PROJETO ESCOLAS - REFERÊNCIA
Compromisso com a Excelência na Escola Pública

Cadernos de Informatica

CURSO DE CAPACITAÇÃO EM INFORMÁTICA INSTRUMENTAL

CURSO DE MONTAGEM E MANUTENÇÃO DE COMPUTADORES

CURSO SOBRE O SISTEMA OPERACIONAL LINUX

CURSO DE PROGRAMAÇÃO EM JAVA

CURSO DE INTRODUÇÃO A BANCOS DE DADOS

CURSO DE CONSTRUÇÃO DE WEB SITES

CURSO DE EDITORAÇÃO ELETRÔNICA

CURSO DE ILUSTRAÇÃO DIGITAL

CURSO DE PRODUÇÃO FONOGRÁFICA

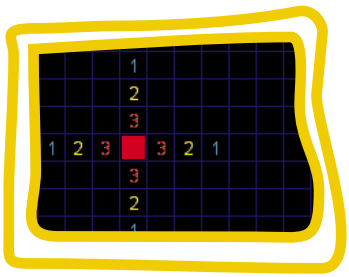
CURSO DE COMPUTAÇÃO GRÁFICA 3D

CURSO DE PROJETO AUXILIADO POR COMPUTADOR

CURSO DE MULTIMÍDIA APLICADA À EDUCAÇÃO

Cadernos de Informatica

CURSO DE PROGRAMAÇÃO EM JAVA



Lucilia Camarão de Figueiredo
Coordenador
Carlos Eduardo Hermeto de Sá Motta

Apresentação

Os computadores que estão sendo instalados pela SEE nas escolas estaduais deverão ser utilizados para propósitos administrativos e pedagógicos. Para isso, desenvolveu-se um conjunto de cursos destinados a potencializar a utilização desses equipamentos. São doze cursos que estão sendo disponibilizados para as escolas para enriquecimento do seu plano curricular. Esses cursos não são profissionalizantes. São cursos introdutórios, de formação inicial para o trabalho, cujo objetivo é ampliar o horizonte de conhecimentodos alunos para facilitar a futura escolha de uma profissão.

Todos os cursos foram elaborados para serem realizados em 40 módulos-aula, cada um deles podendo ser desenvolvidos em um semestre (com 2 módulos-aula semanais) ou em 10 semanas (com 4 módulos-aula semanais). Em 2006, esses cursos deverão ser oferecidos para os alunos que desejarem cursá-los, em caráter opcional e horário extra-turno.

Em 2007, eles cursos deverão ser incluídos na matriz curricular da escola, na série ou séries por ela definida, integrando a Parte Diversificada do currículo.

Esses cursos foram concebidos para dar aos professores, alunos e funcionários uma dimensão do modo como o computador influencia, hoje, o nosso modo de vida e os meios de produção. Para cada curso selecionado pela escola deverão ser indicados pelo menos dois ou, no máximo, três professores (efetivos, de preferência) para serem capacitados pela SEE. Esses professores irão atuar como multiplicadores, ministrando-os a outros servidores da escola e aos alunos.

Curso de Capacitação em Informática Instrumental

Este curso será implantado obrigatoriamente em todas as escolas estaduais em que for instalado laboratório de informática. Iniciando pelas Escolas-Referência, todos os professores e demais servidores serão capacitados para que possam fazer uso adequado e proveitoso desses equipamentos tanto na administração da escola como nas atividades didáticas.

É um curso voltado para a desmistificação da tecnologia que está sendo implantada. O uso do computador ainda é algo difícil para muitas pessoas que ainda não estão muito familiarizadas com essas novas tecnologias que estão ocupando um espaço cada vez maior na escola e na vida de todos. Este curso vai motivar os participantes para uma aproximação com essas tecnologias, favorecendo a transformação dos recursos de informática em instrumentos de produção e integração entre gestores, professores e demais servidores. As características dos equipamentos e as funcionalidades dos programas serão apresentadas de maneira gradual e num contexto prático. Essas.situações práticas serão apresentadas de maneira que o participante perceba o seu objetivo e o valor de incorporá-las ao seu trabalho cotidiano. Os participantes serão preparados para navegar e pesquisar na internet; enviar, receber e administrar correspondência

eletrônica, além de criar e editar documentos (textos, planilhas e apresentações) de interesse acadêmico e profissional. Esse é um curso fundamental, base e pré-requisito para todos os demais.

Curso de Montagem e Manutenção de Computadores

Este curso será implantado em, pelo menos, uma escola do município sede de cada Superintendência Regional de Ensino. A indicação da escola deverá ser feita pela própria S.R.E, levando-se em conta as condições de infra-estrutura nas Escolas-Referência existentes no município. Nas escolas escolhidas será montado um laboratório de informática especialmente para a oferta desse curso.

O objetivo deste curso é capacitar tecnicamente os alunos de ensino médio que queiram aprender a montar, fazer a manutenção e configurar microcomputadores. Pode ser oferecido para alunos de outras escolas, para professores e demais servidores da escola e para a comunidade, aos finais de semana ou horários em que o laboratório esteja disponível.

