de latir, encantador, abanar o rabo). Bicicletas também têm estado (marcha corrente, cadência corrente do pedal, velocidade corrente) e comportamento (mudança de marcha, mudança de cadência do pedal, aplicação de marchas). Identificar o estado e o comportamento dos objetos do mundo real é um ótimo caminho para pensar em termos de programação orientada a objetos.



Pare um minuto agora para observar os objetos do mundo real que estão à sua volta. Para esses objetos que você vê, faça a si mesmo duas perguntas: "Em que possíveis estados esses objetos podem estar?" e "Quais os possíveis comportamentos esses objetos podem apresentar?". Escreva suas observações. Como você pode ver, os objetos do mundo real variam em complexidade; seu monitor pode ter dois estados (ligado ou desligado) e dois possíveis comportamentos (de frente e virado), mas seu rádio pode ter estados adicionais (de frente, virado, volume atual, estação atual) e comportamento (ligado, desligado, aumento de volume, procura de estação, estação corrente, música). Você vai reparar que alguns objetos, por sua vez, podem conter outros objetos. Essas observações do mundo real exprimem em outras palavras a programação orientada a objetos.

Objetos de software têm conceito similar a objetos do mundo real: eles também consistem de estado e comportamento associado. Um objeto armazena este estado em campos (variáveis em algumas linguagens de programação) e evidencia seu comportamento através de métodos (funções em algumas linguagens de programação). **Métodos** operam em um estado interno do objeto e servem como o mecanismo primário para a comunicação de objeto para objeto. Escondendo o estado interno e requerindo total interação para ser executado sem interrupção, um método de objeto é conhecido como **dado encapsulado** - um princípio fundamental da programação orientada a objetos.

Considere uma bicicleta, por exemplo:



Para atribuição do estado (velocidade corrente, cadência corrente do pedal, e marcha corrente) e provendo métodos para mudanças de estado, permanecem do objeto em controle a forma de como o mundo exterior é autorizado para usá-lo. Por exemplo, se a bicicleta só tem seis marchas, um método para mudar as marchas poderia rejeitar qualquer valor menor que 1 e maior que 6.

Empacotando códigos em objetos individuais pode-se obter vários benefícios, incluindo:

1. Modularidade: o código fonte para um objeto pode ser escrito e feita sua manutenção independentemente do código fonte de outros objetos. Depois de criado, um objeto pode ser facilmente distribuído em partes do sistema.

2. Esconder informações: Ao interagir somente com um método do objeto, os detalhes da implementação interna permanecem escondidos do mundo exterior.

3. Código reutilizável: Se um objeto realmente existe (mesmo escrito por outro desenvolvedor de software), você pode usar aquele objeto em seu programa.

4. Pugabilidade e facilidade de debug: Se um objeto em particular torna-se problemático, você pode simplesmente removê-lo de sua aplicação e inserir um objeto diferente para substituí-lo. Esta é uma analogia com consertos de problemas mecânicos do mundo real. Se uma peça quebra, você troca ela, não a máquina inteira.

//=====

O que é uma classe?

No mundo real, você encontra muitas vezes vários objetos todos do mesmo tipo. Pode haver milhares de outras bicicletas em existência, todas com a mesma marca e modelo. Cada bicicleta foi construída do mesmo conjunto e projeto e por esta razão contém os mesmos componentes. Em termos de orientação a objetos, nós dizemos que sua bicicleta está em uma *instância* da classe de objetos conhecida como bicicleta. A classe é o projeto do qual o objeto individual criado.

A seguinte classe `Bicycle` é uma possível implementação de bicicleta:

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" + cadence + " speed:" + speed + "gear:" + gear);
    }
}
```

A sintaxe da linguagem de programação Java pode parecer nova para você, mas o desenho da classe é baseado na discussão prévia do objeto bicicleta. Nos campos `cadence`, `speed`, e `gear` representam o **estado** do objeto, e o **método** (`changeCadence`, `changeGear`, `speedUp`, etc) define esta interação com o mundo exterior.

Você deve observar que a classe `Bicycle` não contém o método `main`. Isto porque esta não é uma aplicação completa; isto é só o projeto para bicicletas que será usado na aplicação. A responsabilidade de criação e uso da novo objeto `Bicycle` pertence a outra classe em nossa aplicação.

Aqui está uma classe `BicycleDemo` que cria dois objetos `Bicycle` separados e invoca seus métodos:

```
class BicycleDemo {
    public static void main(String[] args) {

        // Cria dois objetos Bicycle diferentes
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoca os métodos desses objetos
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

A saída deste teste mostra ao final cadência, velocidade, e marcha para as duas bicicletas:

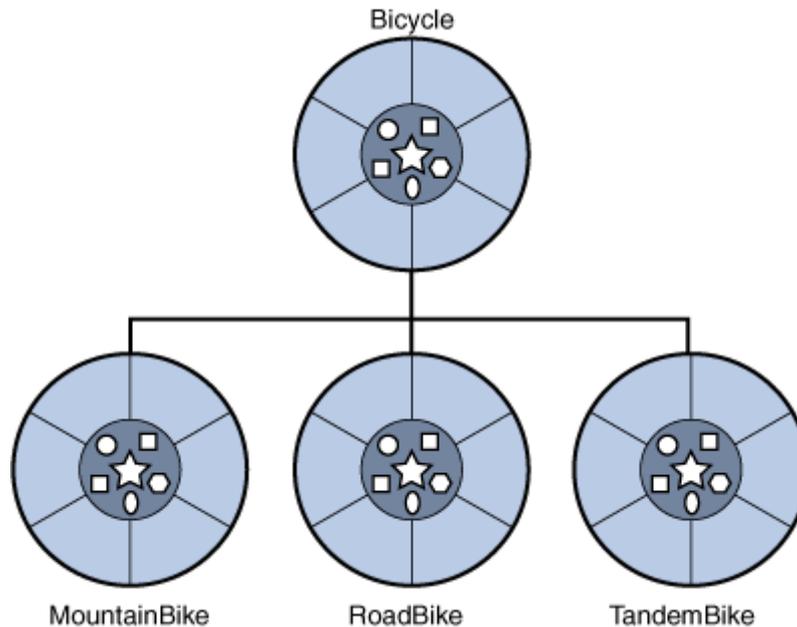
```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

```
//=====
```

O que é uma herança?

Diferentes tipos de objetos podem ter algumas características em comum com outros. **Mountain bikes**, **road bikes**, e **tandem bikes**, por exemplo, compartilham todas as características de bicicletas (velocidade corrente, cadência corrente do pedal, marcha corrente). Além disso, cada um define aspectos adicionais que fazem delas diferentes: **tandem bicycles** têm dois assentos e dois guidões; **road bikes** têm guidões abaixados; algumas **mountain bikes** têm uma corrente adicional, dando a elas uma diferente proporção de marcha.

Programação orientada a objeto permite classes para herdar estados comumente usados e comportamentos para outras classes. Uma nova bicicleta pode vir com a **superclasse** de **mountain bike**, **road bike**, ou **tandem bike**. Na linguagem de programação Java, cada classe é permitida para ter uma superclasse, e cada superclasse tem o potencial para um número ilimitado de subclasses.



A sintaxe para criação de uma subclasse é simples. Começando com a declaração da classe, é usado a palavra reservada **extends**, seguida pelo nome da classe para herdar de:

```

class MountainBike extends Bicycle {

    // novos campos e métodos definindo uma mountain bike vão aqui

}
  
```

Isso pega de **MountainBike** todos os mesmos campos e métodos como **Bicycle**, este código ainda permite focar exclusivamente na característica que faz dela única. Isto faz com que o código para sua subclasse seja fácil de escrever. No entanto, você deve ter cuidado para documentar o estado e o comportamento que cada superclasse define, desde que o código não apareça no arquivo de cada subclasse.

```
//=====
```

O que é uma Interface?

Como você tem aprendido, **objetos** definem suas interações com o mundo exterior de acordo com os **métodos** que eles exibem. **Métodos** configuram a **interface** do **objeto** com o mundo exterior; os botões na frente de sua televisão, por exemplo, são interfaces entre você e a instalação elétrica no outro lado da caixa plástica do aparelho. Você pressiona o botão "Ligar" para ligá-lo e desligá-lo.

De uma forma geral, uma interface é um grupo de métodos relacionados com corpos vazios. Um comportamento de uma bicicleta, se especificado em uma interface, apresenta aparência como segue:

```
interface Bicycle {

    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

Para implementar esta interface, o nome da classe poderia ser **ACMEBicycle**, por exemplo, e você poderia usar a palavra reservada **implements** na declaração da **classe**:

```
class ACMEBicycle implements Bicycle {

    // lembrança dessa classe implementada como antes
}
```

Interfaces formam um contrato entre a classe e o lado exterior do mundo, e este contrato é forçado em tempo de construção para o compilador. Se sua classe reivindica a implementação de uma interface, todos os métodos definidos pela interface aparecerão no código fonte antes da classe que sucederá a compilação.

Nota: Para atualmente compilar a classe **ACMEBicycle**, você precisaria adicionar a palavra reservada **public** antes de implementar os métodos da interface.

```
//=====
```

O que é um pacote?

Um pacote é um espaço nomeado que organiza o conjunto de classes e interfaces declaradas. Conceitualmente você pode pensar em pacotes como sendo similar a diferentes pastas em seu computador. Você mantém páginas HTML em uma pasta, imagens em outra, e **scripts** ou aplicações em outras. Pelo motivo de softwares escritos em Java poderem ser compostos de centenas de milhares de classes individuais, faz sentido organizar as classes e interfaces em pacotes.

A plataforma Java é provida de uma enorme biblioteca de classes (um conjunto de pacotes) apropriados para uso em suas aplicações. Esta biblioteca é conhecida como a **"Application Programming Interface"**, ou **"API"**. Estes pacotes representam as tarefas mais comumente associadas com o propósito de programação. Por exemplo, um objeto **String** contém estado e comportamento para caracteres **strings**; um objeto **File** permite ao programador facilitar a criação, deleção, inspeção, comparação, ou modificação de um arquivo no sistema de arquivos; um objeto **Socket** permite a criação e uso de **sockets** de rede; vários objetos **GUI** controlam botões e **checkboxes** e qualquer coisa relacionada a interfaces gráficas. Há literalmente milhares de classes para escolher lá. Isto permite ao programador, focar no design de uma aplicação em particular, antes que a infraestrutura requeira fazer este trabalho.

A Especificação **Java Platform API** contém uma lista completa de todos os pacotes, interfaces, classes, campos, e métodos suportados pela **Java Platform 6, Standard Edition**.

Variáveis.

Um objeto armazena seu estado em campos.

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

Quais são as regras e convenções para nomear um campo? Além de `int`, que outros tipos de dados existem? Campos são inicializados quando eles são declarados? Os campos são atribuídos com valor **default** se eles não forem explicitamente inicializados? Na linguagem de programação Java, os termos “campo” e “variável” são ambos usados; Isso é uma fonte comum de confusão entre novos desenvolvedores, desde que ambos parecem referir-se às mesmas coisas.

A linguagem de programação Java define os seguintes tipos de variáveis:

- **Variáveis de Instância (campos não-estáticos)** : Tecnicamente falando, objetos armazenam seus estados individuais em “campos não estáticos”, ou seja, campos declarados sem a palavra reservada `static`. Campos não-estáticos são também conhecidos como **variáveis de instância** porque seus valores são únicos para cada **instância da classe** (como cada objeto, em outras palavras); a `currentSpeed` de uma bicicleta é independente da `currentSpeed` de outra.
- **Variáveis da Classe (campos estáticos)** : Uma **variável da classe** é qualquer campo declarado com o modificador `static`; ele diz ao compilador que há exatamente uma cópia da variável em existência, indiferentemente de quantas vezes a classe está sendo instanciada. Um campo definindo o número de marchas para um tipo particular de bicicleta poderia ser marcado como `static` desde que conceitualmente o mesmo número de marchas será aplicado para todas as instâncias. O código `static int numGears = 6;` seria criado como um campo estático. Adicionalmente, a palavra reservada `final` seria adicionada para indicar que o número de marchas nunca mudaria.
- **Variáveis Locais** : Da mesma forma que um objeto armazena seu estado em campos, um método muitas vezes armazena seu estado temporário em **variáveis locais**. A sintaxe para a declaração de uma variável local é similar à declaração do campo (por exemplo, `int count = 0;`). Não há palavras reservadas especiais para designação de uma variável como local; esta determinação vem completa do local em que a variável é declarada – que é entre a abertura e o fechamento de chaves do método. Dessa maneira, variáveis locais são visíveis somente para os métodos em que eles são declarados; eles não são acessíveis para o resto da classe.
- **Parâmetros** : Você sempre verá exemplos de parâmetros, como na classe `Bicycle` e no método `main` da aplicação “Hello World!”. Recorde que a marca para o método `main` é `public static void main(String[] args)`. Aqui, a variável `args` é o parâmetro do método. É importante lembrar que os parâmetros são sempre classificados como “variáveis” e não “campos”. Isso se aplica para outros construtores de aceitação de parâmetros também (como os construtores e manipuladores de exceções).

É importante lembrar que este tutorial usa as seguintes diretrizes gerais quando discute campos e variáveis. Se você está falando a respeito de “campos em geral” (excluindo variáveis e locais e parâmetros), nós simplesmente dizemos “campos”. Se a discussão se aplica a todos os acima, nós dizemos simplesmente “variáveis”. Você também verá ocasionalmente o termo “membro” usado como fonte. Um tipo de campos, métodos e tipos aninhados são coletivamente chamados de **membros**.

//=====

Nomeando Variáveis.

Todas as linguagens de programação têm seu próprio conjunto de regras e convenções para os tipos de nomes que você pode usar, e a linguagem de programação Java não é diferente. As regras e convenções para você nomear suas variáveis podem ser resumidas como segue:

- **Nomes de variáveis são case-sensitive** : O nome da variável pode ter qualquer identificador válido – um tamanho ilimitado de sequência de letras **Unicode** e dígitos, começando com uma letra, o sinal de dólar (“\$”), ou um underscore (“_”). Por convenção, no entanto, é sempre bom começar os nomes de suas variáveis com uma letra, não um “\$” ou “_”. Adicionalmente, o sinal de dólar, por convenção, nunca é usado em tudo. Você encontrará algumas situações onde nomes auto-gerados conterão um sinal de dólar, mas seus nomes de variáveis deveriam evitar de usá-lo. Uma convenção similar existe para o caracter underscore; embora seja permitido começar seus nomes de variáveis com “_”, esta prática é desencorajada. Espaços em branco não são permitidos.
- **Caracteres subseqüentes podem ser letras, dígitos, sinal de dólar, ou caracterer underscore.** Quando estiver escolhendo um nome para suas variáveis, use palavras completas em vez de abreviações misteriosas. Dessa forma seu código será fácil de ler e entender. Em muitos casos isso também fará de seus códigos documentarem a si mesmos; campos nomeados **cadence**, **speed**, e **gear**, por exemplo, são muito mais intuitivos que versões abreviadas, como um **s**, **c**, e **g**. Também mantenha em mente que o nome que você escolher não pode ser uma **palavra reservada**.
- **Se o nome que você escolher consiste de uma palavra, escreva a palavra toda em letras minúsculas.** Se o nome consiste de mais de uma palavra, coloque a primeira letra da próxima palavra em letra maiúscula. Os nomes **gearRatio** e **currentGear** são exemplos desta convenção. Se suas variáveis armazenam um valor constante, como um **static final int NUM_GEAR = 6**, a convenção muda levemente, tornando maiúscula qualquer letra e separando palavras subseqüentes com um caracter underscore. Por convenção, o caracter underscore nunca é usado em outra parte.

//=====

Tipos de Dados Primitivos.

A linguagem de programação Java é fortemente tipada, o que faz com que todas as variáveis precisem ser declaradas antes que elas possam ser usadas. Isto envolve a declaração do tipo da variável e o nome, como você pode ver:

```
int gear = 1;
```

Fazendo assim é informado ao seu programa que o campo nomeado "gear" existe, atribui ao dado o tipo numérico, e o inicializa com um valor "1". Os tipos de dados das variáveis determinam os valores que elas contêm, determinando as operações que podem ser utilizadas nela. Em adição ao `int`, a linguagem de programação Java suporta sete outros **tipos de dados primitivos**. Um tipo primitivo é pré-definido pela linguagem e é nomeada por um palavra reservada. Valores primitivos não compartilham estados com outros valores primitivos. Os oito tipos de dados primitivos que são suportados pela linguagem de programação Java são:

- **byte**: O tipo de dado `byte` é um tipo 8-bits designado para indicar inteiros. Ele tem um valor mínimo de -128 e valor máximo de 127 (inclusive). O tipo de dado `byte` pode ser proveitoso para economizar memória em **arrays** grandes, onde a memória economizada nesse momento importa. Ele pode ser usado em lugar de `int` aonde seus limites ajudam a clarear seu código; o fato de que o alcance da variável é limitado pode servir como uma forma de documentação.
- **short**: O tipo de dado `short` é um tipo 16-bits designado para indicar inteiros. Ele tem um valor mínimo de -32.768 e um valor máximo de 32.767 (inclusive). Como o tipo `byte`, as mesmas diretrizes se aplicam: você pode usar um `short` para economizar memória em **arrays** grandes, em situações onde a memória economizada nesse momento importa.
- **int**: O tipo de dado `int` é designado para indicar inteiros. Ele tem um valor mínimo de -2.147.483.648 e um valor máximo de 2.147.483.647 (inclusive). Para valores integrais, este tipo de dado é geralmente escolhido como **default** a menos que haja uma razão (como mencionado acima) para escolher alguma coisa diferente. Este tipo de dado provavelmente será maior que os números que seu programa usará, mas se você precisa de um extenso alcance de valores, use `long` em seu lugar.
- **long** : O tipo de dado `long` é designado para indicar inteiros. Ele tem um valor mínimo de -9.223.372.036.854.775.808 e um valor máximo de 9.223.372.036.854.775.807 (inclusive). Use este tipo de dado quando você precisa de um alcance de valores mais extenso que os fornecidos pelo `int`.
- **float** : O tipo de dado `float` é um tipo de ponto flutuante 32-bit IEEE 754. Este alcance de valores está além do escopo dessa discussão. Assim como recomendado para os tipos `byte` e `short`, use um `float` (em lugar de `double`) se você precisa economizar memória em **arrays** grandes de números de ponto flutuante. Este tipo de dado nunca deveria ser usado para valores precisos, assim como para moedas. Para isso, você precisa usar a classe `java.math.BigDecimal`.
- **double** : O tipo de dado `double` é um tipo de dados de dupla precisão 32-bit IEEE 754. Este alcance de valores está fora do escopo dessa discussão. Para valores decimais, este tipo de dado é geralmente a escolha padrão. Como mencionado acima, este tipo de dado nunca deve ser usado para valores precisos, assim como para moedas.
- **boolean** : O tipo de dado `boolean` tem só dois possíveis valores: `true` e `false`. Use este tipo de dado para sinalizar as condições **true/false**. Este tipo de dado representa um bit de informação, mas seu "tamanho" não é precisamente definido.
- **char** : O tipo de dado `char` é um simples caracter **Unicode** 16-bit. Ele tem um valor mínimo de `'\u0000'` (ou 0) e um valor máximo de `'\uffff'` (ou 65.535 inclusive).

Em adição aos oito tipos primitivos de dados listados acima, a linguagem de programação Java também fornece suporte especial para caracteres strings através da classe `java.lang.String`. Envolvendo seus caracteres entre aspas duplas você automaticamente criará um novo objeto `String`; por exemplo, `String s = "This is a string"`; objetos `String` são **imutáveis**, o que significa que quando criados, seus valores não podem ser mudados. A classe `String` não é tecnicamente um tipo de dado primitivo, mas considerando o suporte especial dado pela linguagem, você provavelmente tenderá a pensar desta maneira.

Valores Default de Variáveis.

Nem sempre é necessário atribuir um valor quando o campo é declarado. Campos que são declarados mas não são inicializados serão ajustados para um padrão racional pelo compilador. Genericamente falando, o padrão será zero ou nulo (`null`), dependendo do tipo de dado. Confiar tanto nos valores padrão, no entanto, é geralmente considerado um estilo ruim de programação.

O seguinte mapa resume os valores padrão para os tipos de dados citados:

Tipo de Dado	Valor padrão (para campos)
<code>byte</code>	<code>0</code>
<code>short</code>	<code>0</code>
<code>int</code>	<code>0</code>
<code>long</code>	<code>0L</code>
<code>float</code>	<code>0.0f</code>
<code>double</code>	<code>0.0d</code>
<code>char</code>	<code>'\u0000'</code>
<code>String</code> (ou qualquer objeto)	<code>null</code>
<code>boolean</code>	<code>false</code>

Nota: Variáveis locais são um pouco diferentes; o compilador nunca atribui um valor padrão para uma variável local não inicializada. Se você não pode inicializar sua variável local onde ela está declarada, certifique-se de atribuir um valor antes de tentar usá-la. Acessando uma variável local não inicializada resultará em um erro em tempo de compilação.

//=====

Literais.

A palavra reservada **new** não é usada quando estiver inicializando uma variável de um tipo primitivo. Tipos primitivos são tipos especiais de tipos de dados construídos na linguagem; eles não são objetos criados de uma classe. Um literal é o código fonte de representação de um valor fixo; literais são representados diretamente em seu código sem computação requerida. Como é mostrado abaixo, é possível atribuir um literal para uma variável de um tipo primitivo:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Os tipos integrais (**byte**, **short**, **int**, e **long**) podem ser expressados usando sistema decimal, octal, ou números hexadecimais. Decimal é um sistema de números que você usa todo dia; é baseado em 10 dígitos, numerados de 0 até 9. O sistema de números octal é base 8, consistindo dos dígitos de 0 a 7. O sistema hexadecimal é base 16, onde dígitos são os números de 0 até 9 e as letras A até F. Para o propósito geral de programação, o sistema decimal é apropriado para ser o único sistema numérico que você sempre usará. No entanto, se você precisar do octal ou hexadecimal, o seguinte exemplo mostra a sintaxe correta. O prefixo **0** indica octal, enquanto **0x** indica hexadecimal.

```
int decVal = 26;    // O número 26, em decimal
int octVal = 032;  // O número 26, em octal
int hexVal = 0x1a; // O número 26, em hexadecimal
```

Os tipos de ponto flutuante (**float** e **double**) só podem ser expressados usando E ou e (para notação científica), F ou f (32-bit float literal) e D ou d (64-bit double literal; este é o padrão e por convenção é omitido).

```
double d1 = 123.4;
double d2 = 1.234e2; // mesmo valor que d1, mas em notação científica
float f1 = 123.4f;
```

Literais do tipo **char** e **String** podem conter qualquer caracter **Unicode** (UTF-16). Se seu editor e sistema de arquivos permití-lo, você pode usar qualquer caracter diretamente em seu código. Se não, você pode usar uma "saída **Unicode**" como em `'\u0108'` (letra maiúscula C com circunflexo), ou `"\u00EDse\u00Flor"` (*Si Señor* em espanhol). Sempre use 'aspas simples' para literais **char** e "aspas duplas" para literais **Strings**. Seqüências de saídas Unicode podem ser usadas em outra parte em um programa (como em um campo nomes, por exemplo), não somente em literais **char** ou **String**.

A linguagem de programação Java também suporta algumas saídas especiais de seqüências para literais **char** e **String**: `\b` (backspace), `\t` (tab), `\n` (alimenta linha), `\f` (alimenta formulário), `\r` (retorno de carro), `\"` (aspas duplas), `\'` (aspas simples), e `\\` (barra).

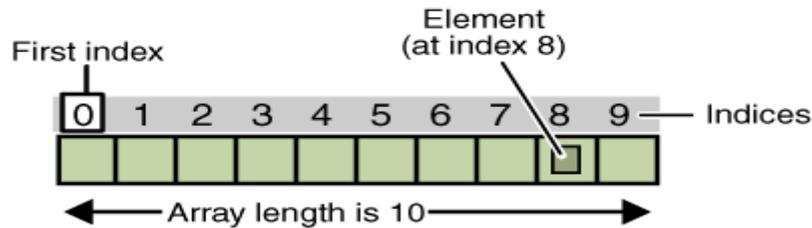
Há também o literal especial **null** que pode ser usado como um valor para referenciar qualquer tipo. O literal **null** pode ser atribuído para qualquer variável, exceto variáveis de tipo primitivo. Há pouco o que você pode fazer com um valor **null** além de fazer testes de presença. No entanto, **null** é frequentemente usado em programas como um marcador para indicar que alguns objetos estão desabilitados.

Finalmente, há também um tipo especial de literal chamado de **classe literal**, formada pela captura de um nome de tipo e acrescentando `".class"`, por exemplo, `String.class`. Isto atribui para o objeto (ou tipo `Class`) que representa o tipo dele mesmo.

```
//=====
```

Arrays.

Um **array (vetor)** é um **container** (recipiente) que abriga um número fixo de valores ou um tipo simples. O tamanho de um **array** é estabelecido quando o **array** é criado. Depois da criação, este tamanho é fixado.



Cada item em um array (vetor) é chamado um **elemento**, e cada elemento é acessado por um index (**índice**) numérico. Como mostra a ilustração acima, a numeração dos **índices** inicia com 0. O 9º elemento, por exemplo, por essa razão é acessado pelo índice 8.

O seguinte programa, **ArrayDemo**, cria um array de inteiros, coloca alguns valores nele, e mostra cada valor na saída padrão.

```
class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray;           //declara um array (vetor) de inteiros

        anArray = new int[10];   //aloca memória para 10 inteiros

        anArray[0] = 100; // inicializa o primeiro elemento
        anArray[1] = 200; // inicializa o segundo elemento
        anArray[2] = 300; // etc
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Elemento no índice 0: " + anArray[0]);
        System.out.println("Elemento no índice 1: " + anArray[1]);
        System.out.println("Elemento no índice 2: " + anArray[2]);
        System.out.println("Elemento no índice 3: " + anArray[3]);
        System.out.println("Elemento no índice 4: " + anArray[4]);
        System.out.println("Elemento no índice 5: " + anArray[5]);
        System.out.println("Elemento no índice 6: " + anArray[6]);
        System.out.println("Elemento no índice 7: " + anArray[7]);
        System.out.println("Elemento no índice 8: " + anArray[8]);
        System.out.println("Elemento no índice 9: " + anArray[9]);
    }
}
```

A saída do programa é:

```
Elemento no índice 0: 100
Elemento no índice 1: 200
Elemento no índice 2: 300
Elemento no índice 3: 400
Elemento no índice 4: 500
Elemento no índice 5: 600
Elemento no índice 6: 700
Elemento no índice 7: 800
Elemento no índice 8: 900
Elemento no índice 9: 1000
```

No mundo real da programação, você provavelmente usará um dos **construtores de looping** (**for**, **while**, e **do-while**) para iterar cada elemento do vetor, ao invés de escrever cada linha individualmente. Este exemplo é para ilustrar a sintaxe de um vetor.

Declarando uma Variável para Acessar um Array.

O programa abaixo declara uma **array** com a seguinte linha de código:

```
int[] anArray; // declara um array de inteiros
```

Como as declarações para variáveis de outros tipos, uma declaração tem dois componentes: o tipo e o nome do **array**. Um tipo de **array** é escrito como `type[]`, onde `type` é o tipo de dado dos elementos contidos; os colchetes (`[]`) são símbolos especiais indicando que esta variável refere-se a um **array**. O tamanho do **array** não é parte do tipo (por esse motivo os colchetes estão vazios). Um nome de **array** pode ser qualquer coisa que você quiser, contanto que ele siga as regras e convenções que foram previamente discutidas na seção *“Nomeando Variáveis”*. Assim como variáveis de outros tipos, a declaração não cria verdadeiramente um **array** – ela simplesmente diz ao compilador que esta variável contém um vetor de um determinado tipo.

Similarmente, você pode declarar **arrays** de outros tipos:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

Você pode também colocar os colchetes depois do nome do **array**:

```
float anArrayOfFloats[]; // Este é um formato desencorajado
```

No entanto, convenções desencorajam essa forma; os colchetes identificam o tipo do **array** e devem aparecer com a designação do tipo.

Criando, Inicializando, e Acessando um Array.

Um caminho para criar um **array** é o operador `new`. A próxima declaração no programa `ArrayDemo` aloca um **array** com memória suficiente para dez elementos inteiros e atribui o **array** para a variável `anArray`.

```
AnArray = new int[10]; // cria um array de inteiros
```

Se essa declaração estivesse faltando, o compilador mostraria um erro como o seguinte, e a compilação falharia:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

As próximas linhas atribuem valores para cada elemento de um **array**:

```
anArray[0] = 100; // inicializa o primeiro elemento
anArray[1] = 200; // inicializa o segundo elemento
anArray[2] = 300; //etc.
```

Cada elemento do **array** é acessado pelo seu índice numérico:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 0: " + anArray[1]);
System.out.println("Element 3 at index 0: " + anArray[2]);
```

Alternativamente, você pode usar um atalho na sintaxe para criar e inicializar um **array**:

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

Aqui o tamanho do **array** é determinado pelo número de valores fornecidos entre `{` e `}`.

Você pode também declarar um **array** de **arrays** (também conhecido como um **array multidimensional**) usando dois ou mais conjuntos de colchetes, como em `String [] [] names`. Cada elemento, dessa forma, será acessado por um número correspondente de valores indexados.

Na linguagem de programação Java, um **array multidimensional** é simplesmente um **array** cujos componentes são seus próprios **arrays**. Esse é um **array** diferente em **C** ou **Fortran**. A consequência disso é que são permitidas variações no tamanho das seqüências, como mostra o seguinte programa:

MultiDimArrayDemo:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[] [] names = {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"};
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones
    }
}
```

A saída do programa é:

```
Mr. Smith
Ms. Jones
```

Finalmente, você pode usar a propriedade embutida **length** para determinar o tamanho de um **array**. O código:

```
System.out.println(anArray.length);
```

mostrará o tamanho do **array** na saída padrão.

Copiando Arrays.

A classe **System** tem um método **arraycopy** que você pode usar para copiar eficientemente dados de uma **array** para outro:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

Os dois argumentos **Object** especificam o **array** do qual os dados serão copiados e o **array** de destino. Os argumentos **int** especificam a posição inicial no **array** fonte, iniciando a posição no **array** de destino, e o número de elementos que serão copiados.

O seguinte programa, **ArrayCopyDemo**, declara um **array** de elementos **char**, soletrando a palavra **“decaffeined”**. Ele usa **arraycopy** para copiar a seqüência de componentes do **array** para um segundo **array**:

```
class ArrayCopyDemo {
    public static void main (String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n',
            'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

A saída do programa é:

```
caffein
```

//=====

Operadores.

A partir do momento que já sabemos como declarar e inicializar variáveis, podemos conhecer o que podemos fazer com elas. Aprendendo os operadores da linguagem de programação Java é um bom lugar para começar. Operadores são símbolos especiais que executam operações específicas em um, dois, ou três operandos, e então retornam um resultado.

Os operadores na próxima tabela são listados de acordo com a ordem de precedência. Os que se encontram no topo da tabela, têm maior prioridade. Operadores com maior prioridade são avaliados antes dos operadores com prioridade relativamente menor. Operadores na mesma linha têm prioridade igual. Quando operadores de igual prioridade aparecem na expressão, uma regra obrigatória determina que o primeiro tem prioridade. Todos os operadores binários exceto para operadores de atribuição têm prioridade da esquerda para a direita; operadores de atribuição têm prioridade da direita para a esquerda.

Prioridade de Operadores	
posicionado após	<code>expr++ expr--</code>
unário	<code>++expr --expr + expr - expr ~ !</code>
multiplicativo	<code>* / %</code>
aditivo	<code>+ -</code>
deslocamento	<code><< >> >>></code>
relacional	<code>< > <= >= na instância</code>
igualdade	<code>== !=</code>
AND (E)	<code>&</code>
XOR (XOU)	<code>^</code>
OR (OU)	<code> </code>
lógico AND (E)	<code>&&</code>
lógico OR (OU)	<code> </code>
ternário	<code>? :</code>
atribuição	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>

Para o propósito geral da programação, certos operadores tendem a aparecer mais freqüentemente que outros; por exemplo, o operador de atribuição “=” é muito mais comum que o operador de mudança à direita “>>>”. A seguinte discussão foca primeiro nos operadores que você provavelmente usará regularmente, e termina focando nos que são menos comuns. Cada discussão é acompanhada por um exemplo de código que você pode compilar e executar. Estudando essas saídas ajudará você a reforçar o que você já aprendeu.

Operadores de Atribuição, Aritméticos e Unários.

O Operador de Atribuição Simples.

Um dos operadores mais comuns que você vai encontrar é o operador de atribuição simples “=”. Você viu esse operador na classe `Bicycle`; ele atribui o valor da direita para o operando da esquerda:

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

Este operador só pode ser usado em objetos para atribuir *referências ao objeto*.

Os Operadores Aritméticos.

A linguagem de programação Java fornece operadores que realizam adição, subtração, multiplicação, e divisão. Eles são fáceis de reconhecer por serem similares aos operadores matemáticos. O único símbolo que parece novo é o “%”, que divide um operando por outro e retorna o resto como resultado:

```
+ operador de adição (também usado para concatenação de strings)
- operador de subtração
* operador de multiplicação
/ operador de divisão
% operador de resto
```

O seguinte programa, `ArithmeticDemo`, testa os operadores matemáticos.

```
class ArithmeticDemo {
    public static void main (String[] args) {
        int result = 1 + 2; // result agora é 3
        System.out.println(result);

        result = result - 1; // result agora é 2
        System.out.println(result);

        result = result * 2; // result agora é 4
        System.out.println(result);

        result = result / 2; // result agora é 2
        System.out.println(result);

        result = result + 8; // result agora é 10
        result = result % 7; // result agora é 3
        System.out.println(result);
    }
}
```

Você pode também combinar operadores aritméticos com o operador de atribuição simples para criar atribuições compostas. Por exemplo, `x += 1`; e `x = x + 1`; ambos incrementam o valor de `x` em 1.

O operador `+` pode também ser usado para concatenar (juntar) duas strings, como mostra o seguinte programa `ConcatDemo`:

```
class ConcatDemo {
    public static void main(String[] args) {
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

//=====

Os Operadores Unários.

Os operadores unários requerem somente um operando; eles efetuam várias operações como uma incrementação/decrementação de um valor em um, negando uma expressão, ou invertendo o valor de um boolean.

- +** Operador unário de adição; indica valor positivo (números são positivos sem ele, no entanto)
- Operador unário de menos; nega uma expressão
- ++** Operador de incremento; incrementa o valor em 1
- Operador de decremento; decrementa o valor em 1
- !** Operador de complemento lógico; inverte o valor de um **boolean**

O seguinte programa, `UnaryDemo`, testa os operadores unários:

```
class UnaryDemo {

    public static void main(String[] args) {
        int result = +1; //result agora é 1
        System.out.println(result);
        result--; //result agora é 0
        System.out.println(result);
        result++; //result agora é 1
        System.out.println(result);
        result = -result; //result agora é -1
        System.out.println(result);
        boolean success = false;
        System.out.println(success); //false (falso)
        System.out.println(!success); //true (verdadeiro)
    }
}
```

Os operadores de incremento e decremento podem ser aplicados antes (prefixo) ou depois (sufixo) do operando. O código `result++`; e `++result`; ambos terminarão em `result` sendo incrementado em 1. A única diferença é que a versão prefixo (`++result`) calcula para o valor incrementado, enquanto a versão **postfix** (`result++`) calcula o valor original. Se você está somente efetuando um incremento/decremento simples, não faz diferença qual versão você escolher. Mas se você usa este operador como parte de uma expressão grande, o qual você escolher pode fazer uma diferença significativa.

O seguinte programa, `PrePostDemo`, ilustra o operador unário de incremento prefixo/sufixo:

```
class PrePostDemo {
    public static void main(String[] args) {
        int i = 3;
        i++;
        System.out.println(i); // "4"
        ++i;
        System.out.println(i); // "5"
        System.out.println(++i); // "6"
        System.out.println(i++); // "6"
        System.out.println(i); // "7"
    }
}
```

//=====

Operadores de Igualdade, Relacionais e Condicionais.

Os Operadores de Igualdade e Relacionais.

Os operadores de igualdade e relacionais determinam se um operando é maior que, menor que, ou diferente de outro operador. A maioria desses operadores você provavelmente achará familiar. Lembre-se de usar “==”, não “=”, quando testando de dois valores primitivos são iguais.

```

==   igual a
!=   diferente de
>    maior que
>=   maior ou igual que
<    menor que
<=   menor ou igual que

```

O seguinte programa, `ComparisonDemo`, testa os operadores de comparação:

```

class ComparisonDemo {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2) System.out.println("value1 == value2");
        if(value1 != value2) System.out.println("value1 != value2");
        if(value1 > value2) System.out.println("value1 > value2");
        if(value1 < value2) System.out.println("value1 < value2");
        if(value1 <= value2) System.out.println("value1 <= value2");
    }
}

```

Saída:

```

value1 != value2
value1 < value2
value1 <= value2

```

//=====

Os Operadores Condicionais.

Os operadores `&&` e `||` efetuam operações **Conditional-AND** (E condicional) e **Conditional-OR** (OU condicional) em duas expressões **booleanas**. Estes operadores exibem comportamentos *bit a bit*, de forma que o segundo operador é calculado somente se necessário.

```

&& Conditional-AND (E condicional)
|| Conditional_OR (OU condicional)

```

O seguinte programa, `ConditionalDemo1`, testa estes operadores:

```

class ConditionalDemo1 {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2)) System.out.println("value1
is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1)) System.out.println("value1
is 1 OR value2 is 1");
    }
}

```

Outro operador condicional é `?:`, que pode ser considerado como um substituto mais curto para um `if-then-else` (que será discutido mais tarde). Este operador é também conhecido como um **operador ternário** porque ele usa três operandos. No seguinte exemplo, este operador é mostrado lendo um “`IfsomeCondition` é `true`, atribui o valor do `value1` para `result`. Do contrário atribui o valor de `value2` para `result`.”

O seguinte programa, `ConditionalDemo2`, testa o operador `?:` :

```
class ConditionalDemo2 {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;

        System.out.println(result);
    }
}
```

Porque `someCondition` é verdadeiro, o programa mostra “1” na tela. Use o operador `?:` ao invés de uma declaração `if-then-else` se este tornar mais seu código mais legível; por exemplo, quando a expressão é compacta e sem efeitos secundários (como em atribuições).

O Operador de Comparação de Tipo `instanceof`.

O Operador `instanceof` compara um objeto de um tipo específico. Você pode usá-lo para testar se um objeto é uma instância de uma classe, uma instância de uma subclasse, ou uma instância de uma classe que implementa uma interface particular.

O seguinte programa, `InstanceofDemo`, define uma classe pai (nomeada `Parent`), uma interface simples (nomeada `MyInterface`), e uma classe filha (nomeada `Child`) que herda da pai e implementa a interface.

```
class InstanceofDemo {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        Parent obj2 = new Child();
        System.out.println("obj1 instanceof Parent: " + (obj1
instanceof Parent));
        System.out.println("obj1 instanceof Child: " + (obj1
instanceof Child));
        System.out.println("obj1 instanceof MyInterface: " + (obj1
instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: " + (obj2
instanceof Parent));
        System.out.println("obj2 instanceof Child: " + (obj2
instanceof Child));
        System.out.println("obj2 instanceof MyInterface: " + (obj2
instanceof MyInterface));
    }
}
class Parent{}
class Child extends Parent implements MyInterface{}
interface MyInterface{}
```

Saída:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

Operadores Bitwise e BitShift.

A linguagem de programação Java também fornece operadores que efetuam operações *bitwise* e *bitshift* em tipos integrais. Os operadores discutidos nesta seção são menos comumente usados. No entanto, sua cobertura é breve; a intenção é simplesmente tornar você ciente que estes operadores existem.

O complemento unário *bitwise* “~” inverte um padrão de *bits*; ele pode ser aplicado para qualquer um dos tipos inteiros, fazendo de cada “0” um “1” e de cada “1” um “0”. Por exemplo, um *byte* contém 8 *bits*; aplicando o operador para um valor cujo padrão é “00000000” mudaria seu padrão para “11111111”.

O operador *left shift* “<<” desloca um padrão de *bit* para a esquerda, e o operador *right shift* “>>” desloca um padrão de *bits* para a direita. O padrão de *bit* é dado pelo operando do lado esquerdo, e o número de posições para o deslocamento pelo lado operando do lado direito. O operador *right shift* “>>>” muda para zero na posição mais à esquerda, enquanto a posição mais à esquerda depois de “>>” depende da extensão no sinal.

O operador *bitwise* & fornece uma operação *bitwise AND* (E).

O operador *bitwise* ^ fornece uma operação *bitwise exclusive OR* (OU exclusivo).

O operador *bitwise* | fornece uma operação *bitwise inclusive OR* (OU inclusivo).

O seguinte programa, *BitDemo*, usa o operador *bitwise AND* para mostrar o número “2” na saída padrão.

```
class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x2222;
        System.out.println(val & bitmask); //Mostra "2"
    }
}
```

//=====

Expressões, Declarações e Blocos.

Operadores podem ser usados na construção de expressões, como calcular valores; expressões são os componentes centrais de declarações; declarações podem ser agrupadas em blocos.

Expressões.

Uma expressão é um construção feita de variáveis, operadores, e invocação de métodos, que são construídos de acordo com a sintaxe da linguagem, que calcula um valor único.

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Elemento 1 no índice 0: " + an Array[0]);

int result = 1 + 2; //resultado agora é 3
if(value1 == value2) System.out.println("value1 == value2");
```

O tipo de dado do valor retornado em uma expressão depende dos elementos usados na expressão. A expressão `cadence = 0` retorna `int` porque o operador de atribuição retorna um valor do mesmo tipo de dado que está no operando mais à esquerda; neste caso, `cadence` é um `int`. Como você pode ver nas outras expressões, uma expressão pode retornar outros tipos de valores como fonte, como um `boolean` ou `String`.

A linguagem de programação Java permite a você construir expressões compostas de várias pequenas expressões desde que o tipo de dado requerido por uma parte da expressão seja igual ao tipo de dado da outra. Por exemplo:

```
1 * 2 * 3
```

Neste exemplo em particular, a ordem em que a expressão é avaliada não tem importância porque o resultado da multiplicação independe da ordem; a saída é sempre a mesma, não importa em qual ordem você aplica as multiplicações. No entanto, isto não é verdade para todas as expressões. Por exemplo, a seguinte expressão retorna resultados diferentes, dependendo se você coloca o operador de adição ou de divisão primeiro:

```
x + y / 100 //ambíguo
```

Você pode especificar exatamente como uma expressão será avaliada usando parênteses: (e). Por exemplo, para fazer a expressão anterior sem ambigüidade, você poderia escrever como segue:

```
(x + y) / 100 //não ambígua, recomendada
```

Se você não indicar explicitamente a ordem para as operações serem calculadas, a ordem é determinada pela prioridade designada para o operador em uso na expressão. Operadores que têm uma prioridade maior são calculados primeiro. Por exemplo, o operador de divisão tem uma prioridade maior que o operador de adição. Portanto, as duas declarações a seguir são equivalentes:

```
x + y / 100
x + (y / 100) //não ambígua, recomendada
```

Quando estiver escrevendo expressões compostas, seja explícito e indique com parênteses quais operadores devem ser avaliados primeiro. Esta prática faz com que seu código seja mais fácil de ler e de fazer manutenção.

```
//=====
```

Declarações.

Declarações são de uma forma rudimentar equivalentes a sentenças na linguagem natural. Uma declaração (**statement**) forma uma unidade completa de execução. Os seguintes tipos de expressões podem ser feitas em uma declaração terminando a expressão com um ponto-e-vírgula (;):

- Declaração de especificação;
- Qualquer uso de `++` ou `--`;
- Invocação de métodos;
- Expressão de criação de objetos.

Declarações também são chamadas de **expressions statements** (expressões de declaração). A seguir são mostrados alguns exemplos de expressões de declaração:

```
aValue = 8933.234;           // declaração de especificação
aValue++;                   // declaração de incremento
System.out.println("Hello World!"); // declaração de invocação de método
Bicycle myBike = new Bicycle(); // declaração de criação de objeto
```

Em adição às expressões de declaração, há dois outros tipos de declarações: **declaration statement** (sentença de declaração) e **control flow statements** (declaração de controle de fluxo). Uma sentença de declaração declara uma variável. Você verá muitos exemplos de sentenças de declaração prontas:

```
double aValue = 8933.234; // sentença de declaração
```

Finalmente, as declarações de controle de fluxo, regulam a ordem em que as declarações serão executadas. As declarações dentro de seu código fonte são geralmente executadas de cima para baixo, na ordem em que elas aparecem. Declarações de controle de fluxo, no entanto, param o fluxo da execução usando comandos de decisão, **looping**, e desvio de execução, habilitando seu programa para executar condicionalmente blocos particulares do código. As declarações de controle de fluxo são as declarações de comandos de decisão (**if-then**, **if-then-else**, **switch**), as declarações de **looping** (**for**, **while**, **do-while**), e declarações de desvio (**break**, **continue**, **return**) suportadas pela linguagem de programação Java.

Blocos.

Um bloco é um grupo de zero ou mais declarações entre chaves e podem ser usadas em qualquer lugar que uma declaração simples é permitida. O seguinte exemplo, **BlockDemo**, ilustra o uso de blocos:

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // começa o bloco 1
            System.out.println("Condição é verdadeira.");
        } // fim do bloco 1
        else { // começa o bloco 2
            System.out.println("Condição é falsa.");
        } // fim do bloco 2
    }
}
```

```
//=====
```

Declarações de Controle de Fluxo.

As declarações dentro de seus arquivos fonte são geralmente executadas de cima para baixo, na ordem em que elas aparecem. As declarações de controle de fluxo, no entanto, quebram o fluxo da execução empregando controles de decisão, **looping**, e desvio de execução, habilitando seu programa para executar condicionalmente um bloco particular do código. Esta seção descreve as declarações de controle de decisão (**if-then**, **if-then-else**, **switch**), as declarações de **looping** (**for**, **while**, **do-while**), e as declarações de desvio (**break**, **continue**, **return**) suportadas pela linguagem de programação Java.

As Declarações **if-then**.

A declaração **if-then** é a mais básica de todas as declarações de controle de fluxo. Ela diz a seu programa para executar uma certa seção do código somente se um teste em particular for verdadeiro (**true**). Por exemplo, a classe **Bicycle** poderia permitir uma parada para diminuição da velocidade da bicicleta **somente se** a bicicleta estivesse em movimento. Uma possível implementação do método **applyBrakes** poderia ser como a seguinte:

```
void applyBrakes() {
    if (isMoving) { // a cláusula "if": bicicleta precisa estar se movendo
        currentSpeed--; // a cláusula "then": diminui a velocidade atual
    }
}
```

Se este teste for avaliado como **false** (considerando que a bicicleta não está em movimento), o controle pula para o fim da declaração **if-then**.

Em adição, a abertura e fechamento das chaves é opcional, desde que a cláusula "**then**" contenha somente uma declaração:

```
void applyBrakes() {
    if (isMoving) currentSpeed--; //igual à anterior, mas sem chaves
}
```

Decidir quando omitir as chaves é assunto de gosto pessoal. Omitindo elas pode fazer o código mais frágil. Se uma segunda declaração é mais tarde adicionada para a cláusula "**then**", um erro comum poderia ser esquecer de adicionar as chaves novamente requeridas. O compilador não percebe este tipo de erro; você só conseguirá um resultado incorreto.

//=====

As Declarações **if-then-else**.

A declaração **if-then-else** fornece um caminho secundário para a execução quando uma cláusula "**if**" é avaliada como **false**. Você poderia usar uma declaração **if-then-else** no método **applyBrakes** para executar alguma ação se os freios são aplicados quando a bicicleta não está em movimento. Neste caso, a ação é simplesmente mostrar uma mensagem de erro dizendo que a bicicleta já está parada.

```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.out.println("A bicicleta está parada no momento!");
    }
}
```

O seguinte programa, `IfElseDemo`, especifica uma graduação baseada em um valor de uma teste de contagem: um A para um escore de 90% ou mais, um B para um score de 80% ou mais e assim por diante:

```
class IfElseDemo {
    public static void main(String[] args) {

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

A saída do programa é:

```
Grade = C
```

Você deve ter observado que o valor do `testscore` pode satisfazer mais de uma expressão na declaração composta `76 >= 70` e `76 >= 60`. No entanto, se a condição é satisfeita, a declaração apropriada será executada (`grade = 'C'`) e o restante das condições não serão avaliadas.

A Declaração `switch`.

Diferente de `if-then` e `if-then-else`, a declaração `switch` é permitida para qualquer número de possíveis caminhos de execução. Uma declaração `switch` trabalha com os tipos de dados primitivos `byte`, `short`, `char`, e `int`. Ela também trabalha com tipos enumerados e um pequeno número de classes especiais que abrigam certos tipos primitivos: `Character`, `Byte`, `Short`, e `Integer`.

O seguinte programa, `SwitchDemo`, declara um `int` nomeado `month` cujo valor representa um mês do ano. O programa mostra o nome do mês, baseado no valor do mês, usando a declaração `switch`:

```
class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        switch (month) {
            case 1: System.out.println("January"); break;
            case 2: System.out.println("February"); break;
            case 3: System.out.println("March"); break;
            case 4: System.out.println("April"); break;
            case 5: System.out.println("May"); break;
            case 6: System.out.println("June"); break;
            case 7: System.out.println("July"); break;
            case 8: System.out.println("August"); break;
            case 9: System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid month"); break;
        }
    }
}
```

Neste caso, "`August`" é mostrado na saída padrão.

O corpo de uma declaração `switch` é conhecido como **bloco switch**. Qualquer declaração imediatamente contida no bloco `switch` pode ser rotulada com um ou mais rótulos `case` ou `default`. A declaração `switch` avalia esta expressão e executa o `case` apropriado.

Naturalmente, você poderia também implementar a mesma coisa com a declaração `if-then-else`:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
} ... // e assim por diante
```

Decidir se usa declaração `if-then-else` ou declaração `switch` é algumas vezes um convite à análise da questão. Você pode decidir se usa um ou outro baseado na legibilidade e em outros fatores. Uma declaração `if-then-else` pode ser usada para fazer decisões baseadas no alcance de valores ou condições, considerando que uma declaração `switch` pode fazer decisões baseadas somente em um valor inteiro simples ou enumerado.

Outro ponto interessante é a declaração `break` depois de cada `case`. Cada declaração `break` termina envolvendo uma declaração `switch`. O controle de fluxo continua com a primeira declaração seguida do bloco `switch`. A declaração `break` é necessária porque sem ela, a declaração `case` seguiria sem interrupção; ou seja, sem um `break` explícito, o controle de fluxo seqüencialmente seguiria para a declaração `case` subsequente. O seguinte programa, `SwitchDemo2`, ilustra porque isto pode ser útil para ter declarações `case` seguindo sem interrupção:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if ( ((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0) )
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

A saída do programa é: `Number of Days = 29`

As Declarações `while` e `do-while`.