Neste curso o participante aprenderá a função de cada um dos componentes do microcomputador. Aprenderá como montar um computador e como configurá-lo, instalando o sistema operacional, particionando e formatando discos rígidos, instalando placas de fax/modem, rede, vídeo, som e outros dispositivos. Conhecerá, ainda, as técnicas de avaliação do funcionamento e configuração de microcomputadores que esteja precisando de manutenção preventiva ou corretiva, além de procedimentos para especificação de um computador para atender as necessidades requeridas por um cliente.

Dos cursos que se seguem, as Escolas-Referência deverão escolher pelo menos dois para implantar em 2006.

No período de 13 a 25 de março/2006, estará disponível no sítio da SEE (www.educacao.mg.gov.br) um formulário eletrônico para que cada diretor das Escolas-Referência possa informar quais os cursos escolhidos pela sua escola e quais os professores que deverão ser capacitados. Durante o período de capacitação, os professores serão substituídos por professores-designados para que as atividades didáticas da escola não sejam prejudicadas.

1. Curso sobre o Sistema Operacional Linux

É destinado àqueles que desejam conhecer ferramentas padrão do ambiente Unix. É um curso voltado para a exploração e organização de conteúdo. São ferramentas tipicamente usadas por usuários avançados do sistema operacional. Tem por finalidade apresentar alguns dos programas mais simples e comuns do ambiente; mostrar que, mesmo com um conjunto pequeno de programas, é possível resolver problemas reais; explicar

a comunicação entre programas via rede e estender o ambiente através de novos programas. O texto didático deste curso apresenta os recursos a serem estudados e propõe exercícios. É um curso para aqueles que gostam de enfrentar desafios.

Ementa: Histórico e desenvolvimento do Unix e Linux. Login no computador. Explorando o computador (processos em execução, conexões abertas). Descrição dos conceitos de arquivo e diretório. Operações simples sobre arquivos e diretórios. Sistema de permissões e quotas.

Procurando arquivos e fazendo backups. Executando e controlando programas. Processamnto de texto. Expressões regulares. Estendendo o ambiente. Trabalho em rede. Um sistema de chat. Comunicação segura no chat (criptografia). Ainda criptografia. Sistema de arquivos como um Banco de Dados. Um programa gráfico. Programando para rede.

2. Curso de Programação Em Java

É um curso de programação introdutório que utiliza a linguagem Java. Essa linguagem se torna, a cada dia, mais popular entre os programadores profissionais. O curso foi desenvolvido em forma de tutorial. O participante vai construir na prática um aplicativo completo (um jogo de batalha naval) que utiliza o sistema gráfico e que pode ser utilizado em qualquer sistema operacional. Os elementos de programação são apresentados em atividades práticas à medida em que se fazem necessários. Aqueles que desejam conhecer os métodos de produção de programas de computadores terão, nesse curso, uma boa visão do processo.

Ementa: Conceitos de linguagem de programação, edição, compilação, depuração e execução de programas. Conceitos fundamentais de linguagens de programação orientada a objetos.

Tipos primitivos da linguagem Java, comandos de atribuição e comandos de repetição. Conceito de herança e programação dirigida por eventos. Tratamento de eventos. Programação da interface gráfica. *Array*s. Números aleatórios.

3. Curso de Introdução ao Bancos de Dados

Este curso mostrará aos participantes os conceitos fundamentais do armazenamento, gerenciamento e pesquisa de dados em computadores. Um banco de dados é um repositório de informações que modelam entidades do mundo real. O Sistema Gerenciador do Banco de Dados permite introduzir, modificar, remover, selecionar e organizar as informações armazenadas. O curso mostra como os bancos de dados são criados e estruturados através de exemplos práticos. Ao final, apresenta os elementos da linguagem SQL (Structured Query Language – Linguagem Estruturada de Pesquisa) que é uma

linguagem universal para gerenciamento de informações de bancos de dados e os elementos básicos da administração desses repositórios de informação..Apesar de ser de nível introdutório, o curso apresenta todos os tópicos de interesse relacionados à área. É um curso voltado para aqueles que desejam conhecer os sistemas que gerenciam volumes grandes e variados de informações, largamente utilizados no mundo empresarial.

Ementa: Modelagem de dados. Normalização. Linguagem SQL. Mecanismos de consulta. Criação e alteração de tabelas. Manipulação e formatação de dados. Organização de resultados de pesquisa. Acesso ao servidor de bancos de dados. Contas de usuários. Segurança. Administração de bancos de dados. Manutenção. Integridade.

4. Curso de Construção de Web Sites

Este curso mostrará aos participantes como construir páginas HTML que forma a estrutura de um "site" na internet. A primeira parte do curso é voltada para a construção de páginas; a segunda parte, para a estruturação do conjunto de páginas que formação o "site", incluindo elementos de programação. Explicará os conceitos elementares da web e mostrará como é que se implementa o conjunto de páginas que forma o "site" num servidor.

Ementa: Linguagem HTML. Apresentação dos principais navegadors disponíveis no mercado.

Construção de uma página HTML simples respeitando os padrões W3C. Recursos de formatação de texto. Recursos de listas, multimídia e navegação. Tabelas e *Frame*s. Folha de Estilo. Elementos de Formulário. Linguagem Javascript. Interação do Javascript com os elementos HTML. Linguagem PHP. Conceitos de Transmissão de Site e critérios para avaliação de servidores.

1. Curso de Editoração Eletrônica

Voltado para a produção de documentos físicos (livros, jornais, revistas) e eletrônicos. Apresenta as ferramentas de produção de texto e as ferramentas de montagem de elementos gráficos numa página. O texto é tratado como elemento de composição gráfica, juntamente com a pintura digital, o desenho digital e outros elementos gráficos utilizados para promover a integração dos elementos gráficos.

O curso explora de maneira extensiva os conceitos relacionados à aparência do texto relativos aos tipos de impressão (fontes). Mostra diversos mecanismos de produção dos mais variados tipos de material impresso, de texto comum às fórmulas matemáticas. Finalmente, discute a metodologia de gerenciamento de documentos.

Ementa: Editor de textos. Formatadores de texto. Tipos e Fontes. Gerenciamento de projetos.

Publicações. Programas para editoração. Programas acessórios. Impressão. Desenvolvimento de um projeto.

2. Curso de Ilustração Digital

Desenvolvido sobre um único aplicativo de tratamento de imagens e pintura digital, o GIMP (GNU Image Manipulation Program – Programa de Manipulação de Imagens GNU).

Este curso ensina, passo a passo, como utilizar ferramentas do programa para produzir ilustrações de qualidade que podem ser utilizadas para qualquer finalidade. A pintura digital é diferente do desenho digital. O desenho se aplica a diagramas e gráficos, por exemplo. A pintura tem um escopo muito mais abrangente e é uma forma de criação mais livre, do ponto de vista formal. É basicamente a diferença que há entre o desenho artístico e o desenho técnico. É, portanto, um curso voltado para aqueles que têm interesses e vocações artísticas.

Ementa: A imagem digita*l.* Espaç*os* de core*s.* Digitalização de imagen*s.* Fotomontagem e colagem digital. Ferramentas de desenh*o.* Ferramentas de pintura. Finalização e saída.

3. Curso de Produção Fonográfica

Curso voltado para aqueles que têm interesse na produção musical. Explica, através de programas, como é que se capturam, modificam e agrupam os sons musicais para produzir arranjos musicais. É um curso introdutório com uma boa visão da totalidade dos procedimentos que levam à produção de um disco.

Ementa: O Fenômeno Sonoro. O Ambiente Sonoro. A Linguagem Musical. Pré-Produção. O Padrão MIDI. A Gravação. A Edição. Pós-processamento. Mixagem. Finalização.

4. Curso de Computação Gráfica

Curso introdutório de modelagem, renderização e animação de objetos tridimensionais.

Esse curso é a base para utilização de animações tridimensionais em filmes. Conduzido como um tutorial do programa BLENDER, apresenta a interface do programa e suas operações elementares. Destinado àqueles que têm ambições de produzir animações de alta qualidade para a educação ou para a mídia.

Ementa: Introdução à Computação Gráfica. Conceitos básicos 2D e 3D. Interface principal do programa Blender. Espaço de trabalho. Navegação em 3D. Modelagem em 3D. Primitivas básicas. Movimentação de objetos. Edição de objetos. Composição de cenas. Materiais e texturas. Aplicação de materiais. UV Mapping. Luzes e Câmeras. Iluminação de cena. Posicionamento e manipulação de câmera. Renderização still frame. Formatos

de saída. Animação básica. Movimentação de câmera e objetos. Renderização da animação. Formatos de saída.

5. Curso de Projeto Auxiliado Por Computador

Os programas de CAD (Computer Aided Design – Projeto Auxiliado por Computador) são utilizados para composição de desenhos técnicos. Diferentemente dos programas de pintura eletrônica (como o GIMP), fornecem ao usuário ferramentas para desenhar com precisão e anotar os desenhos de acordo com as normas técnicas. Além de ensinar ao usuário a utilizar um programa de CAD (QCad), o curso apresenta elementos básicos de desenho técnico e construções geométricas diversas visando preparar o participante para um aprimoramento em áreas típicas das engenharias e da arquitetura..Ementa: Informática aplicada ao desenho técnico. Conceitos básicos: construções geométricas, escalas, dimensionamento, projeções ortográficas e perspectivas. Sistemas de coordenadas cartesiano e polar. Novas entidades geométricas básicas: polígonos e círculos.

Operações geométricas básicas. Tipos de unidades de medida. Criação de um padrão de

formato. Organização de um desenho por níveis. Construções geométricas diversas. A teoria dos conjuntos aplicada ao desenho. Propriedades dos objetos. Edição do desenho.

Movimento, rotação, escalamento e deformação de objetos. Agrupamento de objetos em blocos.

6. Curso de Multimídia na Educação

O curso está dividido em três partes: a) utilização da multimídia no contexto educacional; b) autoria de apresentações multimídia; c) projetos de aprendizagem mediada por tecnologia. Este curso é o fundamento para a criação dos cursos de educação a distância.