A declaração `while` executa continuamente um bloco de declarações enquanto uma condição em particular é `true` (verdadeira). Esta sintaxe pode ser expressada como:

```
while (expressão) {
    declaração (s)
}
```

A declaração `while` avalia a expressão, que deve ser um valor `boolean` (booleano). Se a expressão é avaliada como `true`, a declaração `while` executa a declaração (s) no bloco `while`. A declaração `while` continua testando a expressão e executando o bloco até que o expressão seja avaliada como `false` (falsa). Por exemplo, para usar a declaração `while` para mostrar o valor de 1 até 10 pode ser feito como no seguinte programa `WhileDemo`:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

Você pode implementar um `loop` infinito usando a declaração `while` como segue:

```
while (true) {
    // aqui vai seu código
}
```

A linguagem de programação Java também fornece uma declaração `do-while`, que pode ser expressada como segue:

```
do {
    declaração (s)
} while (expressão);
```

A diferença entre `do-while` e `while` é que `do-while` avalia esta expressão de baixo do `loop` para cima. Por essa razão, as declarações dentro do bloco `do` são sempre executadas no mínimo uma vez, como mostra o seguinte programa `DoWhileDemo`:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count id: " + count);
            count++;
        } while (count <= 11);
    }
}
```

//=====

A Declaração `for`.

A declaração `for` fornece um modo compacto para iterar um alcance de valores. Programadores freqüentemente referem-se a ela como “*for loop*” por causa do modo em que este *loop* é repetido até que uma condição em particular é satisfeita. A forma geral da declaração `for` pode ser expressada é a seguinte:

```
for (inicialização; terminação; incremento) {
    declaração (s)
}
```

Quando usando esta versão da declaração `for`, mantenha em mente que:

- A expressão de *inicialização* inicializa o *loop*; ela é executada uma vez, quando o *loop* começa.
- Quando a expressão de *terminação* é avaliada como `false`, o *loop* termina.
- A expressão de *incremento* é invocada depois de cada iteração até o fim do *loop*; isto é perfeitamente aceitável para esta expressão para incrementar ou decrementar um valor.

O seguinte exemplo de programa, `ForDemo`, usa uma forma geral para a declaração `for` para mostrar os números de 1 até 10 na saída:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

A saída do programa é:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Note como o código declara uma variável dentro da expressão de inicialização. O escopo da variável estende-se desta declaração para o final do bloco governado pela declaração `for`, ela pode ser usada como fonte na expressão de terminação e incremento. Se a variável que controla a declaração `for` não é necessária do lado de fora do *loop*, é melhor declarar a variável na expressão de inicialização. Os nomes `i`, `j`, e `k` são freqüentemente usados para controlar *loops for*; declarando elas dentro da expressão de inicialização limita seu período de vida e reduz erros.

As três expressões para o *loop for* são opcionais; um *loop* infinito pode ser criado como segue:

```
for ( ; ; ) { // loop infinito
    // seu código vai aqui
}
```

A declaração `for` também possui outras formas proporcionais para iteração dentro de `Collections` e `arrays`. Esta forma é algumas vezes referida como uma *declaração for otimizado* (*enhanced for*), e pode ser usada para fazer seus *loops* mais compactos e fáceis de ler. Para demonstrar, considere o seguinte `array`, que recebe os números de 1 a 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

O seguinte programa, `EnhancedForDemo`, usa o **for otimizado** para o loop atravessar de uma parte a outra do **array**:

```
class EnhancedForDemo {
    public static void main(String[] args) {
        int[] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
            System.out.println("Count is: " + item);
        }
    }
}
```

Neste exemplo, a variável `item` recebe o valor corrente do **array** de números. A saída deste programa é a mesma de antes:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

É recomendável usar esta forma de declaração **for** ao invés da forma geral sempre que possível.

//=====

Declarações de Desvio.

A Declaração **break**.

A declaração **break** tem duas formas: **rotulada** e **não rotulada**. Você viu a forma não rotulada na discussão prévia da declaração **switch**. Você também pode usar um **break** sem rótulo para terminar um **loop for**, **while**, ou **do-while**, como mostra o seguinte programa `BreakDemo`:

```
class BreakDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                             2000, 8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

O programa procura pelo número 12 no **array**. A declaração **break** termina o **loop for** quando o valor é encontrado. O controle de fluxo então transfere para a declaração de saída o fim do programa. A saída é:

```
Found 12 at index 4
```

Um **break** sem rótulo termina uma declaração **switch**, **for**, **while**, ou **do-while**, mas um **break** rotulado termina uma declaração exterior. O seguinte programa, **BreakWithLabelDemo**, é similar ao programa anterior, mas usa **loops for** aninhados para procurar por um valor em um **array** bi-dimensional. Quando o valor é encontrado, o **break** rotulado termina o **loop for** exterior (rotulado "**search**"):

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = { { 32, 87, 3, 589 },
                                { 12, 1076, 2000, 8 },
                                { 622, 127, 77, 955 }
                                };

        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor +
                               " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

A saída do programa é:

```
Found 12 at 1, 0
```

A declaração **break** termina a declaração rotulada; ele não transfere o controle de fluxo para o rótulo. O controle de fluxo é transferido para a declaração imediatamente seguinte à declaração rotulada (terminada).

```
//=====
```

A Declaração `continue`.

A declaração `continue` salta a iteração corrente de um loop `for`, `while`, ou `do-while`. A forma não rotulada salta para o final do corpo do loop interior e avalia a expressão booleana que controla o loop. O seguinte programa, `ContinueDemo`, percorre através de uma `String`, contando as ocorrências da letra "p". Se o carácter corrente não é um `p`, a declaração `continue` salta o resto do loop e continua no próximo carácter. Se ele é um "p", o programa incrementa o contador de letras `count`.

```
class ContinueDemo {
    public static void main(String[] args) {

        String searchMe = "peter piper picked a peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interessado somente em p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // processa p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Esta é a saída do programa:

```
Found 9 p's in the string.
```

Para ver o efeito mais claramente, tente remover a declaração `continue` e recompile. Quando você rodar o programa novamente o contador estará errado, dizendo que encontrou 35 p's ao invés de 9.

Uma declaração rotulada **continue** salta a iteração corrente de um outro **loop** marcado com o rótulo dado. O seguinte programa exemplo, `ContinueWithLabelDemo`, usa **loops** aninhados para procurar por uma **substring** dentro de uma **string**. Dois **loops** aninhados são requeridos: um para iterar acima a **substring** e um para iterar acima a **string** sendo procurada. O seguinte programa, `ContinueWithLabelDemo`, usa a forma rotulada de **continue** para saltar uma iteração em outro **loop**.

```
class ContinueWithDemo {
    public static void main(String[] args) {

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++)
                    != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
            break test;
        }
        System.out.println(foundIt ? "Found it" :
                            "Didn't find it");
    }
}
```

Esta é a saída do programa:

```
Found it
```

A Declaração **return**.

A última das declarações de desvio é a declaração **return**. A declaração **return** sai do método corrente, e o controle de fluxo retorna para onde o método foi invocado. A declaração **return** tem duas formas: uma que retorna um valor, uma que não. Para retornar um valor, simplesmente coloque o valor (ou a expressão que calcula o valor) depois da palavra reservada **return**.

```
return ++count
```

O tipo de dado do valor retornado deve ser igual ao tipo do método declarado para retornar o valor. Quando um método é declarado **void** (vazio), use a forma **return** que não retorna um valor.

```
return
```

Classes.

Na introdução aos conceitos de orientação a objeto foi usado uma classe `Bicycle` em um exemplo, com *racing bikes*, *mountain bikes*, e *tandem bikes* como *subclasses*. Aqui está um código exemplo para uma possível implementação de uma classe `Bicycle`, para dar uma visão geral de uma declaração de classe. A seção subsequente desta lição explicará declarações de classe passo a passo:

```
public class Bicycle {

    // a classe Bicycle tem três campos
    public int cadence;
    public int gear;
    public int speed;

    // a classe Bicycle tem um construtor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // a classe Bicycle tem quatro métodos
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

A declaração de classe para uma classe `MountainBike` que é uma subclasse de `Bicycle` poderia ser parecida com isto:

```
public class MountainBike extends Bicycle {

    // a subclasse MountainBike tem um campo
    public int seatHeight;

    // a subclasse MountainBike tem um construtor
    public MountainBike(int startHeight, int startCadence, int startSpeed,
        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // a subclasse MountainBike tem um método
    public void seatHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

`MountainBike` herda todos os campos e métodos de `Bicycle` e adiciona o campo `seatHeight` e um método para iniciá-lo (*mountain bikes* têm assentos (*seats*) que podem ser movidos acima e abaixo de acordo com o terreno).

Declarando Classes.

Você deve ter visto classes definidas no seguinte estilo:

```
class MyClass {
    // declarações de campo, construtor, e métodos
}
```

Isto é uma **declaração de classe**. O **corpo da classe** (a área entre as chaves) contém todo o código que prepara o ciclo de existência dos objetos criados da classe: construtores para inicializar novos objetos, declarações para os campos que fornecem o estado da classe deste objeto, e métodos para implementar o comportamento da classe e seus objetos.

Você pode fornecer mais informações a respeito da classe, como o nome de sua superclasse, se ela implementa alguma interface e assim por diante, no início da declaração da classe. Por exemplo,

```
class MyClass extends MySuperClass implements YourInterface {
    // declarações de campos, construtores e métodos
}
```

informa que **MyClass** é uma subclasse de **MySuperClass** e que implementa a interface **YourInterface**.

Você também pode adicionar modificadores como **public** ou **private**. Os modificadores **public** e **private** determinam se outra classe pode acessar **MyClass**. A lição sobre **interfaces** e **heranças** explicará como e porque você deve usar as palavras reservadas **extends** e **implements** em uma declaração de classe.

Em geral, declarações de classes podem incluir estes componentes, em ordem:

- Modificadores como **public**, **private**, e outros que você encontrará mais tarde.
- O nome da classe, com a letra inicial em maiúscula por convenção.
- O nome da classe pai (superclasse), nesse caso, precedido pela palavra reservada **extends**. Uma classe só pode estender (subclasse) um pai.
- Uma lista de interfaces separadas por vírgula implementadas pela classe, nesse caso, precedida pela palavra reservada **implements**. A classe pode implementar mais de uma interface.
- O corpo da classe, envolvida por chaves.

Declarando Membros Variáveis.

Há vários tipos de variáveis:

- Membros variáveis em uma classe – estas são chamadas **fields** (campos).
- Variáveis em um método ou bloco de código – estas são chamadas **variáveis locais**.
- Variáveis na declaração do método – estas são chamadas **parâmetros**.

A classe **Bicycle** usa as seguintes linhas de código para definir estes campos:

```
public int cadence;
public int gear;
public int speed;
```

Declarações de campo são compostas de três componentes, em ordem:

- Zero ou mais modificadores, como um **public** ou **private**.
- O tipo de campo.
- O nome do campo.

Os campos **Bicyce** são nomeados **cadence**, **gear**, e **speed** e são todos tipos de dado inteiro (**int**). A palavra reservada **public** identifica estes campos como membros públicos, acessíveis por qualquer objeto que acesse a classe.

Modificadores de Acesso.

O primeiro modificador (mais à esquerda) usado deixa você controlar se outras classes têm acesso a um campo membro. Por este momento, considere somente `public` e `private`. Outros modificadores de acesso serão discutidos mais tarde.

- Modificador `public` – este campo é acessível de todas as classes.
- Modificador `private` – este campo é acessível somente dentro da própria classe.

No espírito do encapsulamento, é comum fazer campos `private`. Isto faz com que somente sejam acessados diretamente da classe `Bicycle`. Nós poderemos precisar acessar estes valores, no entanto. Isto pode ser feito indiretamente adicionando o método `public` que obtêm um valor de campo para nós:

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public int getGear() {
        return gear;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

//=====

Definindo Métodos.

Aqui está um típico exemplo de declaração de método:

```
public double calculateAnswer(double sinSpan, int numberOfEngines,
                             double length, double grossTons) {
    // faça os cálculos aqui
}
```

Os únicos elementos requeridos de uma declaração de método é o tipo de retorno do método, nome, um par de parenteses, `()`, e o corpo entre chaves, `{}`.

De forma geral, declarações de métodos têm seis componentes, em ordem:

- **Modificadores** – como `public`, `private`, e outros que você aprenderá mais tarde.
- **O tipo de retorno** – o tipo de dado do valor retornado pelo método, ou `void` se o método não retorna um valor.
- **O nome do método** – as regras para nomes de campos se aplicam para o nome de métodos também, mas a convenção é um pouco diferente.
- **A lista de parâmetros entre parênteses** – uma vírgula delimita a lista de parâmetros de saída, precedida pelo seu tipo de dado, envolvida por parênteses, `()`. Se não há parâmetros, você simplesmente usa parênteses vazios.
- **Uma lista de exceções** – que será discutida mais tarde.
- **O corpo do método, envolvido entre chaves** – o código do método incluindo a declaração de variáveis locais, vai aqui.

Modificadores, tipos de retorno, e parâmetros serão discutidos mais tarde nesta lição. Exceções serão discutidas em outra lição.

Definição: dois componentes de uma declaração de método compõem a assinatura do método – o nome do método e os tipos de parâmetros.

A assinatura do método declarado acima mencionado:

```
calculateAnswer(double, int, double, double)
```

Nomeando um Método.

Apesar de um nome de método poder ser qualquer identificador válido, convenções de código restringem a nomeação de métodos. Por convenção nomes de métodos devem ser um verbo em letras minúsculas ou uma palavra composta que começa com um verbo em letras minúsculas, seguida por adjetivos, substantivos, etc. Em nomes compostos, a primeira letra de cada segunda e seguintes palavras deve ser maiúscula. Aqui está um exemplo:

```
run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty
```

Tipicamente, um método tem um único nome dentro da classe. No entanto, um método pode ter o mesmo nome de outro método previsto para **method overloading** (método sobrecarregado).

```
//=====
```

Métodos sobrecarregados (overloading methods).

A linguagem de programação Java suporta métodos sobrecarregados, e Java pode distinguir entre métodos com diferentes assinaturas. Isto significa que métodos dentro de uma classe podem ter o mesmo nome se eles tiverem diferentes listas de parâmetros (há alguns requisitos para isso que serão discutidos na lição “Interfaces e Herança”).

Supondo que você tem uma classe que pode usar caligrafia para desenhar vários tipos de dados (**strings**, **integers**, a assim por diante) e que contém um método para desenhar cada tipo de dado. É incômodo usar um novo nome para cada método – por exemplo, **drawString**, **drawInteger**, **drawFloat**, e assim por diante. Na linguagem de programação Java, você pode usar o mesmo nome para todos os métodos de desenho passando uma lista de argumentos diferentes para cada método. Desse modo, a classe que desenha dados pode declarar quatro métodos nomeados **draw**, cada um com uma lista de parâmetros diferente.

```
Public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Métodos sobrecarregados são diferenciados pelo número e o tipo de argumentos passados no método. No código exemplo, **draw(String s)** e **draw(int i)** são métodos distintos e únicos porque requerem diferentes tipos de argumentos.

Você não pode declarar mais de um método com o mesmo nome e mesmo número e tipo de argumentos, porque o compilador não pode distingui-los como diferentes.

O compilador não considera tipo de retorno quando diferenciando métodos então você não pode declarar dois métodos com a mesma assinatura se eles têm um tipo de retorno diferente.

Nota: Métodos sobrecarregados deveriam ser usados raramente, pois podem fazer seu código muito menos legível.

```
//=====
```

Produzindo Construtores para suas Classes.

Uma classe contém construtores que são invocados para criar objetos da classe projetada. Declarações de construtores parecem com declarações de métodos – exceto que elas usam o nome da classe e não têm tipo de retorno. Por exemplo, `Bicycle` tem um construtor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

Para criar um novo objeto `Bicycle` chamado `myBike`, um construtor é chamado pelo operador `new`:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

`new Bicycle(30, 0, 8)` cria espaço na memória para o objeto e inicializa estes campos.

Apesar de `Bicycle` só ter um construtor, ele poderia ter outros, incluindo um construtor sem argumentos:

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

`Bicycle yourBike = new Bicycle();` invoca um construtor sem argumento para criar um novo objeto `Bicycle` chamado `yourBike`.

Ambos os construtores poderiam ser declarados em `Bicycle` porque têm diferentes listas de argumentos. Assim como métodos, a plataforma Java diferencia construtores baseada no número de argumentos na lista de seus tipos. Você não pode escrever dois construtores que têm o mesmo número e tipo de argumentos para a mesma classe, porque a plataforma não poderia diferenciá-los. Se fizer assim, causará um erro em tempo de compilação.

Você não pode fornecer qualquer construtor para sua classe, mas você pode ser cuidadoso ao fazer isso. O compilador automaticamente fornece um construtor padrão sem argumento para qualquer classe sem construtores. O construtor padrão (**default**) poderá chamar o construtor sem argumento da superclasse. Nesta situação, o compilador verifica se a superclasse não tem um construtor sem argumentos, então você precisa certificar-se de que ele tenha. Se sua classe não tem uma superclasse explícita, então ele precisa de uma superclasse implícita do `Object`, o qual trabalha possuindo um construtor sem argumento.

Você pode usar uma superclasse construtora dela mesma. A classe `MountainBike` no começo da lição faz justamente isso. Isso será discutido mais tarde, na lição sobre interfaces e herança.

Você pode acessar modificadores em uma declaração de construtor para controlar como outras classes podem chamar o construtor.

Nota: Se qualquer classe não pode chamar um construtor `MyClass`, ela não pode criar diretamente objetos `MyClass`.

```
//=====
```

Passando Informações para um Método um um Construtor.

A declaração para um método ou um construtor declara o número e o tipo de argumentos para aquele método ou construtor. Por exemplo, o seguinte método computa os pagamentos mensais para uma casa de empréstimos, baseado no valor do empréstimo, na taxa de juros, o tamanho do empréstimo (o número de períodos), e o valor futuro do empréstimo:

```
public double computePayment(double loanAmt,
                             double rate,
                             double futureValue,
                             int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest), -numPeriods);
    double denominador = (1 - partial1) / interest;
    double answer = (-loanAmt / denominador)
        - (( futureValue * partial1) / denominador);
    return answer;
}
```

Este método tem quatro parâmetros: o montante do empréstimo, a taxa de juros o valor futuro e o número de períodos. Os primeiros três são números de pontos flutuantes de dupla precisão, e o quarto é um inteiro. Os parâmetros são usados no corpo de método e em tempo de execução receberão nos valores os argumentos que serão passados nele.

Nota: Parâmetros referem-se à lista de variáveis na declaração do método. **Argumentos** são os valores atuais que são passados nele quando o método é invocado. Quando você invoca um método, os argumentos usados precisam coincidir com a declaração dos parâmetros em tipo e ordem.

//=====

Tipos de Parâmetros.

Você pode usar qualquer tipo de dado para um parâmetro de um método ou construtor. Isto inclui tipos de dados primitivos, como **doubles**, **floats**, e **integers**, como você viu no método `computePayment`, e referência a tipos de dados, como os objetos e **arrays**.

Aqui está um exemplo de um método que aceita um **array** em um argumento. Neste exemplo, o método cria um novo objeto `Polygon` e inicializa ele de um objeto **array** de `Point` (assuma que `Point` é uma classe que representa uma coordenada x, y):

```
public Polygon polygonFrom(Point[] corners) {
    // corpo do método vai aqui
}
```

Nota: A linguagem de programação Java não deixa você passar métodos em métodos. Mas você pode passar um objeto em um método e então invocar o método do objeto.

//=====

Número Arbitrário de Argumentos.

Você pode usar um construtor chamado **varargs** para passar um número arbitrário de valores para um método. Você usa **varargs** quando você não sabe como muitos argumentos de um tipo em particular serão passados para o método. Isto é um atalho para a criação manual de um **array** (o método prévio poderia usar **varargs** preferivelmente ao **array**).

Para usar **varargs**, você segue o tipo do último parâmetro por reticências (três pontos, ...), então um espaço, e o nome do parâmetro. O método pode então ser chamado com qualquer número daquele parâmetro, inclusive nenhum.

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)*(corners[1].x - corners[0].x)
        + (corners[1].y - corners[0].y)*(corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);
    // mais códigos do corpo de método seguem o que foi criado
    // e retornam um polígono conectando pontos
}
```

Você pôde ver que dentro do método, **corners** (ângulos) são tratados como um **array**. O método pode ser chamado igualmente com um **array** ou com uma seqüência de argumentos. O código no corpo do método tratará o parâmetro como um **array** em qualquer caso.

Você verá mais comumente **varargs** com os métodos **printing**; por exemplo, este método **printf**:

```
public PrintStream printf(String format, Object... args)
```

permite a você mostrar um arbitrário número de objetos. Ele pode ser chamado assim:

```
System.out.println("%s: %d, %s%n", name, idnum, address);
```

ou algo como isto:

```
System.out.println("%s: %d, %s, %s, %s%n", name, idnum, address, phone,
email);
```

ou ainda com um número diferente de argumentos.

```
//=====
```

Nomes de Parâmetros.

Quando você declara um parâmetro para um método ou um construtor, você fornece um nome para aquele parâmetro. Este nome é usado dentro do corpo de método para referir-se para o argumento passado.

O nome do parâmetro precisa ser único neste escopo. Ele não pode ter o mesmo nome de outro parâmetro para o mesmo método ou construtor, e ele não pode ser o nome de uma variável local dentro do método ou construtor.

Um parâmetro pode ter o mesmo nome que um campo da classe. Se este é o caso, o parâmetro é chamado para **sombrear** o campo. **Sombreamento de campos** pode fazer seu código difícil de ler e é convencionalmente usado somente dentro de construtores e métodos que apontam um campo em particular. Por exemplo, considere a seguinte classe **Circle** e seu método **setOrigin**:

```
public class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...
    }
}
```

A classe **Circle** teve três campos: **x**, **y**, e **radius**. O método **setOrigin** teve dois parâmetros, cada um deles teve o mesmo nome de um dos campos. Cada parâmetro de método sombreou o campo que compartilha seu nome. Então usando os nomes simples **x** e **y** dentro do corpo do método referiu ao parâmetro, não ao campo. Para acessar o campo, você precisaria usar um nome qualificador. Isto será discutido mais tarde na lição intitulada "Usando a Palavra Reservada **this**."

Passando Argumentos de Tipos Primitivos de Dados.

Argumentos primitivos, como um `int` ou um `double`, são passados nos métodos por valor. Isto significa que quaisquer mudanças nos valores dos parâmetros existem somente dentro do escopo do método. Quando o método retornar, o parâmetro será dado e qualquer mudanças nele serão perdidas. Aqui está um exemplo:

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        // invoca passMethod() com x como argumento
        passMethod(x);

        // mostra x para ver se este valor foi modificado
        System.out.println("Depois de invocar passMethod, x = " + x);

    }

    // muda o parâmetro em passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

Quando você roda o programa, a saída é:

```
Depois de invocar passMethod, x = 3
```

Passando Referência de Argumentos de Tipos de Dados.

Parâmetros de Referência de Tipos de Dados, assim como em objetos, são também passados nos métodos por valor. Isto significa que quando o método retorna, a referência passada referencia o mesmo objeto como antes. No entanto, os valores dos campos do objeto podem ser mudados no método se eles tiverem o próprio acesso nivelado.

Por exemplo, considere o método em uma classe arbitrária que move um objeto `Circle`:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // código para mover origem do círculo para x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // código para apontar uma nova referência para o círculo
    circle = new Circle(0, 0);
}
```

Deixe o método ser invocado com estes argumentos:

```
moveCircle(myCircle, 23, 56)
```

Dentro do método, `circle` inicialmente refere-se para `myCircle`. O método muda as coordenadas `x` e `y` do objeto ao qual `circle` faz referência (i.e., `myCircle`) para 23 e 56, respectivamente. Essas mudanças persistirão quando o método retornar. Então `circle` é apontado para referenciar para um novo objeto `Circle` com `x = y = 0`. A redesignação não permanece, no entanto, porque a referência foi passada nele por valor e não pode mudar. Dentro do método, o objeto dirigido para `circle` foi mudado, mas, quando o método retornar, `myCircle` não obstante referencia o mesmo objeto `Circle` do mesmo modo que antes do método ser chamado.

```
//=====
```

Objetos.

Um programa típico Java cria muitos objetos, que como você tem visto, interagem invocando métodos. Durante a interação desses objetos, o programa pode executar várias tarefas, como uma implementação de uma **GUI**, rodar animações, ou mandar e receber informações entre uma rede. Logo que um objeto termina o trabalho para o qual foi criado, seus recursos são reciclados para uso de outros objetos.

Aqui está um pequeno programa, chamado `CreateObjectDemo`, que cria três objetos: um objeto `Point` e dois objetos `Rectangle`. Você precisará de todos os três arquivos fonte para compilar o programa:

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declara e cria um objeto ponto
        // e dois objetos retângulo.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        // mostra largura, altura e área de rectOne
        System.out.println("Largura de rectOne: " + rectOne.width);
        System.out.println("Altura de recOne: " + rectOne.height);
        System.out.println("Área de rectOne: " + rectOne.getArea());

        // aponta para a posição de rectTwo
        rectTwo.origin = originOne;

        // mostra a posição de rectTwo
        System.out.println("Posição X de rectTwo: " + rectTwo.origin.x);
        System.out.println("Posição Y de rectTwo: " + rectTwo.origin.y);

        // move rectTwo e mostra sua nova posição
        rectTwo.move(40, 72);
        System.out.println("Posição X de rectTwo: " + rectTwo.origin.x);
        System.out.println("Posição Y de rectTwo: " + rectTwo.origin.y);
    }
}
```

O programa cria, manipula, e mostra informações sobre vários objetos. Aqui está a saída:

```
Largura de rectOne: 100
Altura de rectOne: 200
Area de rectOne: 20000
Posição X de rectTwo: 23
Posição Y de rectTwo: 94
Posição X de rectTwo: 40
Posição Y de rectTwo: 72
```

As três sessões seguintes usam o exemplo anterior para descrever o ciclo de existência de um objeto dentro de um programa. Nelas, você aprenderá como escrever códigos que criam e usam objetos em seus próprios programas. Você aprenderá também como o sistema é limpo depois que um objeto tem sua existência terminada.