Apresenta os elementos que compõem os sistemas de multimídia, as comunidades virtuais de aprendizagem, o planejamento e a preparação de uma apresentação e de uma lição de curso e, finalmente, a tecnologia de objetos de aprendizado multimídia.

Ementa: Introdução à Multimídia e seus componentes. Multimídia na Educação. Comunidades Virtuais de Aprendizagem. “Webquest”: Desafios Investigativos baseados na Internet (Web).

Preparação de uma apresentação multimídia.

SUMÁRIO

Módulo 1

Introdução a Java2SDK e BlueJ

Introdução

Este módulo tem como objetivo orientar sobre a instalação e utilização do ambiente de programação a ser usado no curso. Você irá realizar, neste módulo, as seguintes tarefas:

1. Instalar em seu computador o ambiente de programação Java 2 SDK (Software Development Kit).

2. Instalar o ambiente integrado e desenvolvimento de programas BlueJ IDE (Integrated Development Environment).

3. Compilar, depurar e executar um primeiro projeto Java "pronto".

4. Criar um novo projeto Java e escrever suas primeiras linhas de código de programa.

Antes de iniciar essas tarefas, entretanto, vamos procurar entender um pouco sobre computadores, programas, linguagens de programação.

Computadores, Programas e Linguagens de Programação.

Antes de começar a desenvolver programas, precisamos entender alguns conceitos fundamentais, tais como:

O que é um programa?

O que é uma linguagem de programação?

Como um programa é executado no computador?

O que é um programa?

Ao consultar um dicionário, você poderá encontrar mais de um significado para a palavra programa. Para nós, um programa significa, simplesmente:

Uma seqüência de instruções que um computador é capaz de executar, de maneira a coletar, manipular ou produzir dados.

Se você alguma vez já tentou cozinhar, seguindo as instruções de uma receita, certamente esse conceito de programa lhe parecerá familiar. Considere a seguinte receita simples (e um pouco imprecisa), para fazer um bolo:

Pegue alguns ovos.

Pegue um pouco de farinha.

Pegue um pouco e açúcar.

Pegue um pouco de manteiga.

Pegue um pouco de leite.

Pegue uma pitada de fermento.

Aqueça previamente o forno.

Misture a farinha e o fermento.

Bata a manteiga, o açúcar, os ovos e o leite.

Misture tudo.e leve ao forno por 20 minutos.

Imagine que você é o cozinheiro (ou cozinheira). Ao seguir a receita, você estará obtendo alguns dados (pegando os ingredientes), manipulando dados (misturando e levando ao forno) e produzindo uma saída (o bolo). Para isso, você segue um conjunto de instruções, em uma determinada ordem, ou seja, executa um programa. É claro que você é capaz de executar essa receita, ou programa, uma vez que você está familiarizado com seus ingredientes e com a linguagem em que estão escritas as instruções. Você acha que um computador seria capaz de executar esse programa? Você seria capaz de executá-lo, se as instruções estivessem escritas em chinês?

Linguagens de programação...

É claro que um computador não seria capaz de executar um programa tal como a nossa receita de bolo! Infelizmente, ele não saberia reconhecer os ingredientes, nem entende a linguagem na qual as instruções estão descritas.

Um computador é, de fato, uma máquina capaz de manipular dados muito simples – representados em linguagem binária, ou seja, como seqüências de zeros e uns. Além disso, ele é capaz de executar apenas um conjunto muito reduzido de instruções bastante simples (também representadas em linguagem binária), tais como somar dois números (em representação binária), comparar se dois números são iguais, ou desviar para a execução de uma outra instrução. Essa linguagem é usualmente chamada de linguagem de máquina. Se de fato é assim, como é possível que se venha utilizando computadores para realizar os mais diversos tipos de tarefas, muitas delas extremamente complexas?

A grande versatilidade de um computador vem do fato de que é possível combinar essas instruções simples, de maneira a descrever, passo a passo, como executar tarefas mais complexas – essa descrição é o que chamamos de programa.

Escrever programas na linguagem de máquina de um computador seria, entretanto, uma tarefa extremamente difícil e tediosa, além de muito sujeita a erros – um programa para executar mesmo uma tarefa bem simples consistiria de milhares de instruções. Gostaríamos, ao contrário, de poder escrever programas em linguagens de mais alto nível, nas quais pudéssemos expressar, de maneira mais simples e natural, as abstrações que fazemos usualmente, ao procurar entender e descrever fenômenos, tarefas e soluções de problemas do mundo real. Tais linguagens de alto nível é que são usualmente chamadas de linguagens de programação.

A linguagem Java, que utilizaremos neste curso, é uma linguagem de programação bastante moderna, que inclui diversos conceitos e recursos para facilitar a atividade de programação e torná-la mais produtiva.

Você provavelmente estará se perguntando: “Se programas são escritos em linguagens de programação de alto nível, muito diferentes da linguagem de máquina do computador, como esses programas podem ser executados?”

Compilando e executando programas.

Voltemos à nossa metáfora entre programas de computador e receitas de cozinha. Seguir uma receita escrita em uma linguagem que você não é capaz de entender parece ser mesmo impossível... Mas a solução é simples: bastaria contar com alguém capaz de traduzir as instruções da receita para a sua linguagem natural.

Também é assim, no caso de programas. Programas escritos em linguagens de programação de alto nível primeiramente têm que ser traduzidos para programas equivalentes em linguagem de máquina, antes de poderem ser executados pelo computador. Esse processo de tradução é chamado de compilação e o programa que faz essa tradução é chamado de compilador.

Você já deve ter percebido que desenvolver e executar um programa envolve várias etapas. Primeiro, é preciso escrever o programa, ou seja, descrever, passo a passo, como executar a tarefa desejada. Essa descrição deve ser feita em uma linguagem adequada – uma linguagem de programação. Então, temos que compilar o programa, isto é, traduzi-lo para uma linguagem que o computador seja capaz de interpretar. Só então a execução do programa pode ser iniciada.

Ambientes Integrados de Desenvolvimento de programas.

Para realizar as diversas etapas do desenvolvimento de programas, precisamos de algumas ferramentas: um programa editor de textos, para escrever o programa, um compilador, para traduzir o programa para a linguagem que o computador é capaz de interpretar... é útil também contar com ferramentas que auxiliem na depuração de erros em programas, na visualização e gerenciamento dos diversos arquivos de programa que compõem um projeto de programação... além de manuais eletrônicos "on-line" da linguagem de programação utilizada. Para facilitar a atividade de programação, essas ferramentas são usualmente integradas em um único programa – um Ambiente Integrado de Desenvolvimento de programas (ou IDE – Integrated Development Environment). Ao longo deste curso, vamos utilizar o IDE BlueJ para desenvolver nossos programas.

Agora que você já tem uma idéia sobre o processo de desenvolvimento de programas, vamos "colocar a mão na massa". Para começar a desenvolver programas, você primeiramente precisará instalar, em seu computador, o compilador e outros recursos da linguagem Java, e o ambiente integrado de programação BlueJ. As instruções para isso estão descritas nas seções a seguir.

Instalando o Java 2 SDK

Nesta seção vamos descrever como executar as seguintes tarefas:

1. Fazer o "download" do JDK 2 v1.5.
2. Instalar o JDK em seu computador.

Para executar essas tarefas, seu computador deverá estar conectado à Internet.

Fazendo o "download" do JDK (Java Development Kit)

Se você já está conectado à Internet, pode então fazer o download do JDK. Você pode

obter uma versão do JDK para Linux na página web de Java: http://java.sun.com/j2se/1.5.0/download.jsp. Se você tiver alguma dificuldade nesta tarefa, consulte o seu professor.

Salve o arquivo em algum diretório do seu computador, tal como, por exemplo, no diretório 'temp'.

Instalando o JDK em uma máquina Linux

Assim que terminar o "download", inicie a execução do arquivo que você acabou de salvar, abrindo um terminal de comandos na pasta onde o arquivo foi salvo e digitando. /jdk-1_5_0_01-linux-rpm.bin. Isto irá executar o instalador do J2SDK, como mostrado na figura a seguir:

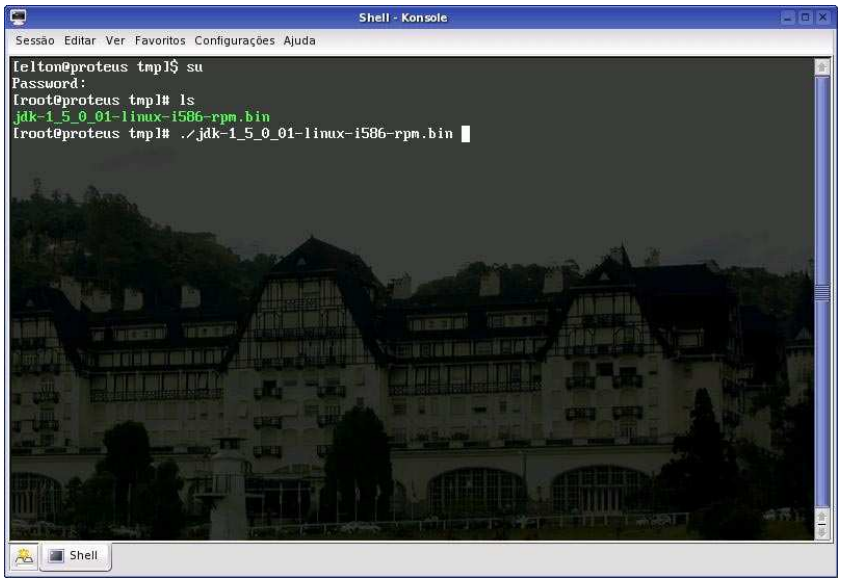


Figura 1 – Executando o instala

Observe, na figura, acima que mudamos de usuário para executar o instalador. É necessário que você seja o usuário root para instalar o J2SDK. Em seguida, será exibida uma mensagem contendo os termos para uso do software. Você deverá concordar com os termos para uso, respondendo "yes" (ou "y") à pergunta: "Do you agree to the above license terms? [yes or no]". O J2SDK será então instalado em seu computador, como ilustrado a seguir:

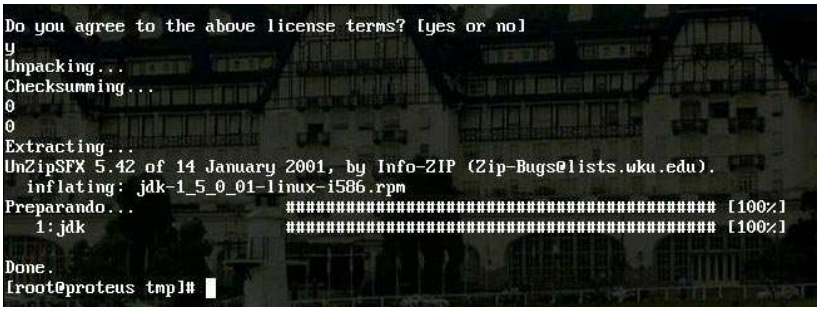


Figura 2 – Instalando o J2SDK

No Linux, assim como em outros sistemas operacionais, existe uma variável de ambiente, denominada PATH, que armazena os caminhos até os diretórios dos programas usados mais comumente. Para que você possa chamar o compilador Java a partir de qualquer diretório, o caminho até o JDK deve estar no PATH. Para testar se o PATH contém o diretório do JDK, abra um terminal e digite javac -version. O resultado deve ser algo parecido com:

```
javac 1.5.0_01
javac: no source files
Usage: javac <options> <source files>
```

Se você obtiver como resultado a mensagem "bash: javac: command not found", pode ser que o sistema ainda não conheça o caminho para o JDK. Você deverá então especificar esse caminho manualmente. Para fazer isso, você deverá saber onde o JDK foi instalado. Em geral, o diretório de instalação é: /usr/java/jdk1.5.0_01. Os compiladores e outras ferramentas estão na pasta bin. Sabendo isso, podemos especificar o caminho usando o comando export PATH=$PATH:/usr/java/jdk1.5.0_01/bin.
O problema é que isto só é válido para a sessão atual. Uma maneira mais permanente de se fazer isso é editar o arquivo profile, que está no diretório /etc (você deverá estar "logado" como um usuário que tenha permissão para fazer isso). Outra opção é modificar o seu perfil, no arquivo .bashrc (arquivo oculto). Em ambos os casos, a modificação é a mesma: adicione ao final arquivo o comando:

```
export PATH=$PATH:/usr/java/jdk1.5.0_01/bin
```

Agora reinicie a sua sessão e confirme que o compilador Java – javac - está funcionando.

Instalando a Documentação

É recomendável que você instale a documentação do J2SE™. Infelizmente, a Sun não permite a redistribuição desses arquivos e, por isso, você terá de fazer o download dos mesmos diretamente do site de Java, mencionado anteriormente. A página principal contém um link para a página do J2SE. Ao clicar nesse link, você será levado a uma página na qual poderá escolher download do J2SE ou da sua documentação. Em uma dessas duas páginas você terá a opção de baixar a documentação. Como a página da Sun muda freqüentemente, não é possível fornecer informações mais precisas. Se você precisar de ajuda para obter a documentação, consulte seu professor.

Agora que você já instalou o ambiente de programação de Java, vamos instalar o BlueJ.

Instalando o BlueJ IDE

BlueJ é o IDE (Integrated Development Environment), ou seja, Ambiente Integrado de Programação, que você irá utilizar neste curso, para desenvolver seus projetos Java. Você poderá fazer o download do BlueJ a partir do site http://www.bluej.org/.

Existem diversos outros ambientes integrados de programação para Java. Se você quiser saber um pouco sobre esses outros ambientes, veja as referências sobre Java incluídas no final deste tutorial. Vamos usar o BlueJ porque ele é simples e foi projetado para auxiliar programadores iniciantes a entenderem a linguagem Java.

Instalando o BlueJ em uma máquina Linux

Abra um prompt de comandos no diretório onde você salvou o arquivo. Digite o comando java -jar bluej-205.jar. Se o JDK estiver corretamente instalado, será exibida uma janela como a mostrada a seguir:

Figura 3 – Janela de instalação do BlueJ

No campo – Directory to install to: – você deve selecionar o diretório no qual você deseja instalar o BlueJ. Se você desejar fazer uma instalação na qual todos os usuários do sistema possam utilizar o BlueJ, é recomendável que você o instale no diretório / usr/local/bluej – é necessário ter permissões de administrador para fazer isso – , ou então você pode instalar em seu próprio perfil, como indicado na figura acima. Para escolher o diretório onde será instalado o BlueJ, clique no botão Browse (Navegar) para encontrar o diretório desejado. Isso fará aparecer uma nova janela de diálogo, como a mostrada a seguir:

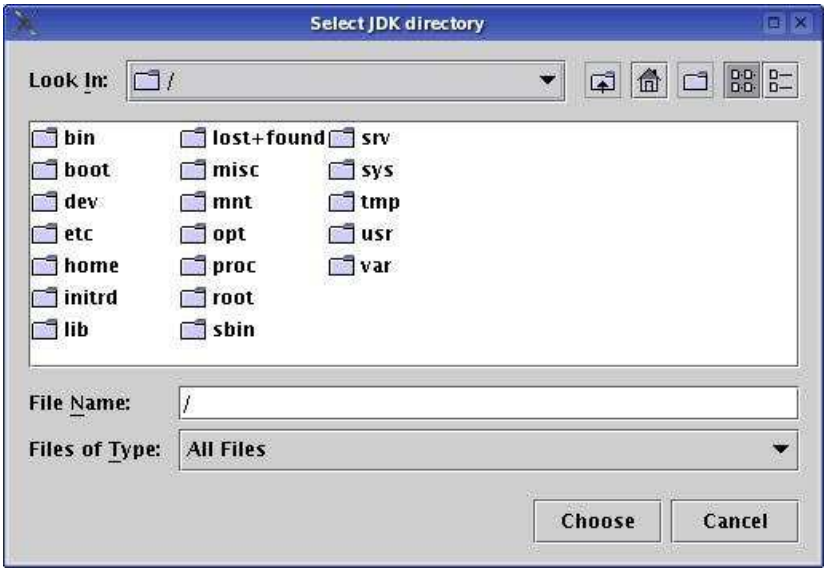


Figura 4 – Escolhendo o diretório de instalação do BlueJ

Utilize os botões do canto superior direito para navegar na estrutura de diretórios de seu computador. Clique em Choose (Escolher) para escolher o diretório onde você deseja instalar o BlueJ, ou clique no botão Cancel (Cancelar) para cancelar esta ação.

No campo – Java (JDK) directory: – você deverá selecionar o diretório onde está instalado o Java 2 SDK. O lugar onde o J2SDK está instalado depende da sua distri-

buição, mas em geral ele fica em /usr/Java, como pode ser visto na figura 5. Em seguida, clique em Install (Instalar), para instalar o BlueJ. Se o diretório de instalação do J2SDK não foi informado corretamente, aparecerá na tela uma janela como a mostrada a seguir:

Figura 5 – Mensagem de erro na especificação do diretório JDK

O texto exibido nessa janela pode ser traduzido assim: "O diretório Java que você especificou não é um diretório JDK válido. O diretório JDK é aquele no qual o Java 2 SDK foi instalado. Ele deve conter um subdiretório lib, o qual contém um arquivo tools.jar. Você poderá usar o botão Browse (Navegar), da caixa de diálogo Java (JDK) directory, para encontrar o diretório correto em que o JDK foi instalado. Depois disso, clique novamente no botão Install. Se você ainda tiver dificuldade, peça ajuda ao seu professor.Quando a instalação tiver sido concluída, aparecerá, ao lado do botão Install, um botão com o rótulo Done (Concluído). Clique neste botão para concluir a instalação do BlueJ.

Executando o BlueJ

Depois de instalar o BlueJ, existirá um arquivo bluej no diretório em que BlueJ foi instalado. Você pode executar o BlueJ com um clique sobre o arquivo, ou digitando ./ bluej em uma linha de comando.

Dependendo da configuração do seu sistema, você poderá ter que informar ao BlueJ qual é a máquina virtual Java que ele deverá usar. Se isso acontecer, apenas identifique a máquina correspondente ao js2sdk1.5.0_04.

Uma vez que tudo tenha sido instalado, vamos começar brincando com um projeto já existente, na próxima seção deste módulo.

Abrindo um projeto já existente

OK, você agora já tem tudo instalado: o ambiente de programação Java 2 SDK e o ambiente integrado de desenvolvimento de programas BlueJ. Podemos então iniciar nosso trabalho...

Nesta seção, vamos instalar um projeto Java já existente e mostrar como podemos compilar, depurar e executar programas Java, utilizando o ambiente BlueJ. Para isso, primeiramente iremos criar um diretório (ou pasta) no qual vamos armazenar esse projeto e os futuros projetos Java que você irá desenvolver. Em seguida, você irá fazer o download do projeto nesse novo diretório.

Comecemos então criando o diretório no qual vamos armazenar nosso projeto. Para

isso, inicie a execução do BlueJ e selecione a opção Project (Projeto) e, em seguida, selecione a opção New Project (Novo Projeto), tal como mostrado na janela a seguir.

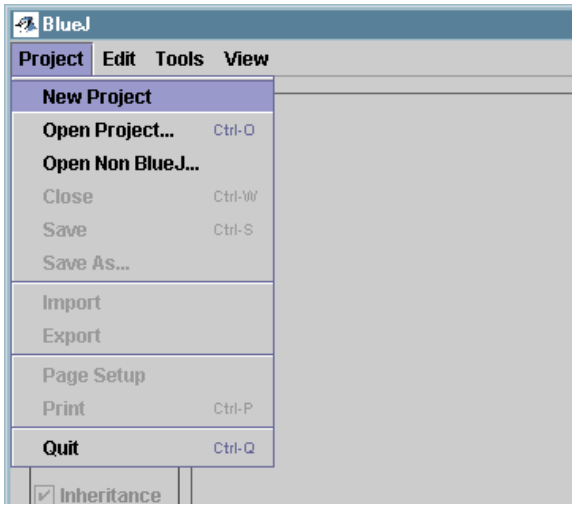


Figura 6 – Criando um novo projeto.