```
//=====
```

Criando Objetos.

Como você sabe, classes fornecem o projeto para objetos; você cria um objeto de uma classe. Cada uma das declarações seguintes retiradas do programa `CreateObjectDemo` cria um objeto e atribui ele para uma variável:

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

A primeira linha cria um objeto da classe `Point`, e a segunda e a terceira linhas cria cada uma um objeto da classe `Rectangle`.

Cada uma dessas declarações tem três partes:

- **Declaração:** O código no início de cada linha que associa o nome da variável com o tipo do objeto.
- **Instanciação:** A palavra reservada `new` é um operador Java que cria o objeto.
- **Inicialização:** O operador `new` é seguido por uma chamada para um construtor, que inicializa o novo objeto.

Declarando uma Variável para Fazer Referência a um Objeto.

Primeiramente, você aprendeu que para você declarar uma variável, você escreve:

```
type name;
```

Isto avisa o compilador que você usará `name` para referir-se a um dado que tem o tipo `type`. Com as variáveis primitivas, esta declaração também reserva a quantidade apropriada de memória para a variável.

Você também pode declarar uma referência a uma variável na própria linha. Por exemplo:

```
Point originOne;
```

Se você declarar `originOne` dessa forma, este valor será indeterminado até que um objeto seja criado e atribuído para ele. Simplesmente declarando um referência à variável não cria um objeto. Para isto, você precisa usar o operador `new`, como descrito na seção seguinte. Você precisa nomear um objeto para `originOne` antes de você usá-lo em seu código. Do contrário, haverá um erro de compilação.

Uma variável neste estado, que não referencia em direção a um objeto, pode ser ilustrada como segue (a variável `name`, `originOne`, mais a referência apontando para nenhum lugar).

originOne 

Instanciando uma Classe.

O operador `new` instancia uma classe alocando memória para um novo objeto e retornando a referência para aquela memória. O operador `new` também invoca o construtor do objeto.

Nota: A frase “instanciando uma classe” significa a mesma coisa que “criando um objeto”. Quando você cria um objeto, você está criando uma “instância” da classe, portanto, “instanciando” uma classe.

O operador `new` requer um simples argumento escrito após o mesmo: uma chamada para um construtor. O nome do construtor fornece o nome da classe para instanciar.

O operador `new` retorna uma referência para o objeto que foi criado. Esta referência é usualmente atribuída a uma variável do tipo apropriado, como:

```
Point originOne = new Point(23, 94);
```

A referência retornada pelo operador `new` não tem que ser a atribuição da variável. Ela também pode usar diretamente uma expressão. Por exemplo:

```
int height = new Rectangle().height;
```

Esta declaração será discutida na próxima seção.

Inicializando um Objeto.

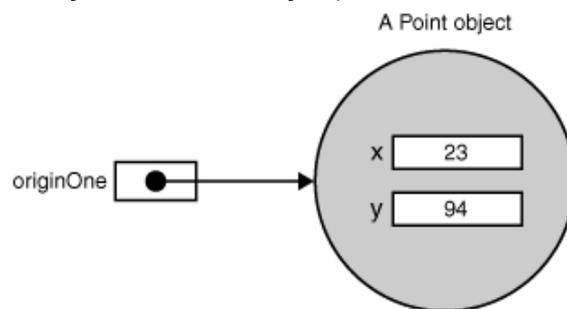
Aqui está um código para a classe `Point`:

```
public class Point {
    public int x = 0;
    public int y = 0;
    //construtor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

Esta classe contém um construtor simples. Você pode identificar o construtor porque esta declaração usa o mesmo nome que a classe e ela não retorna tipo. O construtor na classe `Point` pega dois argumentos inteiro, como declarado pelo código (`int a, int b`). A seguinte declaração fornece os valores 23 e 94 como valores para esses argumentos:

```
Point originOne = new Point(23, 94);
```

O resultado da execução dessa declaração pode ser ilustrado na próxima figura:



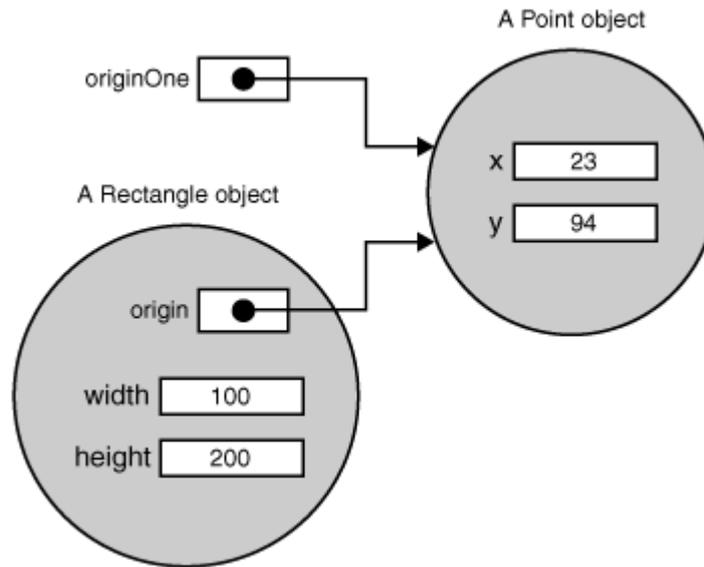
Aqui está o código para a classe `Rectangle`, que contém quatro construtores:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;
    // quatro construtores
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
    // um método para mover o retângulo
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
    // um método para computar a área do retângulo
    public int getArea() {
        return width * height;
    }
}
```

Cada construtor deixa você fornecer valores iniciais para o tamanho e largura do retângulo, usando ambos tipos primitivos e de referência. Se a classe tem múltiplos construtores, eles precisam ter diferentes atribuições. O compilador Java diferencia os construtores baseado no número e no tipo de argumentos. Quando o compilador Java encontra o seguinte código, ele sabe como chamar o construtor na classe `Rectangle` que requer um argumento `Point` seguido por dois argumentos inteiros:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

Isto chama um dos construtores de `Rectangle` que inicializa `origin` para `originOne`. Também, o construtor aponta `width` para 100 e `height` para 200. Agora há duas referências para o mesmo `Point` object – um objeto pode ter múltiplas referências como esta, como mostrado na seguinte figura:



A seguinte linha de código chama o construtor `Rectangle` que requer dois argumentos inteiros, que fornecem os valores iniciais para `width` e `height`. Se você inspecionar o código dentro do construtor, você verá que é criado um novo objeto `Point` onde os valores `x` e `y` são inicializados com 0.

```
Rectangle rectTwo = new Rectangle(50, 100);
```

O construtor `Rectangle` usado na seguinte declaração não pega nenhum argumento, então ele é chamado de **no-argument constructor** (construtor sem argumento).

```
Rectangle rect = new Rectangle();
```

Todas as classes têm no mínimo um construtor. Se a classe não declara nenhum explicitamente, o compilador Java automaticamente fornece um construtor sem argumento, chamado **default constructor** (construtor padrão). Este construtor padrão chama o construtor da classe sem argumento pai, ou o `Object` construtor se a classe não tem outro pai. Se o pai não tem um construtor (`Object` não tem um, o compilador rejeitará o programa).

```
//=====
```

Usando Objetos.

Uma vez que você cria um objeto, você provavelmente quer usá-lo para alguma coisa. Você pode precisar usar o valor de um de seus campos, mudar um desses campos, ou chamar um de seus métodos para executar uma ação.

Referenciando um Campo de Objeto.

Campos de objeto são acessíveis pelo seu nome. Você precisa usar um nome que não seja ambíguo.

Você pode usar um nome simples para um campo dentro de sua classe. Por exemplo, nós podemos adicionar a declaração dentro da classe `Rectangle` que mostra a `width` (largura) e a `height` (altura):

```
System.out.println("Width and height are: " + width + ", " + height);
```

Neste caso, `width` e `height` são nomes simples.

O código que está fora da classe do objeto precisa usar uma referência ao objeto ou expressão, seguida por um operador ponto (`.`), seguido por um nome de campo simples, como em:

```
objectReference.fieldName
```

Por exemplo, o código na classe `CreateObjectDemo` está fora do código da classe `Rectangle`. Então para fazer referência para os campos `origin`, `width`, e `height` no objeto `Rectangle` nomeado `rectOne`, a classe `CreateObjectDemo` precisa usar os nomes `rectOne.origin`, `rectOne.width` e `rectOne.height`, respectivamente. O programa usa dois desses nomes para mostrar a `width` e a `height` de `rectOne`:

```
System.out.println("Width of rectOne: " + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
```

Tentar usar os nomes simples `width` e `height` para o código na classe `CreateObjectDemo` não faz muito sentido – esses campos existem somente dentro do objeto – e resulta em um erro de compilação.

Mais tarde, o programa usa um código similar para mostrar a informação sobre `rectTwo`. Objetos do mesmo tipo têm suas cópias reconhecidas dos mesmos campos da instância. Desse modo, cada objeto `Rectangle` teve seus campos nomeados como `origin`, `width`, e `height`. Quando você acessa uma instância do campo diretamente em uma referência de objeto, você faz referência àquele campo do objeto em particular. Os dois objetos `rectOne` e `rectTwo` no programa `CreateObjectDemo` têm diferentes campos `origin`, `width`, e `height`.

Para acessar um campo, você pode usar uma referência nomeada para um objeto, como no exemplo anterior, ou você pode usar qualquer expressão que retorna uma referência ao objeto. Lembre-se que o operador `new` retorna uma referência para um objeto. Então você poderia usar o valor retornado de `new` para acessar um novo campo do objeto:

```
int height = new Rectangle().height;
```

Esta declaração cria um novo objeto `Rectangle` e imediatamente consegue sua altura. Em essência, a declaração calcula a altura **default** (padrão) de um `Rectangle`. Note que depois da declaração ser executada, o programa não prolonga uma referência para o `Retângulo` criado, porque o programa nunca armazena a referência em nenhum lugar. O objeto perde a referência, e estes recursos estão livres para serem reciclados pela **Java Virtual Machine**.

```
//=====
```

Chamando um Método de Objeto.

Você também usa uma referência a um objeto para invocar um método de objeto. Você junta o nome simples do método à referência ao objeto, com um operador ponto (`.`) intervindo. Além disso, você fornece, dentro de parênteses, qualquer argumento para o método. Se o método não requer nenhum argumento, use parêntese vazios.

```
objectReference.methodName(argumentList);
or
objectReference.methodName();
```

A classe retângulo tem dois métodos: `getArea()` para calcular a área do retângulo e `move()` para mudar a origem do retângulo. Aqui está o código `CreateObjectDemo` que invoca esses dois métodos:

```
System.out.println("Area of rectOne: " + rectOne.getArea());
...
rectTwo.move(40, 72);
```

A primeira declaração invoca o método `getArea()` de `rectOne` e mostra os resultados. A segunda linha move `rectTwo` porque o método `move()` recebe novos valores para o objeto `origin.x` e `origin.y`.

Assim como instância de campos, ***objectReference*** pode ser uma referência a um objeto. Você pode usar um nome variável, mas você também pode usar qualquer expressão que retorna uma referência ao objeto. O operador `new` retorna uma referência ao objeto, então você pode usar o valor retornado de `new` para invocar um novo método de objeto:

```
new Rectangle(100, 50).getArea()
```

A expressão `new Rectangle(100, 50)` retorna uma referência ao objeto que aponta para um objeto `Rectangle`. Como mostrado, você pode usar a notação de ponto (`.`) para invocar um novo método `getArea()` de `Rectangle` para calcular a área de um novo retângulo.

Alguns métodos, como um `getArea()`, retornam um valor. Para métodos que retornam um valor, você pode usar a invocação de métodos em expressões. Você pode atribuir o valor retornado para uma variável, usando os comandos de decisão, ou controlando um ***loop***. Este código atribui o valor retornado por `getArea()` para a variável `areaOfRectangle`:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Relembre, invocando um método em um objeto em particular, é o mesmo que mandar uma mensagem para aquele objeto. Neste caso, o objeto `getArea()` que é invocado é o retângulo retornado pelo construtor.

```
//=====
```

O Garbage Collector (Coletor de Lixo de Memória).

Algumas linguagens de orientação a objeto requerem que você mantenha rastros para todos os objetos que você cria e que você destrua explicitamente eles quando não são mais necessários. Gerenciamento de memória manual é tedioso e passível de erros. A plataforma Java permite a você criar quantos objetos você quiser (limitados, é claro, pelo que seu sistema pode controlar), e você não tem que se preocupar em destruí-los. O ambiente Java de ***runtime*** (tempo de execução) deleta objetos quando está determinado que eles não serão mais usados. Esse processo é chamado de ***garbage collection***.

Um objeto é destinado para o ***garbage collection*** quando não há mais referências para aquele objeto. Referências que são mantidas em uma variável são usualmente pingadas quando a variável sai fora do escopo. Ou, você pode explicitamente pingar uma referência a um objeto indicando a variável para o valor especial `null`. Relembre que o programa pode fazer múltiplas referências para o mesmo objeto; todas as referências para um objeto precisam ser pingadas antes que o objeto seja destinado para o ***garbage collector***.

O ambiente Java de ***runtime*** (tempo de execução) tem um ***garbage collector*** que periodicamente libera a memória usada pelos objetos que não são mais referenciados. O ***garbage collector*** não faz seu trabalho automaticamente quando é determinado que o tempo é prioridade.

Mais Sobre Classes.

Retornando um valor de um Método.

Um método retorna para o código que o invocou quando ele:

- completa todas as declarações no método,
- atinge a declaração `return`,
- lança uma exceção (assunto coberto mais adiante),

seja qual for que ocorra primeiro.

Você declara um tipo de retorno do método na declaração do método. Dentro do corpo do método, você usa a declaração `return` para retornar o valor.

Qualquer método declarado como `void` não retorna um valor. Ele não precisa conter uma declaração `return`, mas pode fazer dessa maneira. Em qualquer caso, uma declaração `return` pode ser usada para ramificar o bloco de controle de fluxo e sair do método e isso é feito simplesmente assim:

```
return
```

Se você tentar retornar um valor de um método que é declarado `void`, você conseguirá um erro de compilação.

Qualquer método que não é declarado `void` pode conter uma declaração `return` com o correspondente valor de retorno, dessa forma:

```
return returnValue;
```

O tipo de dado de um valor retornado precisa coincidir com o tipo de retorno declarado no método; você não pode retornar um valor inteiro de um método declarado pra retornar um `boolean`.

O método `getArea()` na classe `Rectangle` que foi discutida na seção sobre objetos retorna um inteiro:

```
// um método para calcular a área de um retângulo
public int getArea() {
    return width * height;
}
```

Este método retorna o inteiro que a expressão `width * height` calcula.

O método `area` retorna um tipo primitivo. Um método pode também retornar uma referência ao tipo. Por exemplo, no programa para manipular objetos `Bicycle`, nós poderíamos ter um método como este:

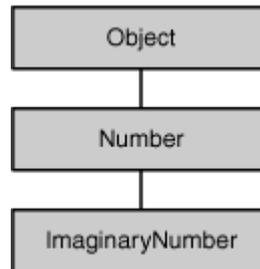
```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle your Bike,
                             Environment env) {
    Bicycle fastest;
    // código para calcular qual bike é mais rápida, dando
    // a cada bike uma marcha e cadência e dando
    // o ambiente (terreno e vento)
    return fastest;
}
```

```
//=====
```

Retornando uma Classe ou Interface.

Se esta seção confundir você, pule e retorne para ela depois que você tiver terminado a lição sobre interfaces e herança.

Quando um método usa um nome de classe como o tipo de retorno, como em `whosFastest` faz, o tipo da classe do objeto retornando precisa ser também uma subclasse, ou a classe exata do tipo de retorno. Supondo que você tenha uma hierarquia de classe em que `ImaginaryNumber` é uma subclasse de `java.lang.Number`, que está a serviço como uma subclasse de `Object`, como ilustrado na seguinte figura:



Agora suponha que você tenha um método declarado para retornar um `Number`:

```
public Number returnANumber() {
    ...
}
```

O método `ReturnANumber` pode retornar um `ImaginaryNumber` mas não um `Object`. `ImaginaryNumber` é um `Number` porque ele é uma subclasse de `Number`. Porém, um `Object` não é necessariamente um `Number` – ele poderia ser uma `String` ou outro tipo.

Você pode passar por cima de um método e defini-lo para retornar uma subclasse do método original, dessa forma:

```
public ImaginaryNumber returnANumber() {
    ...
}
```

Esta técnica, chamada tipo de **retorno co-variante**, significa que o tipo de retorno é autorizado a variar na mesma direção que a subclasse.

Nota: Você também pode usar nomes de interface como tipo de retorno. Neste caso, o objeto retornado precisa implementar a interface especificada.

//=====

Usando a Palavra Reservada `this`.

Dentro de um método da instância ou um construtor, `this` é uma referência para o objeto corrente – o objeto no qual o método ou construtor está sendo chamado. Você pode fazer referência para qualquer membro do objeto corrente de dentro de um método da instância ou um construtor usando `this`.

Usando `this` como um Campo.

A razão mais comum para o uso da palavra reservada `this` é porque o campo é sombreado por um método ou parâmetro do construtor.

Por exemplo, a classe `Point` foi escrita dessa forma:

```
public class Point {
    public int x = 0;
    public int y = 0;

    // construtor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

mas poderia ser escrita dessa forma:

```
public class Point {
    public int x = 0;
    public int y = 0;

    // construtor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Cada argumento para o segundo construtor sombreada um dos campos do objeto – dentro do construtor o `x` é uma cópia local do primeiro argumento do construtor. Para fazer referência ao campo `x` de `Point`, o construtor precisa usar `this.x`.

Usando `this` como um Construtor.

Para escrever um construtor, você pode também usar a palavra reservada `this` para chamar outro construtor na mesma classe. Fazendo dessa forma é chamado de um **invocação explícita do construtor**. Aqui está uma classe `Rectangle`, com uma implementação diferente daquela na seção `Objects`:

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

Esta classe contém um conjunto de construtores. Cada construtor inicializa algumas ou todas as variáveis membros do retângulo. Os construtores fornecem um valor **default** (padrão) para qualquer membro variável cujo valor inicial não é provido por um argumento. Por exemplo, o construtor sem argumento chama os quatro argumentos do construtor com quatro valores `0` e dois argumentos do construtor chamam os quatro argumentos do construtor com dois valores `0`. O compilador determina qual construtor chamar, baseado no número de no tipo de argumentos.

Se presente, a invocação de qualquer construtor precisa ser a primeira linha no construtor.

```
//=====
```

Controlando o Acesso a Membros de uma Classe.

Modificadores de estágios de acesso determinam se outra classe pode usar um campo em particular ou invocar um método em particular. Há dois estágios de controle de acesso;

- No estágio mais acima – **public**, ou **package-private** (sem modificador explícito).
- No estágio do membro – **public**, **private**, **protected**, ou **package-private** (sem modificador explícito).

Uma classe pode ser declarada com o modificador **public**, nesse caso a classe é visível para todas as classes em qualquer lugar. Se a classe não tem modificador (o **default**, também conhecido como **package-private**), é visível somente dentro do próprio pacote (pacotes são nomeados em grupo de classes relacionadas – você aprenderá a respeito deles na próxima lição).

No estágio do membro, você pode também usar o modificador **public** ou o não-modificador (**package-private**) somente com as classes de estágio mais acima, e com o mesmo significado. Para membros, há dois modificadores de acesso adicionais: **private** e **protected**. O modificador **private** especifica que o membro pode somente ser acessado em sua própria classe. O modificador **protected** especifica que o membro somente pode ser acessado dentro de seu próprio pacote (assim como **package-private**) e, em adição, por uma subclasse de sua classe em qualquer pacote.

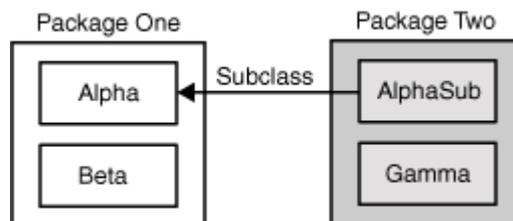
A seguinte tabela mostra o acesso a membros permitido por cada modificador:

Níveis de Acesso				
Modificador	Classe	Pacote	Subclasse	Qualquer lugar
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

A primeira coluna de dados indica que a classe tem acesso a si mesma para o membro definido no estágio de acesso. Como você pode ver, a classe sempre tem acesso para seus próprios membros. A segunda coluna indica que a classe no mesmo pacote que o da classe (indiferente de seu parentesco) tem acesso para o membro. A terceira coluna indica que subclasses das classes – declaradas fora deste pacote – tem acesso para o membro. A quarta coluna indica que todas as classes tem acesso ao membro.

Níveis de acesso afetam você em duas situações. Primeiro, quando você usa classes que vêm de outro código, como no caso das classes da plataforma Java, níveis de acesso determinam quais membros dessas classes suas próprias classes podem usar. Segundo, quando você escreve uma classe, você precisa decidir qual nível de acesso cada membro variável e cada método em sua classe terá.

Dê uma olhada em uma coleção de classes e veja como os níveis de acesso afetam sua visibilidade. A seguinte figura mostra quatro classes neste exemplo e como elas estão relacionadas.



A seguinte tabela mostra onde os membros da classe Alpha são visíveis para cada modificador de acesso que pode ser aplicado a ela.

Visibilidade				
Modificador	Alpha	Beta	Alphasub	Gama
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<code>no modifier</code>	Y	Y	N	N
<code>private</code>	Y	N	N	N

Tipos na Escolha de um Nível de Acesso: os outros programadores usam sua classe, você quer se assegurar que erros provocados pelo abuso não aconteçam. Níveis de acesso podem ajudar você a fazer isso:

- Use o nível de acesso mais restritivo que fizer sentido para um membro em particular. Use `private` a menos que você tenha uma boa razão para não fazê-lo.
- Evite campos `public` exceto para constantes. (Muitos dos exemplos neste tutorial usam campos públicos. Isto ajuda muito a ilustrar alguns pontos concisamente, mas não é recomendado para produção de código). Campos públicos tendem a conectar você para uma implementação em particular e limitam sua flexibilidade nas mudanças em seu código.

//=====

Compreendendo Instância e Membros de Classe.

Nesta seção, nós discutiremos o uso da palavra reservada `static` para criar campos e métodos que pertencem à classe, ao invés de uma instância da classe.

Variáveis de Classe.

Quando um número de objetos é criado do mesmo projeto da classe, eles têm cada um sua cópia distinta das variáveis da instância. No caso da classe `Bicycle`, as variáveis da instância são `cadence`, `gear`, e `speed`. Cada objeto `Bicycle` tem seus próprios valores para estas variáveis, armazenados em diferentes locais de memória.

Algumas vezes, você quer variáveis que sejam comuns para todos os objetos. Elas são acompanhadas pelo modificador `static`. Campos que têm o modificador `static` em suas declarações são chamados de campos estáticos ou variáveis de classe. Eles são associados com a classe, ao invés de com um objeto qualquer. Todas as instâncias da classe compartilham a variável de classe, que está em um local fixado na memória. Qualquer objeto pode mudar o valor de uma variável de classe, mas variáveis de classe podem também ser manipuladas sem a criação de uma instância da classe.

Por exemplo, suponha que você quer criar um número de objetos `Bicycle` e atribuir a cada um um número serial, começando com 1 para o primeiro objeto. Este número identificador é único para cada objeto e por essa razão está em uma variável de instância. Ao mesmo tempo, você precisa de um campo para manter um rastro de como muitos objetos `Bicycle` foram criados para que você saiba qual identidade atribuir para a próxima. Dessa maneira o campo não é relatado para qualquer objeto individual, mas para a classe como um todo. Por isso, você precisa de uma variável de classe, `numberOfBicycles`, como segue:

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    // adiciona uma variável de instância para a ID do objeto
    private int id;
    // adiciona uma variável de classe para o
    // número de objetos Bicycle instanciados
    private static int numberOfBicycles = 0;
    .....
}
```

Variáveis de classe são referenciadas pelo nome da própria classe, como em

```
Bicycle.numberOfBicycles
```

Desse modo mostra claramente que elas são variáveis da classe.

Nota: Você pode também fazer referência a campos estáticos com uma referência ao objeto, como em:

```
myBike.numberOfBicycles
```

mas isto é desencorajado porque não mostra claramente que se trata de variáveis de classes.

Você pode usar o construtor `Bicycle` para apontar a variável de instância `id` e incrementar a variável de classe `numberOfBicycles`:

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        // incrementa o número de Bicycles e atribui um número ID
        id = ++numberOfBicycles;
    }
    // novo método para retornar a variável de instância ID
    public int getID() {
        return id;
    }
    .....
}
```

Métodos de Classe.

A linguagem de programação Java suporta métodos estáticos assim como variáveis estáticas. Métodos estáticos, os quais têm o modificador `static` em suas declarações, precisam ser invocados com o nome da classe, sem a necessidade da criação de uma instância da classe, como em:

```
ClassName.methodName(args)
```

Nota: Você pode também fazer referência a métodos estáticos com uma referência ao objeto, como:

```
instanceName.methodName(args)
```

mas isso é desencorajado porque não mostra claramente que eles são métodos de classe.

Um uso comum para os métodos estáticos é para acessar campos estáticos. Por exemplo, nós poderíamos adicionar um método estático para a classe `Bicycle` para acessar o campo estático `numberOfBicycles`:

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

Nem todas as combinações de variáveis de instância de classe e métodos são permitidos:

- Métodos da instância podem acessar variáveis de instância e métodos de instância diretamente.
- Métodos da instância podem acessar variáveis de classe e métodos da classe diretamente.
- Métodos de Classe podem acessar variáveis de classe e métodos da classe diretamente.
- Métodos da Classe **não podem** acessar variáveis de instância ou métodos de instância diretamente – eles precisam usar uma referência ao objeto. Além disso, métodos de classe **não podem** usar a palavra reservada `this` porque ali não há instância para fazer referência para `this`.

Constantes.

O modificador `static`, em combinação com o modificador `final`, é também usado para definir constantes. O modificador `final` indica que o valor do campo não pode mudar.

Por exemplo, a seguinte declaração de variável define uma constante nomeada `PI`, cujo valor é uma aproximação de pi (o raio de uma circunferência de um círculo para este diâmetro):

```
static final double PI = 3.141592653589793;
```

Constantes definidas dessa forma não podem ser modificadas, e ela vai gerar um erro de tempo de compilação se seu programa tenta fazer isso. Por convenção, o nome da constante de valores é escrito em letras maiúsculas. Se o nome é composto por mais de uma palavra, as palavras são separadas por um underscore (`_`).

Nota: Se um tipo primitivo ou uma string é definida como uma constante e o valor é conhecido em tempo de compilação, o compilador substitui o nome da constante em todos os lugares no código com este valor. Isto é chamado de **compile-time constant** (constante em tempo de compilação). Se o valor da constante no lado de fora muda (por exemplo, se for legislado que pi atualmente será 3.975), você precisará recompilar qualquer classe que use a constante para mudar o valor corrente.

A Classe Bicycle.

Depois de todas as modificações feitas nesta seção, a classe `Bicycle` agora é:

```
public class Bicycle{
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;
    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
        id = ++numberOfBicycles;
    }
    public int getID() {
        return id;
    }
    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }
    public int getCadence(){
        return cadence;
    }
    public void setCadence(int newValue){
        cadence = newValue;
    }
    public int getGear(){
        return gear;
    }
    public void setGear(int newValue){
        gear = newValue;
    }
    public int getspeed(){
        return speed;
    }
    public void applyBrake(int decrement){
        speed -= decrement;
    }
    public void speedUp(int increment){
        speed += increment;
    }
}
```

Inicializando Campos.

Como você viu, você geralmente pode fornecer um valor inicial para um campo como nesta declaração:

```
public class BedAndBreakfast {
    public static int capacity = 10; // inicializa para 10
    public boolean full = false; // inicializa para false
}
```

Isso funciona também quando o valor calculado e a inicialização podem ser colocados em uma linha. No entanto esta forma de inicialização tem suas limitações por causa de sua simplicidade. Se a inicialização requer alguma lógica (por exemplo, tratamentos de erros ou um **loop for** para abastecer um **array** complexo), atribuições simples são inadequadas. Variáveis de instância podem ser inicializadas em construtores, onde tratamentos de erros ou outra lógica podem ser usados. Para fornecer a mesma capacidade para variáveis de classe, a linguagem de programação Java inclui blocos de inicialização estáticos.

Nota: Não é necessário declarar campos no início da definição da classe, embora isso seja a prática mais comum. Somente é necessário que eles sejam declarados e inicializados antes de serem usados.

Blocos de Inicialização Estáticos.

Um bloco de inicialização estático é um bloco normal de código envolvido por chaves, { }, e precedido pela palavra reservada **static**. Aqui está um exemplo:

```
static {
    // qualquer código que for necessário para inicialização vai aqui
}
```

Uma classe pode ter qualquer número de blocos de inicialização estáticos, e eles podem aparecer em qualquer lugar no corpo da classe. O sistema de **runtime** garante que os blocos de inicialização estáticos serão chamados na ordem em que eles aparecem no código fonte.

Há uma alternativa para blocos estáticos – você pode escrever um método estático privativo:

```
class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {
        // código de inicialização vai aqui
    }
}
```

A vantagem de métodos estáticos privados é que eles podem ser reutilizados mais tarde se você precisar reinicializar a variável da classe.

//=====

Inicializando Membros da Instância.

Normalmente, você colocaria o código para inicializar uma variável da instância em um construtor. Há duas alternativas para o uso de um construtor para inicializar as variáveis da instância: blocos de inicialização e métodos finais.

Blocos de inicialização para variáveis da instância parecem com os blocos estáticos de inicialização, mas sem a palavra reservada `static`:

```
{
    // qualquer código necessário para a inicialização vai aqui
}
```

O compilador Java copia blocos de inicialização em cada construtor. Por essa razão, este caminho pode ser usado para compartilhar blocos de código entre múltiplos construtores.

Um método `final` não pode passar por cima em uma subclasse. Isto será discutido na lição sobre interfaces e herança. Aqui está um exemplo do uso de um método `final` para inicialização de uma variável da instância:

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // código de inicialização vai aqui
    }
}
```

Isto é especialmente útil se as subclasses podem querer reusar o método de inicialização. O método é `final` porque chamar métodos não finais durante a inicialização da instância pode causar problemas.

```
//=====
```

Classes Aninhadas.

A linguagem de programação Java permite a você definir uma classe dentro de outra classe. Desta maneira, uma classe é chamada de classe aninhada e é ilustrada aqui:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Uma classe aninhada é um membro que está envolvendo uma classe e, como tal, tem acesso a outros membros da classe envolvida, mesmo que eles sejam declarados privados. Como um membro de `OuterClass`, uma classe aninhada pode ser declarada `private`, `public`, `protected`, ou `package private`. (Relembre que outras classes só podem ser declaradas como `public` ou `package private`).

Terminologia: Classes aninhadas são divididas em duas categorias: estáticas e não estáticas. Classes aninhadas que são declaradas `static` são simplesmente chamadas **static nested classes** (classes aninhadas estáticas). Classes aninhadas não estáticas são chamadas **inner classes** (classes internas).

```
Class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

//=====

Porque Usar Classes Aninhadas?

Há muitos motivos que obrigam o uso de classes aninhadas, entre eles:

- Este é um caminho para logicamente agrupar classes que são usadas somente em um lugar.
- Isto aumenta o encapsulamento.
- Classes Aninhadas podem tornar o código mais legível e fácil de consertar.

Agrupamento lógico de classes – Se a classe é útil para somente uma outra classe, então é mais lógico embuti-la naquela classe e manter as duas juntas. Aninhamentos como “classes de ajuda” fazem seus pacotes mais eficientes.

Aumento do encapsulamento – Considere duas classes de alto nível, A e B, onde B precisa acessar os membros de A que no entanto foram declarados `private`. Colocando a classe B dentro da classe A, os membros de A podem ser declarados `private` e B pode acessá-los mesmo assim. Além disso, B fica escondido do resto do mundo.

Código mais legível e fácil de manter – Aninhando classes pequenas dentro de classes de alto nível posiciona o fechamento do código onde ele é usado.

//=====

Classes Aninhadas Estáticas.

Assim como métodos e variáveis da classe, uma classe aninhada está associada com esta outra classe. E como métodos da classe estática, uma classe aninhada estática não pode fazer referência diretamente para as variáveis da instância ou métodos definidos nesta classe envolvida – ela pode usá-la somente usando uma referência ao objeto.

Nota: Uma classe aninhada estática interage com os membros da instância da outra classe (e outras classes) somente com qualquer outra classe de alto nível semelhante. Com efeito, uma classe aninhada estática tem um comportamento de uma classe de alto nível que foi aninhada em outra classe de alto nível por conveniência de acondicionamento.

Classes aninhadas estáticas são acessadas usando o nome da classe envolvida:

```
OuterClass.StaticNestedClass
```

Por exemplo, para criar um objeto para a classe aninhada estática, use esta sintaxe:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();  
  
//=====
```

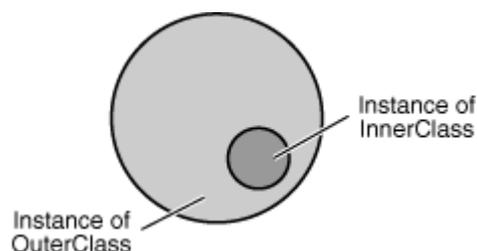
Classes Internas.

Assim como métodos e variáveis da instância, uma classe interna está associada com uma instância se estiver envolvendo a classe e tem acesso direto para aqueles métodos e campos do objeto. Além disso, pelo motivo de uma classe interna estar associada com uma instância, ela não define qualquer membro estático nela mesma.

Objetos que são instanciados de uma classe interna existem dentro dos limites da instância da outra classe. Considere a seguinte classe:

```
class OuterClass {  
    ...  
    class InnerClass {  
        ...  
    }  
}
```

Uma instância de `InnerClass` pode existir somente dentro da instância de `OuterClass` e tem direito ao acesso para os métodos e campos da instância que está envolvendo. A próxima figura ilustra a idéia:



Para instanciar uma classe interna, você precisa primeiro instanciar a outra classe. Então, crie o objeto dentro de outro objeto com a sintaxe:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Adicionalmente, há dois tipos especiais de classes internas: classes locais e classes anônimas (também chamadas classes internas anônimas). Ambas serão discutidas resumidamente no próxima seção.

Exemplo de Classe Aninhada.

Para ver uma classe interna em uso, considere um simples pilha de inteiros. Pilhas, que são tipos comuns de dados estruturados em programação, são como pilhas de recipientes. Quando você adiciona um recipiente para a pilha, você o coloca no topo; quando você remove um, você o remove do topo. O acrônimo para isto é **LIFO (last in, first out)**. Recipientes na parte mais baixa da pilha podem ficar lá por um longo tempo enquanto a parte superior dos recipientes vem e vão.

A classe `StackOfInts` a seguir é implementada como um **array**. Quando você adiciona um inteiro (chamado '**pushing**'), ele vai para o primeiro elemento disponível vazio. Quando você remove um inteiro (chamado "**popping**"), você remove o último inteiro no **array**.

A classe `StackOfInts` a seguir (uma aplicação) consiste de:

- A classe exterior, `StackOfInts`, que inclui métodos para empurrar um inteiro na pilha, disparar um inteiro cancelado na pilha, e testa para ser se a pilha está vazia.
- A classe interna `StepThrough`, que é similar a um iterador padrão Java. Iteradores são usados para percorrer por completo uma estrutura de dados e tipicamente têm métodos para testar pelo último elemento, recuperar o elemento atual, e mover para o próximo elemento.
- Um método `main` que instancia um **array** `StackOfInts` (`StackOne`) e abastece ele com inteiros (`0, 2, 4`, etc.), então instancia um objeto `StepThrough` (`iterator`) e o usa para mostrar a saída dos argumentos de `stackOne`.

Na próxima página você tem a aplicação completa:

```

public class StackOfInts {
    private int[] stack;
    private int next = 0; // indexa o último item em stack + 1
    public StackOfInts(int size) {
        // cria um array grande o suficiente para receber a pilha
        stack = new int[size];
    }
    public void push(int on) {
        if(next < stack.length)
            stack[next++] = on;
    }
    public boolean isEmpty() {
        return (next == 0);
    }
    public int pop() {
        if (!isEmpty())
            return stack[--next]; // item no topo da lista
        else
            return 0;
    }
    public int getStackSize() {
        return next;
    }
    private class StepThrough {
        // inicia a ação de percorrer de uma parte a outra com i=0
        private int i = 0;

        // incrementa o índice
        public void increment() {
            if (i < stack.length)
                i++;
        }
        // recupera o elemento corrente
        public int current() {
            return stack[i];
        }
    }

    // último elemento na pilha?
    public boolean isLast() {
        if(i == getStackSize() - 1)
            return true;
        else
            return false;
    }
}
public StepThrough stepThrough() {
    return new StepThrough();
}
public static void main(String[] args) {
    // instancia classe exterior como "stackOne"
    StackOfInts stackOne = new StackOfInts(15);
    // povoa stackOne
    for (int j = 0; j < 15; j++) {
        stackOne.push(2 * j);
    }
    // instancia classe interna com "iterator"
    StepThrough iterator = stackOne.stepThrough();
    // mostra stackOne[i], um por linha
    while(!iterator.isLast()) {
        System.out.println(iterator.current() + " ");
        iterator.increment();
    }
    System.out.println();
}
}

```

A saída é:

```
0 2 4 6 8 10 12 14 16 18 20 22 24 26
```

Note que a classe `StepThrough` faz referência diretamente para a instância da pilha variável de `StackOfInts`.

Classes internas são usadas principalmente para implementar classes *helper* (de ajuda) como a mostrada no exemplo. Se você planeja a manipulação de eventos usuário-interface, você precisará aprender a conhecer a respeito delas usando classes internas porque os mecanismos de eventos manuais fazem uso intensivo delas.

//=====

Classes Internas Locais e Anônimas.

Há dois tipos adicionais de classes internas. Você pode declarar uma classe interna dentro do corpo de um método. Esta classe é conhecida como uma *classe interna local*. Você também pode declarar uma classe interna dentro do corpo do método sem nomeá-la. Estas classes são conhecidas como *classes internas anônimas*. Você encontrará classes semelhantes na programação avançada Java.

//=====

Modificadores.

Você pode usar os mesmos modificadores para classes internas que você usa para outros membros de classes externas. Por exemplo, você pode usar os especificadores de acesso – `private`, `public`, e `protected` – para restringir o acesso a classes internas, da mesma forma que você faz com outros membros de classes.

//=====

Tipos Numeráveis.

Um tipo numerável é um tipo onde os campos consistem de um conjunto fixo de constantes. Exemplos comuns incluem direções circulares (valores de **NORTH**, **SOUTH**, **EAST**, e **WEST**) e os dias da semana.

Porque eles são constantes, os nomes dos campos de um tipo numerável são letras maiúsculas.

Na linguagem de programação Java, você define um tipo numerável usando a palavra reservada **enum**. Por exemplo, você pode especificar um tipo numerável com os dias da semana:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Você pode usar tipos enumeráveis a qualquer momento que você precisar para representar um conjunto fixo de constantes. Isto inclui tipos enumeráveis naturais como os planetas em nosso sistema solar e conjunto de dados onde você conhece todos os possíveis valores em tempo de compilação – por exemplo, as escolhas de um menu, marcadores de linha de comando, e assim por diante.

Aqui está um código que mostra como usar enumerável **Day** definido acima:

```
public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY: System.out.println("Monday are bad.");
                break;
            case FRIDAY: System.out.println("Fridays are better.");
                break;
            case SATURDAY:
            case SUNDAY: System.out.println("Weekends are best.");
                break;
            default: System.out.println("Midweek days are so-so.");
                break;
        }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

A saída é:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

//=====

Os tipos enumeráveis da linguagem de programação Java são muito mais poderosos que seus correlatos em outras linguagens. A declaração `enum` define uma classe (chamada `enum type`). O corpo da classe `enum` pode incluir métodos e outros campos. O compilador automaticamente adiciona alguns métodos especiais quando é criado um `enum`. Por exemplo, eles têm um método de valor `static` que retorna um `array` contendo todos os valores de `enum` na ordem em que eles foram declarados. Este método é comumente usado em combinação com o construtor `for-each` para iterar completamente os valores de um tipo `enum`. Por exemplo, este código da classe exemplo `Planet` abaixo itera completamente todos os planetas no sistema solar.

```
for (Planet p: Planet.values()) {
    System.out.println("Your weight on %s is %f%n",
        p, p.surfaceWeight(mass));
}
```

Nota: Todos os `enums` implicitamente estendem `java.lang.Enum`. Desde que Java não suporta múltiplas heranças, um `enum` não pode estender mais alguma coisa. Em adição para as propriedades do construtor, `Planet` tem métodos que permitem a você recuperar a superfície de gravidade e peso de um objeto em cada planeta.

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURN (5.688e+26, 6.0268e7),
    URANUS (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);
    private final double mass; // em Kilogramas
    private final double radius; // em metros
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }
    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;
    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Your weight on %s is %f%n",
                p, p.surfaceWeight(mass));
    }
}
```

Se você rodar `Planet.class` da linha de comando com um argumento de 175 você conseguirá a saída:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

Anotações.

Anotações fornecem dados a respeito do programa que não fazem parte do programa em si. Elas não têm efeito direto na operação do código ao qual elas comentam.

Anotações tem uma variedade de usos, como estes:

- **Informação para o compilador** – Anotações podem ser usadas pelo compilador para detectar erros ou suprimir advertências.
- **Processamento de tempo de compilação e tempo de desenvolvimento** – Ferramentas de software podem processar informações de anotações para gerar código, arquivos **XML**, e assim por diante.
- **Processamento *runtime* (tempo de execução)** – Algumas anotações são disponibilizadas para serem examinadas em tempo de execução.

Anotações podem ser aplicadas para as declarações de classes, campos e métodos, além de outros elementos do programa.

A anotação aparece primeiro, freqüentemente (por convenção) na própria linha, e pode incluir elementos com valores nomeados ou não:

```
@Author(
    name = "Benjamin Franklin",
    date = "3/27/2003"
)
class MyClass() { }
```

OU

```
@SuppressWarnings(value = "unchecked")
void myMethod() { }
```

Se há somente um elemento nomeado "**value**", então o nome pode ser omitido, como em:

```
@Override
void mySuperMethod() { }
```

Documentação.

Muitas anotações repõem o que de outra maneira seria comentado no código.

Supondo que um grupo de software tem tradicionalmente começado o corpo de cada classe com comentários fornecendo importantes informações:

```
public class Generation3List extends Generation2List {

    // Author: John Doe
    // Date: 3/17/2002
    // Current revision: 6
    // Last modified: 4/12/2004
    // By: Jane Doe
    // Reviewers: Alice, Bill, Cindy

    // class code goes here

}
```

Para adicionar estes mesmos metadados como uma anotação, você precisa primeiro definir o tipo de anotação. A sintaxe para fazer isso é:

```
@interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
    String[] reviewers(); // Note use of array
}
```

O tipo de anotação parece um pouco como uma definição de interface onde a palavra reservada `interface` é precedida pelo caracter `@` (`@` = “**AT**” como em *Annotation Type*). Tipos de anotação são, de fato, uma forma de interface, que será coberta em uma lição mais tarde.

O corpo da definição da anotação acima contém declarações de elementos do tipo anotação, que parecem uma porção de métodos. Note que eles podem ser também definidos com valores opcionais padrão.

Uma vez que o tipo de anotação foi definido, você pode usar anotações de qualquer tipo, com os valores acumulados, como em:

```
@ClassPreamble (
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe"
    reviewers = {"Alice", "Bob", "Cindy"} // Note array notation
)
public class Generation3List extends Generation2List {

    // class code goes here

}
```

Nota: Para fazer a informação em `@ClassPreamble` aparecer no gerador de documentação Javadoc, você precisa anotar a definição da `@ClassPreamble` com a anotação `@Documented`:

```
import java.lang.annotation.*; // import this to use @Documented

@Documented
@interface ClassPreamble {

    // Annotation element definitions

}
```

Anotações Usadas pelo Compilador.

Há três tipos de anotações que são pré-definidas pela auto-especificação da linguagem: `@Deprecated`, `@Override`, e `@SuppressWarnings`.

@Deprecated – a anotação `@Deprecated` indica que o elemento marcado é depreciado e mostra que será pouco usado. O compilador gera uma advertência sempre que o programa usa o método, classe, ou campo com a anotação `@Deprecated`. Quando um elemento é depreciado, também pode ser documentado usando a tag **Javadoc** `@deprecated`, como mostrado no exemplo seguinte. O uso do símbolo “`@`” em ambos, comentários **Javadoc** e anotações não é coincidência – eles são relatados conceitualmente. Além disso, note que a **tag Javadoc** começa com a letra minúscula “`d`” e a anotação começa com a letra maiúscula “`D`”.

```
// Javadoc comment follows
/**
 * @deprecated
 * explanation of why it was deprecated
 */
@Deprecated
static void deprecatedMethod() { }
}
```

@Override – a anotação `@Override` informa ao compilador que o elemento está destinado para passar por cima de um elemento declarado em uma superclasse (passar por cima de métodos será discutido na lição “Interfaces e Herança”).

```
// mark method as a superclass method
// that has seen overridden
@Override
int overriddenMethod() { }
```

Embora não seja necessário usar esta anotação quando passando por cima de um método, isto ajuda a prevenir erros. Se o método marcado com `@Override` falha para corretamente passar por cima de um método em uma de suas superclasses, o compilador gera um erro.

@SuppressWarnings – a anotação `@SuppressWarnings` diz ao compilador para suprimir advertências específicas que ele geraria caso contrário. No exemplo seguinte, um método depreciado é usado e o compilador normalmente geraria uma advertência. Neste caso, no entanto, a anotação causa a supressão da advertência.

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
    objectOne.deprecatedMethod(); //deprecation warning - suppressed
}
```

Cada mensagem do compilador tem seu lugar próprio para uma categoria. A especificação da linguagem Java lista duas categorias: “*deprecation*” (depreciação) e “*unchecked*” (não checado). A advertência “*unchecked*” pode ocorrer quando interagindo com código legado escrito antes do advento dos genéricos (assunto discutido na lição intitulada “Genéricos”). Para suprimir mais de uma categoria de advertências, use a seguinte sintaxe:

```
@SuppressWarnings({ "unchecked", "deprecation" })
```

Processando Anotações.

O mais avançado uso de anotações inclui escrever um processamento de anotações que pode ler um programa Java e realizar ações baseado nestas anotações. Isso pode, por exemplo, gerar um código fonte auxiliar, aliviando o programador de ter que criar códigos *boilerplate*, que sempre seguem exemplos prognosticados. Para facilitar este trabalho, a *release* 5.0 do *JDK* inclui uma ferramenta de processamento de anotações, chamada `apt`. Na *release* 6 do *JDK*, a funcionalidade de `apt` é uma parte padrão do compilador Java.

Para fazer informações de anotação disponíveis em tempo de execução, o tipo da anotação precisa ele mesmo ser anotado com:

```
@Retention(RetentionPolicy.RUNTIME), as follows:
```

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {

    // Elements that give information
    // for runtime processing

}
```

```
//=====
```

Interfaces.

Há um número de situações em engenharia de software quando é importante para grupos distintos de programadores concordarem um “contrato” que explique nos mínimos detalhes como seus softwares interagem. Cada grupo tem habilidade para escrever seu código sem conhecimento de como outros grupos escrevem os seus. Genericamente falando, **interfaces** são esses contratos.

Por exemplo, imagine uma sociedade futurista onde carros robôs controlados por computadores transportam passageiros através das ruas da cidade sem um operador humano. Fabricantes automobilísticos escrevem software (Java, é claro) que operam os automóveis – parar, acelerar, virar à esquerda, e assim por diante. Outros grupos industriais, fabricantes de instrumentos eletrônicos de orientação, fazem sistemas de computador que recebem dados de posição **GPS (Global Positioning Satellite)** e transmitem sem fios as condições de tráfego e usam esta informação para dirigir o carro.

Os fabricantes de automóveis precisam publicar uma interface padrão industrial que explica nos mínimos detalhes quais métodos podem ser invocados para fazer o carro mover (qualquer carro, de qualquer fabricante). Os fabricantes de instrumentos podem então escrever softwares que invocam os métodos descritos na interface para comandar os carros. Nenhum grupo industrial precisa saber como o software do outro grupo é implementado. De fato, cada grupo considera este software altamente proprietário e reserva o direito de modificá-lo a qualquer tempo, contanto que continue a ficar fiel à interface publicada.

Interfaces em Java.

Na linguagem de programação Java, uma interface é um tipo de referência, similar a uma classe, que pode conter somente constantes, marcação de métodos, e tipos aninhados. Não há corpos de métodos. Interfaces não podem ser instanciadas – elas podem somente ser implementadas pelas classes ou estendidas (**extended**) por outras interfaces. Extensão será discutida mais tarde nesta lição.

Definir uma interface é similar à criação de uma nova classe:

```
public interface OperateCar {
    // declarações de constantes, talvez nenhuma
    // assinaturas dos métodos
    int turn(Direction direction, // Como enum com valores RIGHT, LEFT
             double radius, double startSpeed, double endSpeed);
    int chageLanes(Direction direction, double startSpeed,
                  double endSpeed);
    int signalTurn(Direction direction, boolean signalOn);
    int gerRadarFront(double distanceToCar, double speedOfCar);
    int getRadarRear(double distanceToCar, double speedOfCar);
    .....
    // mais assinaturas de métodos
}
```

Note que a assinatura do método não tem chaves e são terminadas por um ponto-e-vírgula.

Para usar uma interface, você escreve uma classe que implementa uma interface. Quando uma classe instanciável implementa uma interface, ela fornece um corpo de método para cada um dos métodos declarados na interface. Por exemplo,

```
public class OperadteBMW760i implements OperateCar {
    // o método OperateCar assinalado, com implementação --
    // por exemplo:
    int signalTurn(Direction direction, boolean signalOn) {
        // código para virar BMW para a esquerda acende luz indicadora
        // código para virar BMW para a direita desliga a luz indicadora
        // código para virar BMW para a esquerda liga a luz indicadora
        // código para virar BMW para a direita desliga a luz indicadora
    }

    // outros membros, se necessário -- por exemplo, classes de ajuda
    // invisíveis para os clientes da interface
}
```

No exemplo de carro-robô acima, são os fabricantes de automóveis quem implementam a interface. Uma implementação de **Chevrolet** poderá ser substancialmente diferente de uma **Toyota**, é claro, mas ambos os fabricantes adicionarão a mesma interface. Fabricantes de instrumentos, que são os clientes da interface, construirão sistemas que usam dados **GPS** em uma localização de carros, mapas digitais de estradas, e dados de tráfego para dirigir o carro. Fazendo dessa forma, os sistemas de instrumentos invocarão os métodos da interface; virar, mudar pista, frear, acelerar, e assim por diante.

Interfaces como APIs.

O exemplo de carro-robô mostra uma interface sendo usada como um padrão industrial **Application Programming Interface (API)**. **APIs** são também comuns em produtos de software comercial. Tipicamente, uma companhia vende um pacote de software que contém métodos complexos que outra companhia quer para usar em seu próprio produto de software. Um exemplo poderia ser um pacote de imagens digitais processando métodos que serão vendidos para companhias que fabricam programas gráficos para usuários-finais. A companhia de gráficos então invoca a imagem processando métodos usando a assinatura e tipo de retorno definidos na interface. Enquanto a **API** da companhia de processamento de imagens é feita pública (para seus clientes) esta implementação da **API** é mantida como um segredo bem guardado – de fato, ela precisa revisar a implementação de tempos em tempos e continuar a implementação da interface original na qual os clientes têm confiado.

Interfaces e Múltiplas Heranças.

Interfaces têm um outro papel muito importante na linguagem de programação Java. Interfaces não são parte da hierarquia de classes, apesar de elas trabalharem em combinação com classes. A linguagem de programação Java não permite múltiplas heranças (heranças serão discutidas mais tarde nesta lição), mas interfaces fornecem uma alternativa. Em Java, uma classe pode herdar de somente uma classe mas ela pode implementar mais de uma interface. Dessa forma, objetos podem ter múltiplos tipos: os tipos de suas próprias classes e os tipos de todas as interfaces que serão implementadas. Isto significa que se a variável é declarada para ser um tipo de uma interface, este valor pode fazer referência a qualquer objeto que é instanciado de qualquer classe que implementa uma interface. Isto será discutido mais tarde nesta lição, na seção intitulada “Usando uma Interface como um Tipo”.

Definindo uma Interface.

Uma declaração de interface consiste de modificadores, a palavra reservada **interface**, o nome da interface, uma lista separada por vírgulas das interfaces pai (se houver), e um corpo da interface. Por exemplo:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
    // declaração de constantes
    double E = 2.718282; // base dos logaritmos naturais
    void doSomething(int i double x); // assinaturas de métodos
    int doSomethingElse(String s);
}
```

O acesso **public** especificado indica que a interface pode ser usada por qualquer classe em qualquer pacote. Se você não especificar que a interface é pública, sua interface será acessível somente para classes definidas no mesmo pacote da interface.

Uma interface pode estender outras interfaces, exatamente como uma classe pode estender ou ter uma subclasse de qualquer classe. Contudo, considerando que uma classe pode estender somente uma outra classe, uma interface pode estender qualquer número de interfaces. A declaração da interface inclui uma lista separada por vírgulas de todas as interfaces que ela estende.

O Corpo da Interface.

O corpo da interface contém as declarações dos métodos para todos os métodos incluídos na interface. Uma declaração de métodos dentro de uma interface é seguida por um ponto-e-vírgula, mas não chaves, porque uma interface não fornece implementações para os métodos declarados dentro dela. Todos os métodos declarados em uma interface são implicitamente **public**, então o modificador **public** pode ser omitido.

Uma interface pode conter declarações de constantes em adição às declarações de métodos. Todos os valores constantes definidos em uma interface são implicitamente **public**, **static**, e **final**. Portanto, esses modificadores podem ser omitidos.

Implementando uma Interface.

Para declarar uma classe que implementa uma interface, você inclui uma cláusula `implements` na declaração da classe. Sua classe pode implementar mais de uma interface, mas a palavra reservada `implements` é seguida por uma lista separada por vírgulas de interfaces implementadas pela classe.

Por convenção, a cláusula `implements` segue a cláusula `extends`, se houver uma.

Um Exemplo de Interface, Comentado.

Considere uma interface que define como comparar o tamanho de objetos.

```
public interface Relatable {

    // this (objeto chamando isLargerThan) e
    // outro precisa ser instanciado da mesma classe
    // retorna 1, 0, -1, se este é maior
    // do que, igual a, ou menor que outro
    public int isLargerThan(Relatable other);
}
```

Se você quiser habilitar para comparar objetos similares, não esqueça o que eles são, a classe que instancia eles deve implementar `Relatable`.

Qualquer classe pode implementar `Relatable` se há algum caminho para comparar o tamanho relativo dos objetos instanciados da classe. Para *strings*, eles poderiam ser número de caracteres; para livros, eles poderiam ser número de páginas; para estudantes, eles poderiam ser altura; e assim por diante. Para objetos geométricos planos, a área poderia ser uma boa escolha (veja a classe `RectanglePlus` a seguir), enquanto volume poderia trabalhar com objetos geométricos tri-dimensionais.

Se você quer que a classe implemente `Relatable`, então você sabe que você pode comparar o tamanho dos objetos instanciados daquela classe.

```
//=====
```

Implementando a Interface `Relatable`.

Aqui está a classe `Rectangle` que foi apresentada na seção Criação de Objetos; reescrita para implementar `Relatable`.

```
public class RectanglePlus implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // quatro construtores
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // um método para mover o retângulo
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // um método para a computação da área do retângulo
    public int getArea() {
        return width * height;
    }

    // um método para implementar Relatable
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Porque `RectanglePlus` implementa `Relatable`, o tamanho de quaisquer dois objetos `RectanglePlus` podem ser comparados.

//=====

Usando uma Interface como um Tipo.

Quando você define uma nova interface, você está definindo um novo tipo de dado de referência. Você pode usar nomes de interfaces em qualquer outro lugar e você pode usar qualquer outro nome de tipo de dado. Se você define uma variável de referência cujo tipo é uma interface, qualquer objeto que você assinou para ele pode ter uma instância de uma classe que implementa a interface.

Como um exemplo, aqui está um método para encontrar a extensão de um objeto em um par de objetos, para qualquer objeto que está instanciado da classe que implementa `Relatable`:

```
public Object findLargest(Object object1, Object object2) {
    Relatable obj1 = (Relatable)object1;
    Relatable obj2 = (Relatable)object2;
    if ( (obj1).isLargerThan(obj2) > 0)
        return object1;
    else
        return object2;
}
```

Para lançar `object1` para um tipo `Relatable`, ele pode invocar o método `isLargerThan`.

Se você criar um ponto de implementação `Relatable` em uma vasta variedade de classes, os objetos instanciados de qualquer uma dessas classes podem ser comparados com o método `findLargest()` – estabelecido que ambos os objetos são da mesma classe. Similarmente, todos eles podem ser comparados com os seguintes métodos:

```
public Object findSmallest(Object object1, Object object2) {
    Relatable obj1 = (Relatable)object1;
    Relatable obj2 = (Relatable)object2;
    if ( (obj1).isLargerThan(obj2) < 0)
        return object1;
    else
        return object2;
}

public boolean isEqual(Object object1, Object object2) {
    Relatable obj1 = (Relatable)object1;
    Relatable obj2 = (Relatable)object2;
    if ( (obj1).isLargerThan(obj2) == 0)
        return true;
    else
        return false;
}
```

Esses métodos trabalham para quaisquer objetos “`relatable`”, não importa o que suas classes estão herdando. Quando eles implementam `Relatable`, eles podem ser ambos de seus próprios tipos de classes (ou superclasses) e um tipo `Relatable`. Isto dá a eles algumas das vantagens da múltipla herança, onde eles podem ter comportamento de ambas: a superclasse e a herança.

//=====

Reescrevendo Interfaces.

Considere uma interface que você desenvolveu chamada `DoIt`:

```
public interface DoIt {
    void doSomething(int i, double x);
    int doSomethingElse(String s);
}
```

Suponha que, mais tarde, você quer adicionar um terceiro método para `DoIt`, dessa forma, a interface vem a ser:

```
public interface DoIt {

    void doSomething(int i, double x);
    int doSomethingElse(String s);
    boolean didItWork(int i, double x, String s);
}
```

Se você fizer esta mudança, todas as classes que implementam a antiga interface `DoIt` serão quebradas porque elas não implementam mais a interface. Programadores confiando nesta interface protestarão com estrondo.

Tente avisar todos os usuários de sua interface e especifique isto completamente desde o início. Se isto for impossível, você pode precisar criar mais interfaces mais tarde. Por exemplo, você poderia criar a interface `DoIt` que estende `DoIt`:

```
public interface DoItPlus extends DoIt {

    boolean didItWork(int i, double x, String s);

}
```

Agora os usuários de seu código podem escolher continuar usando a interface antiga ou atualizar para a nova interface.

//=====

Um Exemplo de Herança.

Aqui está um código exemplo para uma possível implementação de uma classe `Bicycle`, que foi apresentada na lição de Classes e Objetos:

```
public class Bicycle {

    // a classe Bicycle tem três campos
    public int cadence;
    public int gear;
    public int speed;

    // a classe Bicycle tem um construtor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // a classe Bicycle tem quatro métodos
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

A declaração para a classe `MountainBike` que é uma subclasse de `Bicycle` pode parecer com isto:

```
public class MountainBike extends Bicycle {

    // a subclasse MountainBike adiciona um campo
    public int seatHeight;

    // a subclasse MountainBike tem um construtor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // a subclasse MountainBike adiciona um método
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

`MountainBike` herda todos os campos e métodos de `Bicycle` e adiciona o campo `seatHeight` e o método para apontá-lo. Exceto para o construtor, isto é como se você tivesse escrito uma nova classe `MountainBike` a partir do início, com seus campos e cinco métodos. No entanto, você não tem de fazer todo o trabalho. Isto pode ser especialmente útil se os métodos na classe `Bicycle` são complexos e levam um tempo substancial para debugar.

O Que Você Pode Fazer em uma Subclasse.

Uma subclasse herda todos os membros `public` e `private` de sua classe pai, não importa em qual pacote a subclasse esteja. Se a subclasse está no mesmo pacote que a classe pai, ela também herda os membros `package-private` da classe pai. Você pode usar os membros herdados assim como, substituí-los, escondê-los, ou supri-los com novos membros:

- Os campos herdados podem ser usados diretamente, assim como quaisquer outros campos.
- Você pode declarar um campo na subclasse com o mesmo nome que um na superclasse, dessa maneira escondendo ele (não recomendado).
- Você pode declarar novos campos na subclasse que não estão na superclasse.
- Os métodos herdados podem ser usados diretamente com eles são.
- Você pode escrever uma nova instância de método na subclasse que tem a mesma assinatura que uma na superclasse, dessa forma passando por cima dela.
- Você pode escrever um novo método `static` na subclasse que não está na superclasse.
- Você pode escrever um construtor na subclasse que invoca o construtor da superclasse, ou implicitamente ou usando a palavra reservada `super`.

A seção seguinte nesta lição explicará estes tópicos.

Membros Privados em uma Superclasse.

Uma subclasse não pode herdar os membros `private` de sua classe pai. No entanto, se a superclasse tiver métodos `public` ou `protected` para acessar estes campos `private`, eles podem também ser usados pela subclasse.

Uma classe aninhada tem acesso para todos os membros `private` da classe que a envolve – inclusive campos e métodos. Conseqüentemente, uma classe aninhada `public` ou `protected` herdada de uma subclasse tem acesso indireto para todos os membros `private` da superclasse.

Lançando Objetos.

Nós vimos que um objeto é do tipo de dado da classe da qual ele foi instanciado. Por exemplo, se você escrever:

```
public MountainBike myBike = new MountainBike();
```

então `myBike` é do tipo `MountainBike`.

`MountainBike` é descendente de `Bike` e `Object`. Conseqüentemente, uma `MountainBike` é uma `Bicycle` e é também um `Object`, e ela pode ser usada em qualquer lugar onde `Bicycle` ou `Object` forem chamados.

O contrário não é necessariamente verdade: uma `Bicycle` pode ser uma `MountainBike`, mas não necessariamente. Similarmente, um `Object` pode ser uma `Bicycle` ou uma `MountainBike`, mas não necessariamente.

Lançamento (**Casting**) mostra o uso de um objeto de um tipo no lugar de outro tipo, entre os objetos permitidos pela herança e implementação. Por exemplo, se você escrever:

```
Object obj = new MountainBike();
```

então `obj` é um `Object` e uma `MountainBike` (até o momento em que `obj` é atribuído a qualquer objeto que não uma `MountainBike`). Isto é chamado de lançamento implícito (**implicit casting**).

Se, por outro lado, você escrever:

```
MountainBike myBike = obj;
```

você pode conseguir um erro em tempo de compilação, porque `obj` não é conhecido para o compilador para ser um `MountainBike`. Porém, você pode dizer ao compilador que você promete atribuir uma `MountainBike` para `obj` por lançamento explícito (***explicit casting***):

```
MountainBike myBike = (MountainBike)obj;
```

Este lançamento insere uma checagem em tempo de execução na qual `obj` é atribuído a `MountainBike`, dessa forma o compilador pode assumir sem perigo que `obj` é uma `MountainBike`. Se `obj` não é uma `MountainBike` em tempo de execução, uma exceção será lançada.

Nota: Você pode fazer um teste lógico de um tipo em particular de objeto usando o operador `instanceof`. Isto pode salvar você de um erro em tempo de execução devido a um lançamento inapropriado. Por exemplo:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Aqui um operador `instanceof` verifica de que maneira um `obj` refere-se a uma `MountainBike` de maneira que nós podemos fazer o lançamento com conhecimento (***cast***) que não haverá lançamento de exceção em tempo de execução.

Passando Por Cima e Escondendo Métodos.

Instância de Métodos.

Uma instância de métodos em uma subclasse com a mesma assinatura (nome, membro adicional e o tipo de seus parâmetros) e retorna o tipo como uma instância do método na superclasse passando por cima (***overriding***) do método da superclasse.

A capacidade de uma subclasse para passar por cima de um método permite a uma classe herdar de uma superclasse aquele comportamento que é “fechado o suficiente” e tem então a modificação do comportamento como necessidade. O método de passar por cima pode também retornar um subtipo de um tipo retornado pelo método sobrescrito. Isto é chamado um ***retorno de tipo covariante***.

Quando passando por cima de um método, você pode querer usar a anotação `@Override` que instrui o compilador que você pretende passar por cima de um método na superclasse. Se, por alguma razão, o compilador detectar que o método não existe na superclasse, ele gerará um erro.

Métodos de Classe.

Se uma subclasse define um método de classe com a mesma assinatura que um método na superclasse, o método na subclasse esconde ele na superclasse.

A distinção entre esconder e passar por cima tem implicações importantes. A versão da passagem por cima do método que é invocado é o mesmo na subclasse. A versão de esconder o método que é invocado depende se ele é invocado da superclasse ou da subclasse. Dê uma olhada no exemplo que contém duas classes. A primeira é `Animal`, que contém um instância do método e um método da classe:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
}
```

A segunda classe, uma subclasse de `Animal`, é chamada `Cat`:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

A classe `Cat` passa por cima do método da instância em `Animal` e esconde o método da classe em `Animal`. O método `main` na classe cria uma instância de `Cat` e chama `testClassMethod()` na classe e `testInstanceMethod()` na instância.

A saída do programa é a seguinte:

```
The class method in Animal.
The instance method in Cat.
```

Como prometido, a versão de esconder o método que é invocado é o mesmo na superclasse, e a versão de passar por cima do método que é invocado é a mesma na subclasse.

Modificadores.

O acesso especial para um método passado por cima pode permitir mais, mas não menos, acesso que o método de passar por cima. Por exemplo, um método da instância `protected` na superclasse pode ser feito `public`, mas não `private`, na subclasse.

Sumário: A seguinte tabela resume o que acontece quando você define um método com a mesma assinatura que um método na superclasse:

Definindo um Método com a Mesma Assinatura que o Método da Superclasse		
	Método da Instância da Superclasse	Método Estático da Superclasse
Método da Instância na Subclasse	Passagem por cima	Gera um erro em tempo de compilação
Método Estático da Subclasse	Gera um erro em tempo de compilação	Esconde

Nota: Na subclasse, você pode sobrecarregar os métodos herdados da superclasse. Dessa maneira nenhum método herdado esconde nem sobrecarrega os métodos da superclasse – eles são novos métodos, exclusivos para a subclasse.

Escondendo Campos.

Dentro da classe, um campo que tem o mesmo nome de um campo na superclasse esconde os campos da superclasse, mesmo se seus tipos são diferentes. Dentro da subclasse, o campo na superclasse não pode ser referenciado por seu nome simples. Em vez disso, o campo pode ser acessado dentro de `super`, que é coberto na próxima seção. Falando genericamente, nós não recomendamos esconder campos pois isso faz o código difícil de ler.

Usando a Palavra Reservada `super`.

Acessando Membros da Superclasse.

Se seu método passa por cima de um dos métodos da superclasse, você pode invocar o método de passar por cima direto usando a palavra reservada `super`. Você também pode usar `super` para referir-se a um campo escondido (apesar de esconder campos ser desencorajado) Considere esta classe, `Superclass`:

```
public class Superclass {
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Aqui está uma subclasse, chamada `Subclass`, que passa por cima de `printMethod()`:

```
public class Subclass extends Superclass {
    public void printMethod() { // passa por cima de printMethod na
                               // Superclasse
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Dentro da `Subclass`, o nome simples `printMethod()` refere-se para o mesmo declarado na `Subclass`, que passa por cima do mesmo na `Superclass`. Desta forma, para referir-se para `printMethod()` herdado da `Superclass`, `Subclass` precisa usar um nome qualificador, usando `super` como mostrado. Compilando e executando `Subclass` mostra o seguinte:

```
Printed in Superclass.
Printed in Subclass.
```

Construtores da Subclasse.

O seguinte exemplo ilustra como usar a palavra reservada `super` para invocar um construtor da superclasse. Relembre que no exemplo `Bicycle`, `MountainBike` é uma subclasse de `Bicycle`. Aqui está o construtor da subclasse `MountainBike` que chama o construtor da superclasse e então adiciona o código de inicialização dela mesma:

```
public MountainBike(int startHeight, int startCadence,
                    int startSpeed, int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

A invocação de um construtor da superclasse pode ser a primeira linha no construtor da subclasse. A sintaxe para chamar um construtor da superclasse é:

```
super();
--or--
super(lista de parâmetros);
```

Com `super()`, o construtor sem argumentos da superclasse é chamado. Com `super (lista de parâmetros)`, o construtor da superclasse com os parâmetros correspondentes na lista é chamado.

Nota: Se um construtor não invoca explicitamente um construtor da superclasse, o compilador Java automaticamente insere uma chamada para o construtor sem argumentos da superclasse. Se a superclasse não tem um construtor sem argumentos, você receberá um erro em tempo de compilação. `Object` atua como um construtor, dessa forma se `Object` é o único na superclasse, não há problema.

Se um construtor na subclasse invoca um construtor desta superclasse, seja implícita ou explicitamente, você pode pensar que será chamada uma série completa de construtores, todos retornando para o construtor de `Object`. De fato, este é o caso. Isto é chamado **constructor chaining** (construtor encadeado), e você precisa tomar cuidado quando há uma longa linha de classes descendentes.

Object como uma Superclasse.

A classe `Object`, no pacote `java.lang`, está colocado no topo da árvore de hierarquia de classes. Cada classe é uma descendente, direta ou indiretamente, da classe `Object`. Você não precisa usar nenhum desses métodos, mas, se você escolher usar, você precisa passar por cima dele com o código que é especificado para sua classe. Os métodos herdados de `Object` que são discutidos nesta seção são:

- `protected Object clone() throws CloneNotSupportedException`
Cria e retorna uma cópia do objeto.
- `public boolean equals(Object obj)`
Indica se algum outro objeto é "igual a" este.
- `protected void finalize() throws Throwable`
Chamada para o garbage collector em um objeto quando o garbage collector determina que não há mais referências para o objeto.
- `public final Class getClass()`
Retorna a classe em tempo de execução de um objeto.
- `public int hashCode()`
Retorna um valor hash code (código revisado) para o objeto.
- `public String toString()`
Retorna uma representação de string de um objeto.

Os métodos `notify`, `notifyAll`, e `wait` de `Object` todos acionam uma parte em atividades de sincronização independentemente da fila corrente em um programa, o que será discutido mais tarde na lição e não será coberto agora. Há cinco desses métodos:

- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait()`
- `public final void wait(long timeout)`
- `public final void wait(long timeout, int nanos)`

Nota: Há alguns aspectos sutis para alguns desses métodos, especialmente o método `clone`.

O Método `clone()`.

Se a classe, ou sua superclasse, implementa a interface `Cloneable`, você pode usar o método `clone()` para criar uma cópia de um objeto existente. Para criar um clone, você escreve:

```
aCloneableObject.clone();
```

A implementação de `Object` deste método checa para ver se o objeto em que `clone()` foi invocado implementa a interface `Cloneable`. Se o objeto não faz isso, o método lança uma exceção `CloneNotSupportedException`. Exceções manuais serão discutidas em uma lição mais tarde. Por este momento, você precisa saber que `clone()` precisa ser declarado como:

```
protected Object clone() throws CloneNotSupportedException
-- ou --
public Object clone() throws CloneNotSupportedException
```

se você está escrevendo um método `clone()` para passar por cima do mesmo em `Object`.

Se o objeto no qual `clone()` foi invocado não implementa a interface `Cloneable`, a implementação de `Object` do método `clone()` cria um objeto da mesma classe que o objeto original e inicializa as novas variáveis membros do objeto para terem os mesmos valores que as variáveis membros do objeto original correspondente.

Um caminho simples para fazer sua classe clonável é adicionar `implements Cloneable` para sua declaração de classe, então seus objetos podem invocar o método `clone()`.

Para algumas classes, o comportamento padrão do método `Object's clone()` trabalha muito bem. Se, no entanto, um objeto conter uma referência para um objeto externo, chamado `ObjExternal`, você precisaria passar por cima de `clone()` para conseguir o comportamento correto. Por outro lado, uma mudança em `ObjExternal` pode fazer com que um objeto seja visível neste clone também. Isto significa que o objeto original e este clone não são independentes – para desassociá-los, você precisa passar por cima de `clone()`, para que ele clone o objeto e `ObjExternal`. Então o objeto original referencia `ObjExternal` e o clone referencia um clone de `ObjExternal`, dessa forma o objeto e seu clone serão verdadeiramente independentes.

O Método `equals()`.

O método `equals()` compara dois objetos pela igualdade e retorna `true` se ele for igual. O método `equals()` fornecido na classe `Object` usa o operador de identidade (`==`) para determinar se dois objetos são iguais. Para tipos de dados primitivos, ele retorna o resultado correto. Para objetos, no entanto, ele não o faz. O método `equals()` fornecido por `Objects` testa se as referências ao objeto são iguais – o que é, se os objetos comparados forem exatamente do mesmo tipo.

Para testar se dois objetos são iguais no sentido da equivalência (contendo a mesma informação), você precisa passar por cima o método `equals()`. Aqui está um exemplo de uma classe `Book` que usa `equals()`:

```
public class Book {
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book) obj.getISBN());
        else
            return false;
    }
}
```

Considere este código que testa a igualdade de duas instâncias da classe `Book`:

```
Book firstBook = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

Este programa mostra `objects are equal` comparando através da referência de dois objetos distintos `firstBook` e `secondBook`. Eles são considerados iguais porque os objetos comparados contêm o mesmo número ISBN.

Você sempre pode passar por cima o método `equals()` se o operador de identificação não é apropriado para sua classe.

Nota: Se você passar por cima `equals()`, você precisa pode passar por cima `hashCode()` da mesma forma.

O Método `finalize()`.

A classe `Object` fornece um método de retorno, `finalize()`, que pode ser invocado em um objeto quando ele torna-se lixo. A implementação de `Object`, `finalize()` não faz nada – você pode passar por cima `finalize()` para fazer limpeza, para liberação de recursos.

O método `finalize()` pode ser chamado automaticamente pelo sistema, mas quando ele é chamado, ou então se ele é chamado, é incerto. No entanto, você não pode confiar neste método para fazer isto por você. Por exemplo, se você não fechar as descrições de arquivos em seu código depois da execução *I/O (input/output)* e você omitir `finalize()` para fechá-lo para fazer isto por você, você pode perder os descritores do arquivo.

O Método `getClass()`.

Você não pode passar por cima `getClass`.

O método `getClass()` retorna um objeto `Class`, que tem métodos que você pode usar para conseguir informações à respeito da classe, da mesma maneira que o nome (`getSimpleName()`), a superclasse (`getSuperclass()`), e a interface que esta implementa (`getInterfaces()`). Por exemplo, o seguinte método pega e mostra o nome da classe de um objeto:

```
void printClassName(Object obj) {
    System.out.println("The object's class is "
        + obj.getClass().getSimpleName());
}
```

A classe `Class`, no pacote `java.lang`, tem um grande número de métodos (mais de 50). Por exemplo, você pode testar para ver se a classe é uma anotação (`isAnnotation()`), uma interface (`isInterface()`), ou uma enumeração (`isEnum()`). Você pode ver o que os campos dos objetos são (`getFields()`) ou o que estes métodos são (`getMethods()`), e assim por diante.

O Método `hashCode()`.

O valor retornado por `hashCode()` é o *hash code* código modificado do objeto, que é o endereço de memória do objeto na memória em hexadecimal.

Por definição, se dois objetos são iguais, seus *hash code* precisam também ser iguais. Se você passa por cima o método `equals()`, você altera o modo como dois objetos são igualados e a implementação de `Object` de `hashCode()` não é mais válida. Portanto, se você passar por cima o método `equals`, você precisa também passar por cima o método `hashCode()`.

O Método `toString()`.

Você deve considerar sempre a passagem do método `toString()` em suas classes.

O método `toString()` de `Object` retorna uma representação `String` do objeto, que é muito útil para debug. A representação `String` para um objeto depende unicamente do objeto, que é o motivo de você precisar passar `toString()` em suas classes.

Você pode usar `toString()` juntamente com `System.out.println()` para mostrar uma representação de um objeto, como uma instância de `Book`:

```
System.out.println(firstBook.toString());
```

que, pelo método da propriedade `toString()` passada, mostra alguma coisa útil, como isto:

```
ISBN: 0201914670; The JFC Swing Tutorial; A Guide to Constructing GUIs,
2nd Editon
```

Escrevendo Métodos e Classes Finais.

Você pode declarar alguns ou todos os métodos da classe como **final**. Você usa a palavra reservada **final** em uma declaração do método para indicar que o método não pode ser passado por cima pela subclasse. A classe **Object** faz isso – um número desses métodos é **final**.

Você pode escolher fazer um método final se ele tem uma implementação que não pode ser mudada e se é crítico para a consistência do estado do objeto. Por exemplo, você pode querer fazer o método **getFirstPlayer** nesta classe final **ChessAlgorithm**:

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Métodos chamados dos construtores podem geralmente ser declarados finais. Se um construtor chama um método não-final, uma subclasse pode redefinir o método com resultados inesperados ou indesejáveis.

Note que você pode também declarar uma classe inteira final – isto evita a classe de ser subclasse. Isto é particularmente útil, por exemplo, quando estiver criando uma classe imutável como a classe **String**.

Classes e Métodos Abstratos.

Uma classe abstrata é uma classe que é declarada **abstract** – ela pode ou não incluir métodos abstratos. Classes abstratas não podem ser instanciadas, mas elas podem ter uma subclasse.

Um método abstrato é um método que é declarado sem uma implementação (sem chaves, e seguido por um ponto-e-vírgula), como este:

```
abstract void moveTo(double destX, double deltaY);
```

Se a classe incluir métodos abstratos, a classe precisa ela mesma ser declarada **abstract**, como em:

```
public abstract class GraphicObject {
    // declare campos
    // declare métodos não abstratos
    abstract void draw();
}
```

Quando uma classe abstrata tem uma subclasse, a subclasse usualmente fornece implementações para todos os métodos abstratos na classe pai. Entretanto, se ela não o faz, a subclasse precisa também ser declarada **abstract**.

Nota: Todos os métodos em uma interface são implicitamente **abstract**, então o modificador **abstract** não é usado com métodos da interface (ele poderia ser – somente isto não é necessário).

Classes Abstratas versus Interfaces.

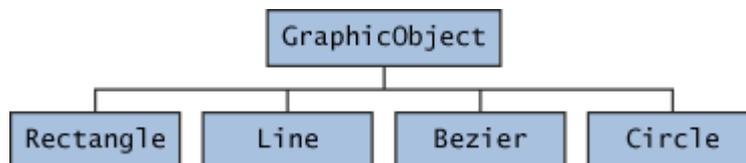
Diferente de interfaces, classes abstratas podem conter campos que não são **static** e **final**, e eles podem conter métodos implementados. Como classes **abstract** são similares a **interfaces**, exceto que elas fornecem uma implementação parcial, deixam para a subclasse a complementação da implementação. Se uma classe **abstract** contém somente declarações de métodos abstratos, ela pode ser declarada como uma **interface** em seu lugar.

Múltiplas interfaces podem ser implementadas pelas classes em qualquer lugar na hierarquia da classe, sendo ou não relatadas para uma outra em qualquer lugar. Pense em **Comparable** e **Cloneable**, por exemplo.

Por comparação, classes **abstract** são comumente subclasses para compartilhar peças da implementação. A classe simples **abstract** é uma subclasse para classes similares que têm uma parte em comum (as partes da implementação da classe abstrata), mas também têm algumas diferenças (os métodos abstratos).

Um Exemplo de Classe Abstrata.

Em uma aplicação de desenho orientado a objetos, você pode desenhar círculos, retângulos, linhas, curvas, e muitos outros objetos gráficos. Todos esses objetos têm certamente estados (por exemplo: **position**, **orientation**, **line color**, **fill color**) e comportamentos (por exemplo: **moveTo**, **rotate**, **resize**, **draw**) em comum. Alguns desses estados e comportamentos são os mesmos para todos os objetos gráficos – por exemplo: **position**, **fill color**, and **moveTo**. Outros requerem implementações diferentes – por exemplo, **resize** ou **draw**. Todos os **GraphicObjects** precisam conhecer como desenhar ou redimensionar eles mesmos; eles só diferem em como eles fazem isso. Esta é uma situação perfeita para uma superclasse **abstract**. Você pode tirar vantagem das similaridades e declarar todos os objetos gráficos para herdarem do mesmo objeto pai – por exemplo, **GraphicObject**, como mostrado na seguinte figura:



Primeiro, você declara uma classe abstrata, **GraphicObject**, para fornecer variáveis e métodos dos membros que são totalmente compartilhados por todas as subclasses, como a posição atual e o método **moveTo**. **GraphicsObject** também declara métodos abstratos para métodos, como um **draw** ou **resize**, que são implementados por todas as subclasses mas precisam ser implementados de diferentes maneiras. A classe **GraphicObject** podem parecer como isto:

```

abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
  
```

Cada subclasse não abstrata de `GraphicObject`, como um `Circle` ou `Rectangle`, pode fornecer implementações para os métodos `draw` e `resize`:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

Quando uma Classe Abstrata Implementa uma Interface.

Na seção *Interfaces*, você deve ter notado que a classe que implementa uma interface pode implementar todos os métodos da interface. Isto é possível, contudo, para definir uma classe que não implementa todos os métodos da interface, certifique-se que a classe é declarada para ser `abstract`. Por exemplo:

```
abstract class X implements Y {
    // implementa tudo exceto um método de Y
}
class XX extends X {
    // implementa o restante do método em Y
}
```

Neste caso, a classe `X` pode ser `abstract` porque ela não implementa completamente `Y`, mas a classe `XX`, de fato, implementa `Y`.

Membros da Classe.

Uma classe abstrata pode ter campos estáticos e métodos estáticos. Você pode usar membros estáticos com um referência à classe – por exemplo, `AbstractClass.staticMethod()` - como você faria com qualquer outra classe.

A Classe Números.

Quando trabalhando com números, muitas vezes você usa tipos primitivos em seus códigos. Por exemplo:

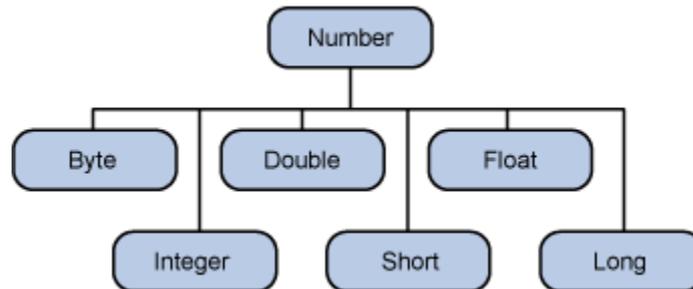
```
int i = 500;
float gpa = 3.65;
byte mask = 0xff;
```

Há, no entanto, razões para usar objetos em lugar de primitivos, e a plataforma Java fornece classes empacotadas para cada tipo de dados primitivos. Essas classes “empacotam” o primitivo em um objeto. Muitas vezes, a embalagem é concluída pelo compilador – se você usa um primitivo ao invés de um objeto esperado, o compilador encaixota o primitivo nessa classe empacotada para você. Similarmente, se você usa um objeto **number** quando um primitivo é esperado, o compilador desempacota o objeto para você.:

```
Integer x, y;
x = 12;
y = 15;
System.out.println(x+y);
```

Quando **x** e **y** são designados como valores inteiros, o compilador encaixota os inteiros porque **x** e **y** são objetos inteiros. Na declaração `println()`, **x** e **y** são desempacotados para que eles possam ser adicionados como inteiros.

Todas as classes embaladas numéricas são subclasses da classe abstrata **Number**:



Nota: Há quatro outras subclasses de **Number** que não são discutidas aqui. **BigDecimal** e **BigInteger** são usadas para cálculos de alta precisão. **AtomicInteger** e **AtomicLong** são usados para aplicações **multi-thread**.

Há três razões para que você use um objeto **Number** ao invés de um primitivo:

- Como um argumento de um método que espera um objeto (muitas vezes usado quando manipulando coleções de números).
- Para usar constantes definidas pela classe, como em **MIN_VALUE** e **MAX_VALUE**, que fornece o maior e o menor salto de um tipo de dado.
- Para usar métodos de classe para conversão de valores para e dos tipos primitivos, para conversão para e de **strings**, e para conversão entre sistemas numéricos (decimal, octal, hexadecimal, binário).

A seguinte tabela lista os métodos da instância que todas as subclasses da classe `Number` implementam.

Métodos Implementados por todas as Subclasses de <code>Number</code>	
Método	Descrição
<pre>byte byteValue() short shortValue() int intValue() long longValue() float floatValue() double doubleValue()</pre>	Converte o valor deste objeto <code>Number</code> para o tipo de dado primitivo retornado.
<pre>int compareTo(Byte anotherByte) int compareTo(Double anotherDouble) int compareTo(Float anotherFloat) int compareTo(Integer anotherInteger) int compareTo(Long anotherLong) int compareTo(Short anotherShort)</pre>	Compara este objeto <code>Number</code> para o argumento.
<pre>boolean equals(Object obj)</pre>	Determina se este objeto <code>Number</code> é igual ao argumento. Os métodos retornam <code>true</code> se o argumento não é <code>null</code> e é um objeto do mesmo tipo e com o mesmo valor numérico. Há alguns requerimentos extras para objetos <code>Double</code> e <code>Float</code> que são descritos na documentação API Java.

Cada classe `Number` contém outros métodos que são úteis para conversão de números para e de *strings* e para conversão entre sistemas de números. A seguinte tabela lista estes métodos na classe `Integer`. Métodos de outras subclasses `Number` são similares.

Métodos de Conversão, Classe <code>Integer</code>	
Método	Descrição
<pre>static Integer decode(String s)</pre>	Decodifica uma <i>string</i> em um inteiro. Pode aceitar representações <i>string</i> de números decimais, octais, ou hexadecimais como entrada.
<pre>static int parseInt(String s)</pre>	Retorna um inteiro (somente decimal)
<pre>static int parseInt(String s, int radix)</pre>	Retorna um inteiro, dando uma representação <i>string</i> de números decimais, binários, octais, ou hexadecimais (<code>radix</code> igual a 10, 2, 8, ou 16, respectivamente) como entrada.
<pre>String toString()</pre>	Retorna um objeto <code>String</code> representando o valor deste <code>Integer</code> .
<pre>static String toString(int i)</pre>	Retorna um objeto <code>String</code> representando o inteiro especificado.
<pre>Static Integer valueOf(int i)</pre>	Retorna um objeto <code>Integer</code> mantendo o valor do primitivo especificado.
<pre>static Integer valueOf(String s)</pre>	Retorna um objeto <code>Integer</code> mantendo o valor da representação da <i>string</i> especificada.
<pre>static Integer valueOf(String s, int radix)</pre>	Retorna um objeto <code>Integer</code> mantendo o valor inteiro da representação da <i>string</i> especificada, analisada gramaticalmente com o valor de <code>radix</code> . Por exemplo, se <code>s = "333"</code> e <code>radix = 8</code> , o método retorna o inteiro de base 10 equivalente para o número octal 333.

Formatando Saídas Numéricas.

Inicialmente você viu o uso dos métodos `print()` e `println()` para mostrar *strings* na saída padrão (`System.out`). Desde que todos os números podem ser convertidos em *strings*, você pode usar esses métodos para mostrar uma combinação de *strings* e números. A linguagem de programação Java tem outros métodos, no entanto, que permitem a você exercer muito mais controle sobre suas saídas quando números são incluídos.

Os Métodos `printf()` e `println()`.

O pacote `java.io` inclui uma classe `PrintStream` que tem duas formatações de métodos que você pode usar para substituir `print()` e `println()`. Esses métodos, `format()` e `printf()`, são equivalentes aos anteriores. O familiar `System.out` que você está usando existe para ser o objeto `PrintStream`, então você pode invocar os métodos `PrintStream` em `System.out`. Dessa forma, você pode usar `format()` ou `printf()` em qualquer lugar em seu código onde você tenha previamente usado `print()` ou `println()`. Por exemplo,

```
System.out.format(.....);
```

A sintaxe para esses dois métodos `java.io.PrintStream` é a mesma:

```
public PrintStream format(String format, Object... args)
```

onde `format` é uma *string* que especifica a formatação a ser usada e `args` é uma lista de variáveis para serem mostradas usando aquela formatação. Um exemplo simples poderia ser:

```
System.out.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);
```

O primeiro parâmetro, `format`, é o formato da *string* especificando como os objetos no segundo parâmetro, `args`, estão sendo formatados. O formato da *string* contém um texto explicando os especificadores de formato, que são caracteres especiais que formatam argumentos de `Object... args`. (A notação `Object... args` é chamada *varargs*, que menciona que o número de argumentos pode variar).

Especificadores de formato começam com o sinal de porcentagem (`%`) e terminam com um conversor. O conversor é um caracter indicando o tipo do argumento para ser formatado. Entre o sinal de porcentagem (`%`) e o conversor você pode ter marcadores e especificadores opcionais. Há muitos conversores, marcadores, e especificadores, que são documentados em `java.util.Formatter`.

Aqui está um exemplo básico:

```
int i = 461012;
System.out.format("The value of i is: %d%n", i);
```

O especificador `%d` especifica que a variável simples é um inteiro decimal. O `%n` é um caracter de nova linha independente de plataforma. A saída é:

```
The value of i is: 461012
```

Os métodos `printf()` e `format()` são sobrecarregados. Cada um tem uma versão com a seguinte sintaxe:

```
public PrintStream format(Locale l, String format, Object... args)
```

Para mostrar números em sistema francês (onde a vírgula é usada em lugar de um ponto decimal na representação inglesa de números de ponto flutuante), por exemplo, você pode usar:

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

Um Exemplo.

A seguinte tabela lista alguns dos conversores e marcadores que são usados em um simples programa, `TestFormat.java`, que é mostrado após a tabela:

Conversores e Marcadores (flags) em <code>TestFormat.java</code>		
Conversor	Marcador (flag)	Explicação
<code>d</code>		Um inteiro decimal.
<code>f</code>		Um float (flutuante)
<code>n</code>		Um caracter de nova linha apropriado para a plataforma rodando a aplicação. Você pode usar sempre <code>%n</code> , ao invés de <code>\n</code> .
<code>tB</code>		Uma conversão date & time – especifica localmente o nome completo do mês.
<code>td, te</code>		Uma conversão date & time – 2 dígitos dia do mês. <code>td</code> foi conduzido para zero, se necessário, <code>te</code> não.
<code>ty, tY</code>		Uma conversão date & time – <code>ty</code> = 2 dígitos do ano, <code>tY</code> = 4 dígitos do ano.
<code>t1</code>		Uma conversão date & time – formato de horas em 12.
<code>tM</code>		Uma conversão date & time – minutos em 2 dígitos, carregando os zeros se necessário
<code>tp</code>		Uma conversão date & time – específica local am/pm (minúsculo)
<code>tm</code>		Uma conversão date & time – meses em 2 dígitos, carregando zeros, se necessário
<code>tD</code>		Uma conversão date & time – data como <code>%tm%td%ty</code>
	<code>08</code>	Oito caracteres de largura, carregando zeros se necessário
	<code>+</code>	Inclui sinais, se positivo ou negativo
	<code>,</code>	Inclui caracteres especificados agrupados localmente
	<code>-</code>	Justificado à esquerda
	<code>.3</code>	Três posições depois do ponto decimal
	<code>10.3</code>	Dez caracteres na largura, justificado à direita, com três posições depois do ponto decimal

O seguinte programa mostra algumas formatações que você pode fazer com `format()`. O pacote `java.util` precisa ser importado pois ele contém as classes `Locale` e `Calendar`:

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

    public static void main(String[] args) {
        long n = 461012;
        System.out.format("%d%n", n);
        System.out.format("%08d%n", n);
        System.out.format("%+8d%n", n);
        System.out.format("%,8d%n", n);
        System.out.format("%+,8d%n%n", n);

        double pi = Math.PI;
        System.out.format("%f%n", pi);
        System.out.format("%.3f%n", pi);
        System.out.format("%10.3f%n", pi);
        System.out.format("%-10.3f%n", pi);
        System.out.format(Locale.FRANCE, "%-10.4f%n%n", pi);

        Calendar c = Calendar.getInstance();
        System.out.format("%tB %te, %tY%n", c, c, c);
        System.out.format("%tl:%tM %tp%n", c, c, c);
        System.out.format("%tD%n", c);
    }
}
```

A saída é:

```
461012
00461012
+461012
461,012
+461,012
```

```
3.141593
3.142
    3.142
3.142
3,1416
```

```
May 29, 2006
2:34 am
05/29/06
```

Nota: A discussão nesta seção cobre somente os métodos básicos de `format()` e `printf()`. Detalhes adicionais podem ser encontrados na seção **Basic I/O** intitulada "Formatando".

A Classe `DecimalFormat`.

Você pode usar a classe `java.text.DecimalFormat` para controlar a saída de zeros direcionados e rastreados, prefixos e sufixos, separadores agrupados (milhares), e o separador decimal. `DecimalFormat` oferece um grande rol de flexibilidade na formatação de números, mas ela pode fazer seus códigos complexos.

O seguinte exemplo cria um objeto `DecimalFormat`, `myFormatter`, passando uma *string* padrão para o construtor `DecimalFormat`. O método `format()`, que `DecimalFormat` herda de `NumberFormat`, é então invocado por `myFormatter` – ele aceita um valor `double` como um argumento e retorna o número formatado em uma *string*:

Aqui está um exemplo de programa que ilustra o uso de `DecimalFormat`:

```
import java.text.*;

public class DecimalFormatDemo {

    static public void customFormat(String pattern, double value) {
        DecimalFormat myFormatter = new DecimalFormat(pattern);
        String output = myFormatter.format(value);
        System.out.println(value + " " + pattern + " " + output);
    }

    static public void main(String[] args) {

        customFormat("###,###.###", 123456.789);
        customFormat("###.##", 123456.789);
        customFormat("000000.000", 123.78);
        customFormat("$###,###.###", 12345.67);
    }
}
```

A saída é:

```
123456.789 ###,###.### 123,456.789
123456.789 ###.## 123456.79
123.78 000000.000 000123.780
12345.67 $###,###.### $12,345.67
```

A seguinte tabela explica cada linha da saída:

Saída de <code>DecimalFormat.java</code>			
Valor	Padrão	Saída	Explicação
123456.789	###,###.###	123,456.789	O sinal “cerquilha” (#) denota um dígito, a vírgula é uma lugar mantido para a separação do grupo, e o ponto é um lugar mantido para o separador decimal
123456.789	###.##	123456.79	O <code>value</code> tinha três dígitos à direita do ponto decimal, mas o padrão tem somente dois. O método <code>format</code> trata isto arredondando acima.
123.78	000000.000	000123.780	O padrão especifica rastro e carregamento de zeros, porque o caracter 0 é usado ao invés do sinal “cerquilha” (#)
12345.67	\$###,###.###	\$12,345.67	O primeiro caracter na configuração é o sinal de dólar (\$). Note que ele imediatamente precede o dígito mais à esquerda na saída formatada.

Além da Aritmética Básica.

A linguagem de programação Java suporta aritmética básica com os operadores: `+`, `-`, `*`, `/` e `%`. A classe `Math` no pacote `java.lang` fornece métodos e constantes para fazer cálculos matemáticos mais avançados.

Os métodos na classe `Math` são todos estáticos, então você precisa chamá-los diretamente da classe, como em:

```
Math.cos(angle);
```

Nota: Usando a característica da linguagem `static import`, você não precisa escrever `Math` na frente de cada função matemática:

```
import static java.lang.Math.*;
```

Isto permite a você invocar os métodos da classe `Math` pelos seus nomes simples. Por exemplo:

```
cos(angle);
```

Constantes e Métodos Básicos.

A classe `Math` inclui duas constantes:

- `Math.E`, que é a base dos logaritmos naturais, e
- `Math.PI`, que é o raio da circunferência de um círculo para seu diâmetro.

A classe `Math` também inclui mais de 40 métodos estáticos. A seguinte tabela lista alguns dos métodos básicos:

Métodos Básicos <code>Math</code>	
Método	Descrição
<pre>double abs(double d) float abs(float f) int abs(int i) long abs(long lng)</pre>	Retorna o valor absoluto do argumento
<pre>double ceil(double d)</pre>	Retorna o menor inteiro que é maior ou igual ao argumento. Retornado como <i>double</i>
<pre>double floor(double d)</pre>	Retorna o maior inteiro que é menor ou igual ao argumento. Retornado como <i>double</i> .
<pre>double rint(double d)</pre>	Retorna o inteiro que é fechado no valor do argumento. Retornado como <i>double</i> .
<pre>long round(double d) int round(float f)</pre>	Retorna o fechado <code>long</code> ou <code>int</code> , como indicado pelo tipo de retorno do método, para o argumento
<pre>double min(double arg1, double arg2) float min(float arg1, float arg2) int min(int arg1, int arg2) long min(long arg1, long arg2)</pre>	Retorna o menor dos dois argumentos
<pre>double max(double arg1, double arg2) float max(float arg1, float arg2) int max(int arg1, int arg2) long max(long arg1, long arg2)</pre>	Retorna o maior dos dois argumentos

O seguinte programa, `BasicMathDemo`, ilustra como usar alguns desses métodos:

```
public class BasicMathDemo {
    public static void main(String[] args) {
        double a = -191.635;
        double b = 43.74;
        int c = 16, d = 45;

        System.out.printf("The absolute value of %.3f is %.3f\n", a, Math.abs(a));
        System.out.printf("The ceiling of %.2f is %.0f\n", b, Math.ceil(b));
        System.out.printf("The floor of %.2f is %.0f\n", b, Math.floor(b));
        System.out.printf("The rint of %.2f is %.0f\n", b, Math.rint(b));
        System.out.printf("The max of %d and %d is %d\n", c, d, Math.max(c, d));
        System.out.printf("The min of %d and %d is %d\n", c, d, Math.min(c, d));
    }
}
```

Aqui está a saída do programa:

```
The absolute value of -191.635 is 191.635
The ceiling of 43.74 is 44
The floor of 43.74 is 43
The rint of 43.74 is 44
The max of 16 and 45 is 45
The min of 16 and 45 is 16
```

Métodos Exponenciais e Logarítmicos.

A próxima tabela lista métodos exponenciais e logarítmicos da classe `Math`:

Métodos Exponenciais e Logarítmicos	
Método	Descrição
<code>double exp(double d)</code>	Retorna a base dos logaritmos naturais, e, para a potência do argumento.
<code>double log(double d)</code>	Retorna o logaritmo natural do argumento
<code>double pow(double base, double exponent)</code>	Retorna o valor do primeiro argumento elevado para a potência do segundo argumento
<code>double sqrt(double d)</code>	Retorna a raiz quadrada do argumento

O seguinte programa, `ExponentialDemo`, mostra o valor de `e`, então chama cada método listado na tabela anterior em números escolhidos arbitrariamente:

```
public class ExponentialDemo {
    public static void main(String[] args) {
        double x = 11.635;
        double y = 2.76;

        System.out.printf("The value of e is %.4f\n", Math.E);
        System.out.printf("exp(%.3f) is %.3f\n", x, Math.exp(x));
        System.out.printf("log(%.3f) is %.3f\n", x, Math.log(x));
        System.out.printf("pow(%.3f, %.3f) is %.3f\n", x, y, Math.pow(x, y));
        System.out.printf("sqrt(%.3f) is %.3f\n", x, Math.sqrt(x));
    }
}
```

Aqui está a saída:

```
The value of e is 2.7183
exp(11.635) is 112983.831
log(11.635) is 2.454
pow(11.635, 2.760) is 874.008
sqrt(11.635) is 3.411
```

Métodos Trigonométricos.

A classe `Math` fornece uma coleção de funções trigonométricas, que são resumidas na seguinte tabela. O valor passado em cada um desses métodos é um ângulo expressado em radianos. Você pode usar o método `toRadians` para converter de graus para radianos.

Métodos Trigonométricos	
Método	Descrição
<code>double sin(double d)</code>	Retorna o seno do valor <i>double</i> especificado
<code>double cos(double d)</code>	Retorna o cosseno do valor <i>double</i> especificado
<code>double tan(double d)</code>	Retorna a tangente do valor especificado <i>double</i>
<code>double asin(double d)</code>	Retorna o arcosseno do valor especificado <i>double</i>
<code>double acos(double d)</code>	Retorna o arcocosseno do valor especificado <i>double</i>
<code>double atan(double d)</code>	Retorna a arcotangente do valor especificado <i>double</i>
<code>double atan2(double y, double x)</code>	Converte coordenadas retangulares (<i>x</i> , <i>y</i>) para coordenadas polares (<i>r</i> , <i>theta</i>) e retorna <i>theta</i>
<code>double toDegrees(double d)</code> <code>double toRadians(double d)</code>	Converte o argumento para graus ou radianos

Aqui está um programa, `TrigonometricDemo`, que usa cada um desses métodos para calcular vários valores trigonométricos para um ângulo de 45 graus:

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;
        double radians = Math.toRadians(degrees);

        System.out.format("The value of pi is %.4f%n", Math.PI);
        System.out.format("The sine of %.1f degrees is %.4f%n",
            degrees, Math.sin(radians));
        System.out.format("The cosine of %.1f degrees is %.4f%n",
            degrees, Math.cos(radians));
        System.out.format("The tangent of %.1f degrees is %.4f%n",
            degrees, Math.tan(radians));
        System.out.format("The arcsine of %.4f is %.4f degrees %n",
            Math.sin(radians),
            Math.toDegrees(Math.asin(Math.sin(radians))));
        System.out.format("The arccosine of %.4f is %.4f degrees %n",
            Math.cos(radians),
            Math.toDegrees(Math.acos(Math.cos(radians))));
        System.out.format("The arctangent of %.4f is %.4f degrees %n",
            Math.tan(radians),
            Math.toDegrees(Math.atan(Math.tan(radians))));
    }
}
```

A saída do programa é:

```
The value of pi is 3.1416
The sine of 45.0 degrees is 0.7071
The cosine of 45.0 degrees is 0.7071
The tangent of 45.0 degrees is 1.0000
The arcsine of 0.7071 is 45.0000 degrees
The arccosine of 0.7071 is 45.0000 degrees
The arctangent of 1.0000 is 45.0000 degrees
```

Números Randômicos.

O método `random()` retorna um número selecionado pseudo-randômico entre 0.0 e 1.0. O alcance inclui 0.0 mas não 1.0. Em outras palavras: $0.0 \leq \text{Math.random()} < 1.0$. Para conseguir um alcance diferente, você pode executar cálculos no valor retornado pelo método `random`. Por exemplo, para gerar um inteiro entre 0 e 9, você pode escrever:

```
int number = (int) (Math.random() * 10);
```

Pela multiplicação do valor por 10, o alcance dos valores possíveis vem a ser $0.0 \leq \text{number} < 10.0$.

Usar `Math.random` é útil quando você precisa gerar um número simples randômico. Se você precisa gerar uma série de números randômicos, você precisa criar uma instância de `java.util.Random` e invocar os métodos nesse objeto para gerar números.

Caracteres.

Muitas vezes, se você estiver usando um simples valor de caracter, você usa o tipo `char`. Por exemplo:

```
char ch = 'a';
char uniChar = '\u0391'; // Unicode para o caracter ômega grego
char[] charArray = { 'a', 'b', 'c', 'd', 'e' }; // um array de caracteres
```

Há outras situações, no entanto, quando você precisa usar um `char` como um objeto – por exemplo, como um argumento do método onde um objeto é esperado. A linguagem de programação Java fornece uma classe empacotadora que “empacota” o `char` em um objeto `Character` para este propósito. Um objeto do tipo `Character` contém um campo simples, cujo tipo é `char`. Esta classe `Character` também oferece um número de métodos de classe úteis (i.e., estáticas) para manipulação de caracteres.

Você pode criar um objeto `Character` com o construtor `Character`:

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

O compilador Java também cria um objeto `Character` para você em outras circunstâncias. Por exemplo, se você passar um primitivo `char` em um método que espera um objeto, o compilador automaticamente converte o `char` para um `Character` para você. Esta característica é chamada **autoboxing** – ou **unboxing**, se a conversão vai pelo caminho inverso.

Aqui está um exemplo de **boxing**:

```
Character ch = 'a'; // o char primitivo 'a' é empacotado
                // no objeto Character
```

e aqui está um exemplo de ambos: empacotamento e desempacotamento:

```
Character test(Character c) {...} // parâmetros do método e
                                // retorno do tipo = objeto Character
char c = test('x'); // primitivo 'x' é empacotado pelo método test,
                   // retorno é desempacotado para char 'c'
```

Nota: A classe `Character` é imutável, dessa forma, uma vez que ela é criada, o objeto `Character` não pode ser mudado.

A seguinte tabela lista alguns dos métodos mais úteis na classe `Character`, mas não é completa. Para consultar uma lista completa dos métodos nesta classe (há mais de 50), recorra à especificação **API** `java.lang.Character`.

Métodos Úteis na Classe <code>Character</code>	
Método	Descrição
<code>boolean isLetter(char ch)</code> <code>boolean isDigit(char ch)</code>	Determina se o valor <i>char</i> especificado é uma letra ou um dígito, respectivamente
<code>boolean isWhiteSpace(char ch)</code>	Determina se o valor <i>char</i> especificado é um espaço em branco
<code>boolean isUpperCase(char ch)</code> <code>boolean isLowerCase(char ch)</code>	Determina se o valor <i>char</i> especificado é uma letra maiúscula ou minúscula, respectivamente
<code>char toUpperCase(char ch)</code> <code>char toLowerCase(char ch)</code>	Retorna a letra maiúscula ou minúscula do valor <i>char</i> especificado
<code>toString(char ch)</code>	Retorna um objeto <i>string</i> representando o valor do caracter especificado – que é um caracter único <i>string</i>

Seqüências de Fuga (*Escape Sequences*).

Um caracter precedido por uma barra invertida (`\`) é uma seqüência de fuga e tem um significado especial para o compilador. Um caracter de nova linha (`\n`) tem sido usado freqüentemente neste tutorial nas declarações `System.out.println()` para avançar para a próxima linha depois que a **string** é mostrada. A seguinte tabela mostra as seqüências de fuga Java:

Seqüências de Fuga (<i>Escape Sequences</i>)	
Seqüência de Fuga	Descrição
<code>\t</code>	Inserir uma tabulação no texto neste ponto
<code>\b</code>	Inserir um retrocesso no texto neste ponto
<code>\n</code>	Inserir uma nova linha no texto neste ponto
<code>\r</code>	Inserir um retorno de carro no texto neste ponto
<code>\f</code>	Inserir uma alimentação de formulário no texto neste ponto
<code>\'</code>	Inserir um caracter aspa simples no texto neste ponto
<code>\"</code>	Inserir um caracter aspa dupla no texto neste ponto
<code>\\</code>	Inserir um caracter barra invertida no texto neste ponto

Quando uma seqüência de fuga é encontrada em uma declaração `print` (mostrar), o compilador interpreta isto adequadamente. Por exemplo, se você quer colocar aspas dentro de aspas, você precisa usar a seqüência de fuga, `\"`, no interior das aspas. Para mostrar a seqüência:

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

Voce precisa escrever:

```
System.out.println("She said \"Hello!\" to me.");
```

Strings.

Strings, que são amplamente usados na programação Java, são uma seqüência de caracteres. Na linguagem de programação Java, *strings* são objetos.

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

Criando Strings.

O caminho mais direto para criar uma *string* é escrever:

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

Neste caso, `"Hello World!"` é uma *string* literal – uma série de caracteres em seu código que é envolvida em aspas duplas. Quando é encontrado uma string literal em seu código, o compilador cria um objeto `String` com este valor – neste caso, `Hello World!`.

Como com qualquer outro objeto, você pode criar objetos `String` usando a palavra reservada `new` e o construtor. A classe `String` tem 11 construtores que fornecem a você um valor inicial da *string* usando fontes diferentes, como em um *array* de caracteres:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

A última linha do código mostra `hello`.

Nota: A classe `String` é imutável, então a partir do momento que um objeto `String` é criado ele não pode ser mudado. A classe `String` tem um número de métodos, alguns serão discutidos a seguir, que aparecem para modificar *strings*. Desde que *strings* são imutáveis, o que estes métodos realmente fazem é criar e retornar uma nova *string* que contém o resultado da operação.

String Length (Extensão da String).

Métodos usados para obter informação a respeito de um objeto são conhecidos como **métodos de acesso**. Um método de acesso que você pode usar com *strings* é o método `length()`, que retorna o número de caracteres contidos no objeto *string*. Depois das seguintes duas linhas de código serem executadas, `len` é igual a 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

Um palíndromo é uma palavra ou sentença que é simétrica – ela é soletrada da mesma forma para a frente e para trás, ignorando maiúsculas e pontuação. Aqui está um pequeno e ineficiente programa para reverter uma *string* palíndroma. Ela invoca o método `String.charAt(i)`, que retorna o *i*th caracter na *string*, contando do 0.

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];
        //coloca a string original em um array de chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] = palindrome.charAt(i);
        }
        // reverte o array de chars
        for (int j = 0; j < len; j++) {
            charArray[j] = tempCharArray[len - 1 - j];
        }
        String reversePalindrome = new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

A saída do programa é:

```
doT saw I was toD
```

Para realizar a reversão da **string**, o programa teve que converter a **string** em um **array** de caracteres (primeiro **loop for**), reverter o **array** em um segundo **array** (segundo **loop for**), e então converter novamente em uma **string**. A classe `String` inclui um método, `getChars()`, para converter uma **string**, ou uma porção de uma **string**, em um **array** de caracteres então nós poderíamos substituir o primeiro **loop for** no programa acima com

```
palindrome.getChars(0, len - 1, tempCharArray, 0);
```

Concatenando Strings.

A classe `String` inclui um método para concatenar duas **strings**:

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

Isto retorna uma nova **string** que é a `string1` com a `string2` adicionada a ela no final.

Você também pode usar o método `concat()` com **strings** literais, como em:

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

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

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

que resulta em:

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

O operador `+` é amplamente usado nas declarações `print`. Por exemplo:

```
String string1 = "saw I was ";
System.out.println("Dot " + string1 + "Tod");
```

que mostra:

```
Dot saw I was Tod
```

Desta maneira uma concatenação pode ser uma mistura de quaisquer objetos. Para cada objeto que não é uma `String`, o método `toString()` é chamado para converter para uma `String`.

Nota: A linguagem de programação Java não permite **strings** literais para ampliar linhas em códigos fontes, então você pode usar o operador de concatenação `+` no final de cada linha em uma **string** multi-linhas. Por exemplo:

```
String quote = "Now is the time for all good " +
    "men to come to the aid of their country.";
```

Quebrar **strings** entre linhas usando o operador de concatenação `+` é, novamente, muito comum em declarações `print`.

Criando Strings Formatadas.

Você tem visto o uso dos métodos `printf()` e `format()` para mostrar a saída com números formatados. A classe `String` tem um método de classe equivalente, `format()`, que retorna um objeto `String` melhor que um objeto `PrintStream`.

Usando método estático `format()` de `String` permite a você criar uma **string** formatada que você pode reusar, em oposição a uma antiga declaração `print`. Por exemplo, ao invés de

```
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);
```

Conversão Entre Números e Strings.

Convertendo Strings em Números.

Freqüentemente, um programa acaba com dados numéricos em um objeto *string* – um valor entrado pelo usuário, por exemplo.

As subclasses *Number* que envolvem tipos numéricos primitivos (*Byte*, *Integer*, *Double*, *Float*, *Long* e *Short*) cada uma fornece um método da classe nomeada *valueOf* que converte uma *string* em um objeto daquele tipo. Aqui está um exemplo, *ValueOfDemo*, que pega duas *strings* da linha de comando, converte elas em números e efetua operações matemáticas nesses valores:

```
public class ValueOfDemo {
    public static void main(String[] args) {

        // este programa requer dois argumentos na linha de comando
        if(args.length == 2) {
            //converte strings em números
            float a = (Float.valueOf(args[0]) ).floatValue();
            float b = (Float.valueOf(args[1]) ).floatValue();

            // faz alguns cálculos
            System.out.println("a + b = " + (a + b) );
            System.out.println("a - b = " + (a - b) );
            System.out.println("a * b = " + (a * b) );
            System.out.println("a / b = " + (a / b) );
            System.out.println("a % b = " + (a % b) );
        } else {
            System.out.println("This program requires two command-lines"+
                " arguments.");
        }
    }
}
```

A saída do programa quando você usa **4.5** e **87.2** na linha de comando é:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b = 4.5
```

Nota: Cada uma das subclasses *Number* que envolvem tipos primitivos numéricos também fornece um método *parseXXXX()* (por exemplo, *parseFloat()*), que pode ser usado para converter *strings* em números primitivos. Desde que um tipo primitivo é retornado ao invés de um objeto, o método *parseFloat()* é mais direto que o método *valueOf()*. Por exemplo, no programa *ValueOfDemo*, nós poderíamos usar:

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Convertendo Números em Strings.

Algumas vezes você precisa converter um número em uma string porque você precisa operar no valor em esta forma de **string**. Há muitas maneiras fáceis para converter um número em uma string:

```
int e;
String s1 = "" + i; // Concatena "i" com uma string vazia;
                  // Conversão é manobrada por você
```

ou

```
String s2 = String.valueOf(i); // O método da classe valueOf
```

Cada uma das subclasses de **Number** inclui um método da classe, **toString()**, que converterá este tipo primitivo em uma **string**. Por exemplo:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

O exemplo **ToStringDemo** usa o método **toString** para converter um número em uma **string**. O programa então usa alguns métodos **string** para calcular o número de dígitos antes e depois do ponto decimal:

```
public class ToStringDemo {

    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int dot = s.indexOf('.');

        System.out.println(dot + " digits before decimal point.");
        System.out.println( (s.length() - dot - 1) +
            " digits after decimal point.");
    }
}
```

A saída do programa é:

```
3 digits before decimal point.
2 digits after decimal point.
```

//=====

Manipulando Caracteres em uma String.

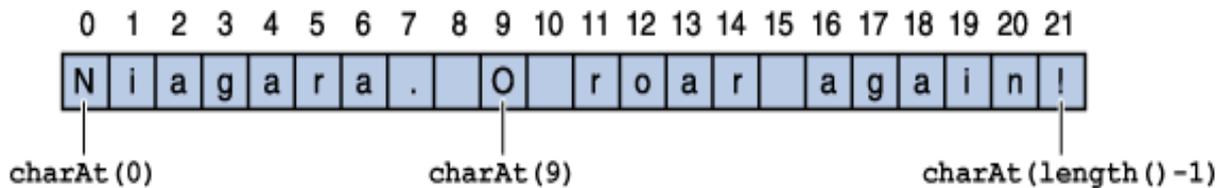
A classe `String` tem um número de métodos para examinar o conteúdo de *strings*, encontrando caracteres ou *substrings* dentro de uma *string*, mudando *case* (maiúscula ou minúscula), e outras tarefas.

Pegando Caracteres e Substrings pelo Índice.

Você pode pegar o caracter de um índice em particular dentro de uma *string* invocando o método de acesso `charAt()`. O índice do primeiro caracter é `0`, enquanto o índice do último caracter é `length() - 1`. Por exemplo, o seguinte código pega o caracter no índice `9` em uma *string*:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Índices começam com `0`, então o caracter no índice `9` é 'o', como ilustrado na seguinte figura:

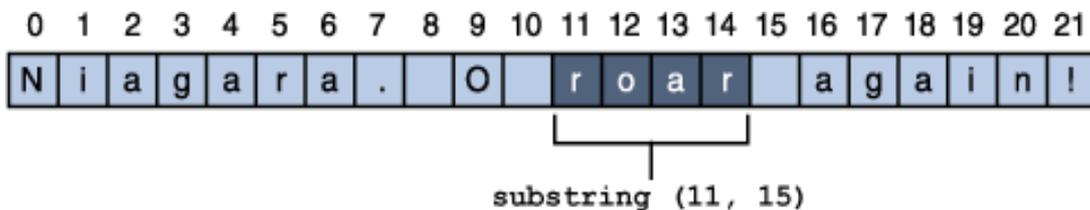


Se você quer pegar mais que um caracter consecutivo de uma *string*, você pode usar o método `substring`. O método `substring` tem duas versões, como mostra a seguinte tabela:

Os Métodos <code>substring</code> na classe <code>String</code> .	
Método	Descrição
<code>String substring(int beginIndex, int endIndex)</code>	Retorna uma nova <i>string</i> que é uma <i>substring</i> dessa <i>string</i> . O primeiro argumento inteiro especifica o índice do primeiro caracter. O segundo argumento integer é o índice do último caracter + 1.
<code>String substring(int beginIndex)</code>	Retorna uma nova <i>string</i> que é uma <i>substring</i> dessa <i>string</i> . O argumento integer especifica o índice do primeiro caracter. Aqui, a <i>substring</i> retornada estende-se para o final da <i>string</i> original.

O seguinte código pega do palíndromo `Niagara` a *substring* que estende-se do índice `11` acima, mas não incluindo o índice `15`, que forma a palavra "roar":

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



Outros Métodos para Manipular Strings.

Aqui estão alguns outros métodos `String` para manipulação de *strings*:

Outros Métodos na Classe <code>String</code> para Manipulação de <i>Strings</i>	
Método	Descrição
<pre>String[] split(String regex) String[] split(String regex, int limit)</pre>	Procura por um igual como especificado pelo argumento <i>string</i> (que contém uma expressão regular) e divide esta <i>string</i> em um <i>array</i> de <i>strings</i> correspondente. O argumento <i>integer</i> opcional especifica o tamanho máximo do <i>array</i> retornado.
<pre>CharSequence subSequence(int beginIndex, int endIndex)</pre>	Retorna uma nova seqüência de caracteres construída do índice <code>beginIndex</code> até <code>endIndex - 1</code> .
<pre>String trim()</pre>	Retorna uma cópia dessa <i>string</i> com espaços em branco removidos
<pre>String toLowerCase() String toUpperCase()</pre>	Retorna uma cópia dessa <i>string</i> convertida para letras minúsculas ou maiúsculas. Se nenhuma conversão for necessária, esses métodos retornam a <i>string</i> original.

Procurando por Caracteres e *Substrings* em uma *String*.

Há alguns outros métodos para encontrar caracteres ou *substrings* dentro de uma *string*. A classe `String` fornece métodos de acesso que retornam a posição dentro da *string* de um caracter específico ou *substring*: `indexOf()` e `lastIndexOf()`. O método `indexOf()` procura do começo para frente da *string*, e o método `lastIndexOf()` procura do final para trás da *string*. Se um caracter ou *substring* não é encontrado, `indexOf()` e `lastIndexOf()` retornam -1.

A classe `String` também fornece um método de busca, `contains`, que retorna *true* se a *string* contém uma seqüência de caracteres em particular. Use este método quando você somente precisar saber que a *string* contém a seqüência de caracteres, mas a localização precisa não seja importante.

A seguinte tabela descreve os vários métodos de busca em *strings*:

Os Métodos de Busca na Classe <code>String</code>	
Método	Descrição
<pre>int indexOf(int ch) int lastIndexOf(int ch)</pre>	Retorna o índice da primeira (última) ocorrência do caracter especificado
<pre>int indexOf(int ch, int fromIndex) int lastIndexOf(int ch, int fromIndex)</pre>	Retorna o índice da primeira (última) ocorrência do caracter especificado, procurando para a frente (para trás) do índice especificado
<pre>int indexOf(String str) int lastIndexOf(String str)</pre>	Retorna o índice da primeira (última) ocorrência da <i>substring</i> especificada
<pre>int indexOf(String str, int fromIndex) int lastIndexOf(String str, int fromIndex)</pre>	Retorna o índice da primeira (última) ocorrência da <i>substring</i> especificada, procurando para a frente (para trás) do índice especificado.
<pre>boolean contains(CharSequence s)</pre>	Retorna <i>true</i> se a <i>string</i> contém a seqüência de caracteres especificada

Nota: `CharSequence` é uma interface que é implementada pela classe `String`. Por essa razão, você pode usar uma *string* como um argumento do método `contains()`.

Substituindo Caracteres e *Substrings* em uma *String*.

A classe `String` tem poucos métodos para inserção de caracteres ou *substrings* em uma *string*. Em geral, não há necessidade: você pode criar uma nova *string* pela concatenação de *substrings*.

A classe `String` tem quatro métodos para substituir caracteres ou *substrings* encontrados, todavia. Eles são:

Métodos para a Manipulação de <i>Strings</i> na Classe <code>String</code>	
Método	Descrição
<code>String replace(char oldChar, char newChar)</code>	Retorna uma nova <i>string</i> resultante da substituição de todas as ocorrências de <code>oldChar</code> nesta <i>string</i> com <code>newChar</code> .
<code>String replace(CharSequence target, CharSequence replacement)</code>	Substitui cada <i>substring</i> da <i>string</i> que é igual à seqüência alvo com a seqüência de substituição literal especificada.
<code>String replaceAll(String regex, String replacement)</code>	Substitui cada <i>substring</i> da <i>string</i> que é igual à expressão regular dada com a substituição dada.
<code>String replaceFirst(String regex, String replacement)</code>	Substitui a primeira <i>substring</i> dessa <i>string</i> que é igual à expressão regular dada com a substituição dada.

Um Exemplo.

A seguinte classe, `Filename`, ilustra o uso de `lastIndexOf()` e `substring()` para isolar diferentes partes de um nome de arquivo.

Nota: Os métodos na seguinte classe `Filename` não checam qualquer erro e assumem que seus argumentos contêm um caminho de diretório completo com uma extensão. Se esses métodos produzissem código, eles poderiam verificar que seus argumentos estavam construídos adequadamente.

```
/**
 * Esta classe assume que a string usada para inicializar
 * o caminho completo tem um caminho de diretório, nome de arquivo, e extensão.
 * Os métodos não funcionam se isto não existir.
 */
public class Filename {
    private String fullPath;
    private char pathSeparator, extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }
    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }
    public String filename() { // pega o nome do arquivo sem a extensão
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }
    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

Aqui está um programa, `FileNameDemo`, que constrói um objeto `Filename` e chama todos os seus métodos:

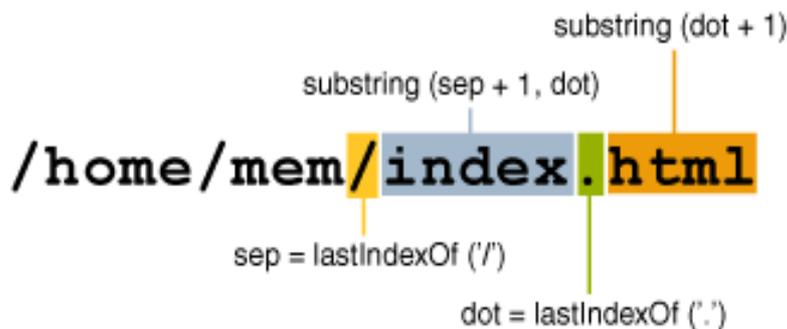
```
public class FileNameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/mem/index.html";
        Filename myHomePage = new Filename(FPATH, '/', '.');

        System.out.println("Extension = " + myHomePage.extension());
        System.out.println("Filename = " + myHomePage.filename());
        System.out.println("Path = " + myHomePage.path());
    }
}
```

E aqui está a saída do programa:

```
Extension = html
Filename = index
Path = /home/mem
```

Como mostra a seguinte figura, seu método `extension` usa `lastIndexOf` para descobrir a última ocorrência do ponto (`.`) no nome do arquivo. Então `substring` usa o valor retornado de `lastIndexOf` para extrair a extensão do nome do arquivo – isto é, a `substring` do ponto até o final da `string`. Este código assume que o nome do arquivo tem um ponto nele; se o nome não tem um ponto, `lastIndexOf` retorna -1, e o método da `substring` lança uma `StringIndexOutOfBoundsException`.



Além disso, repare que o método `extension` usa `dot + 1` como o argumento para `substring`. Se o caracter ponto (`.`) é o último caracter da `string`, `dot + 1` é igual ao tamanho da `string`, que é a extensão do maior índice na `string` (porque índices iniciam com 0). Este é um argumento legal para `substring` porque este método aceita um índice igual a, mas não maior que, o tamanho da `string` e interpreta isto para significar “o final da string”.

Comparando Strings e Porções de Strings.

A classe `String` tem um número de métodos para comparar *strings* e porções de *strings*. A seguinte tabela lista esses métodos:

Métodos para Comparação de <i>Strings</i>	
Método	Descrição
<code>boolean endsWith(String suffix)</code> <code>boolean startsWith(String prefix)</code>	Retorna <code>true</code> se esta <i>string</i> termina ou começa com a <i>substring</i> especificada como um argumento para o método
<code>boolean startsWith(String prefix, int offset)</code>	Considera a <i>string</i> começando pelo índice <code>offset</code> , e retorna <code>true</code> se ela começa com a <i>substring</i> especificada como um argumento
<code>int compareTo(String anotherString)</code>	Compara duas <i>strings</i> lexicograficamente. Retorna um inteiro indicando se esta <i>string</i> é maior que (resultado é > 0), igual a (resultado é = 0), ou menor que (resultado é < 0) o argumento.
<code>int compareToIgnoreCase(String str)</code>	Compara duas <i>strings</i> lexicograficamente, ignorando diferenças entre minúsculas e maiúsculas. Retorna um inteiro indicando se esta <i>string</i> é maior que (resultado é > 0), igual a (resultado é = 0), ou menor que (resultado é < 0) o argumento
<code>boolean equals(Object anObject)</code>	Retorna <code>true</code> se e somente se o argumento é um objeto <code>String</code> que representa a mesma seqüência de caracteres que este objeto.
<code>boolean equalsIgnoreCase(String anotherString)</code>	Retorna <code>true</code> se e somente se o argumento é um objeto <code>String</code> que representa a mesma seqüência de caracteres que este objeto, ignorando diferenças entre minúsculas e maiúsculas.
<code>boolean regionMatches(int toffset, String other, int ooffset, int len)</code>	Testa se a região especificada da <i>string</i> é igual à região do argumento da <code>String</code> . Região é o tamanho <code>len</code> e começa no índice <code>toffset</code> para esta <i>string</i> e <code>ooffset</code> para a outra <i>string</i> .
<code>boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)</code>	Testa se a região especificada da <i>string</i> é igual à região especificada do argumento da <code>String</code> . Região é o tamanho <code>len</code> e começa no índice <code>toffset</code> para esta <i>string</i> e <code>ooffset</code> para outra <i>string</i> . O argumento <code>boolean</code> indica se diferenças entre caracteres minúsculos e maiúsculos podem ser ignorados, se verdadeiro, as diferenças são ignoradas quando comparando caracteres.
<code>boolean matches(String regex)</code>	Testa se esta <i>string</i> é igual à expressão regular.

O seguinte programa, `RegionMatchesDemo`, usa o método `regionMatches` para procurar por uma *string* dentro de outra *string*:

```
public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham";
        String findMe = "Eggs";
        int searchMeLength = searchMe.length();
        int findMeLength = findMe.length();
        boolean foundIt = false;
        for (int i = 0; i <= (searchMeLength - findMeLength); i++) {
            if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
                foundIt = true;
                System.out.println(searchMe.substring(i, i + findMeLength));
                break;
            }
        }
        if (!foundIt) System.out.println("No match found.");
    }
}
```

A saída do programa é `Eggs`.

O programa passa através da *string* referida por `searchMe` um caracter de cada vez. Para cada caracter, o programa chama o método `regionMatches` para determinar se a *substring* começando com o caracter corrente é igual à *string* que o programa está olhando.

A Classe `StringBuilder`.

Objetos `StringBuilder` são como objetos `String`, exceto que eles podem ser modificados. Internamente, esses objetos são tratados como *arrays* de tamanho variável que contêm uma seqüência de caracteres. Em qualquer ponto, o tamanho e conteúdo da seqüência pode ser mudado por meio da invocação de métodos.

Strings podem sempre ser usadas a menos que construtores de *string* ofereçam uma vantagem em termos de simplificação do código (veja o programa exemplo no fim desta seção) ou melhorar a performance. Por exemplo, se você precisa concatenar um grande número de *strings*, escolher um objeto `StringBuilder` é mais eficiente.

Tamanho e Capacidade.

A classe `StringBuilder`, como a classe `String`, tem um método `length()` que retorna o tamanho da seqüência de caracteres no construtor.

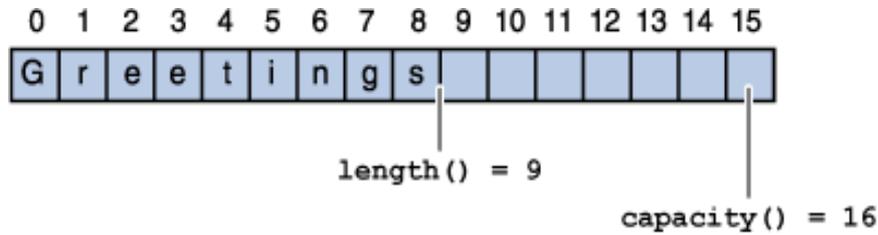
Diferente de *strings*, cada construtor *string* também tem uma capacidade, o número de espaços de caracteres que estão alocados. A capacidade, que é retornada pelo método `capacity()`, é sempre maior ou igual que o tamanho (usualmente maior que) e automaticamente expandirá de acordo com a necessidade para acomodar adições para o construtor *string*.

Construtores <code>StringBuilder</code>	
Construtor	Descrição
<code>StringBuilder()</code>	Cria um construtor <i>string</i> vazio com a capacidade de 16 (16 elementos vazios)
<code>StringBuilder(CharSequence cs)</code>	Cria um construtor <i>string</i> contendo os mesmos caracteres que a <code>CharSequence</code> especificada, mais um extra de 16 elementos vazios seguindo a <code>CharSequence</code> .
<code>StringBuilder(int initCapacity)</code>	Cria um construtor <i>string</i> com a capacidade inicial especificada.
<code>StringBuilder(String s)</code>	Cria um construtor <i>string</i> cujo valor é inicializado pela <i>string</i> especificada, mais um extra de 16 elementos vazios seguindo a <i>string</i> .

Por exemplo, o seguinte código:

```
StringBuilder sb = new StringBuilder();//cria um construtor vazio, capacidade 16
sb.append("Greetings");//adiciona 9 caracteres string no começo
```

produzirá um construtor **string** com um tamanho 9 e uma capacidade 16:



A classe `StringBuilder` tem alguns métodos associados para tamanho e capacidade que a classe `String` não tem:

Métodos de Tamanho e Capacidade	
Método	Descrição
<code>void setLength(int newLength)</code>	Estabelece o tamanho da seqüência de caracteres. Se <code>newLength</code> é menor que <code>length()</code> , o último caracter na seqüência está truncado (incompleto). Se <code>newLength</code> é maior que <code>length()</code> , caracteres nulos são adicionados ao final da seqüência de caracteres.
<code>void ensureCapacity(int minCapacity)</code>	Garante que a capacidade é ao menos igual que o mínimo especificado.

Um número de operações (por exemplo, `append()`, `insert()`, ou `setLength()`) pode aumentar o tamanho da seqüência de caracteres no construtor **string** a fim de que o `length()` (tamanho) resultante seja maior que a `capacity()` (capacidade) resultante. Quando isso acontece, a capacidade é automaticamente aumentada.

Operações StringBuilder.

As principais operações em um `StringBuilder` que não são disponíveis em `String` são os métodos `append()` e `insert()`, que são sobrecarregados assim como aceitam dados de qualquer tipo. Cada um converte este argumento em uma *string* e então anexa ou insere os caracteres dessa *string* para a seqüência de caracteres no construtor *string*. O método `append()` sempre adiciona esses caracteres no final da seqüência de caracteres, enquanto o método `insert()` adiciona os caracteres para o ponto especificado.

Aqui está uma série de métodos da classe `StringBuilder`:

Diversos Métodos StringBuilder	
Método	Descrição
<pre>StringBuilder append(boolean b) StringBuilder append(char c) StringBuilder append(char[] str) StringBuilder append(char[] str, int offset, int len) StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s)</pre>	<p>Anexa o argumento para o construtor <i>string</i>. O dado é convertido para uma <i>string</i> antes da operação <code>append()</code> acontecer.</p>
<pre>StringBuilder delete(int start, int end) StringBuilder deleteCharAt(int index)</pre>	<p>Deleta o(s) caracter(s) especificado neste construtor <i>string</i>.</p>
<pre>StringBuilder insert(int offset, boolean b) StringBuilder insert(int offset, char c) StringBuilder insert(int offset, char[] str) StringBuilder insert(int offset, char[] str, int offset, int len) StringBuilder insert(int offset, double d) StringBuilder insert(int offset, float f) StringBuilder insert(int offset, int i) StringBuilder insert(int offset, long lng) StringBuilder insert(int offset, Object obj) StringBuilder insert(int offset, String s)</pre>	<p>Insero o segundo argumento no construtor <i>string</i>. O primeiro argumento <i>integer</i> indica o índice anterior onde o dado está para ser inserido. O dado é convertido em uma <i>string</i> antes da operação de inserção acontecer.</p>
<pre>StringBuilder replace(int start, int end, String s) void setCharAt(int index, char c)</pre>	<p>Substitui o(s) caracter(s) especificado no construtor <i>string</i>.</p>
<pre>StringBuilder reverse()</pre>	<p>Reverte o(s) caracter(s) especificado no construtor <i>string</i>.</p>
<pre>String toString()</pre>	<p>Retorna uma <i>string</i> que contém a seqüência de caracteres no construtor.</p>

Nota: Você pode usar qualquer método `String` em um objeto `StringBuilder` primeiro convertendo o construtor *string* em uma *string* com o método `toString()` da classe `StringBuilder`. Então convertendo a *string* de volta em um construtor *string* usando o construtor `StringBuilder(String str)`.

Um Exemplo de `StringBuilder`.

O programa `StringDemo` que foi relacionado na seção intitulada “**Strings**” é um exemplo de um programa que poderia ser mais eficiente se uma `StringBuilder` fosse usado ao invés de uma `String`.

`StringDemo` reverte um palíndromo. Aqui, novamente, ele está registrado:

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // coloca a string original em um array de chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] = palindrome.charAt(i);
        }

        // reverte o array de chars
        for (int j = 0; j < len; j++) {
            charArray[j] = tempCharArray[len - 1 - j];
        }

        String reversePalindrome = new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Rodando o programa é produzida esta saída:

```
doT saw I was toD
```

Para executar a reversão da **string**, o programa converte a **string** em um **array** de caracteres (primeiro **loop for**), reverte o **array** em um segundo **array** (segundo **loop for**), e então converte novamente em uma **string**.

Se você converter a **string palindrome** em um construtor **string**, você pode usar o método `reverse()` na classe `StringBuilder`. Isto faz o código simples e fácil de ler:

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";

        StringBuilder sb = new StringBuilder(palindrome);
        sb.reverse(); // reverte isto

        System.out.println(sb);
    }
}
```

Rodando o programa é produzida a mesma saída:

```
doT saw I was toD
```

Note que `println()` mostra o construtor **string**, como em:

```
System.out.println(sb);
```

porque `sb.toString()` é chamado implicitamente, assim como com qualquer outro objeto em uma invocação `println()`.

Nota: Há também uma classe `StringBuffer` que é exatamente a mesma que a classe `StringBuilder`, exceto que ela é enfileirada a salvo pela virtude de ter seus métodos sincronizados. Filas serão discutidas na lição sobre concorrência.