Quando você fizer isso, será exibida uma janela de diálogo, solicitando que você crie um novo projeto. Não é isso o que, de fato, desejamos fazer (faremos isso mais adiante, neste tutorial). Por enquanto, queremos apenas criar um novo diretório – portanto, clique no ícone New Folder (Nova Pasta), exibido no canto superior direito da janela:

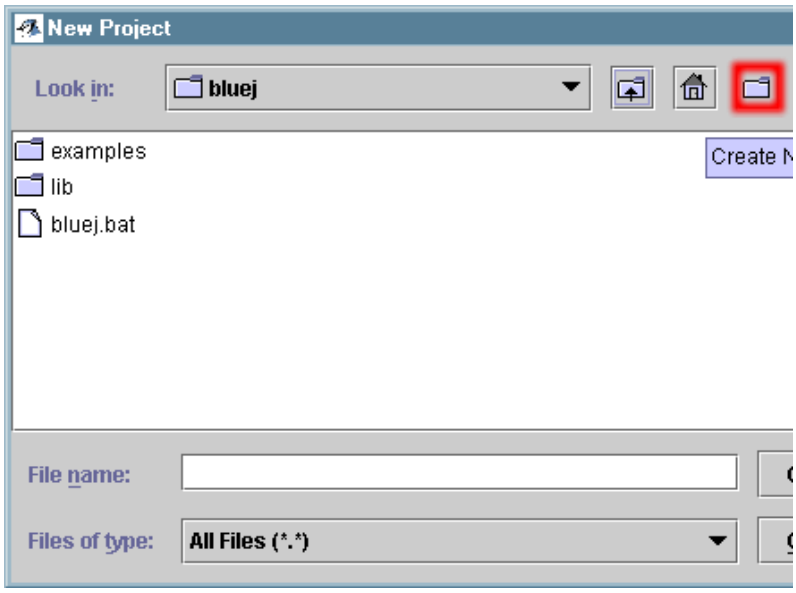


Figura 7 – Criando um diretório de projeto.

Quando você criar um novo diretório, um ícone correspondente será exibido na janela de diálogo, rotulado com o nome New Folder (Nova Pasta). Clique duas vezes, seguidamente, sobre esse rótulo, para poder então alterar o nome desse diretório, para o nome desejado:

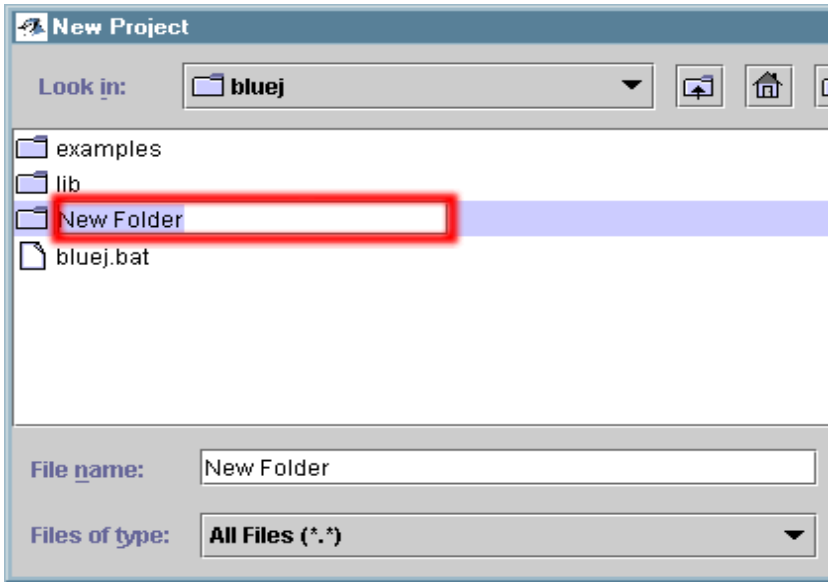


Figura 8 – Especificando o nome do seu diretório de projeto.

Modifique o nome desse diretório para "projetos" (sem as aspas) e aperte a tecla Return (ou Enter). Em seguida, clique sobre o botão Cancel (Cancelar), na janela de diálogo, pois queremos apenas criar um diretório de projetos, e não um novo projeto.
Depois de criar o diretório "projetos", você vai fazer o download do nosso primeiro exemplo de projeto Java, a partir do site

http://see.photogenesis.com.br/lucilia/projetos/Handball.zip.

É claro que, primeiramente, você deverá salvar esse arquivo, ou seja, selecionar a opção Save (Salvar). Use o seu gerenciador de arquivos para encontrar o diretório de projetos que você acabou de criar (esse diretório deverá estar localizado no diretório em que foi originalmente instalado o BlueJ) e salve o arquivo "handball.zip" neste diretório, como mostrado na figura a seguir.

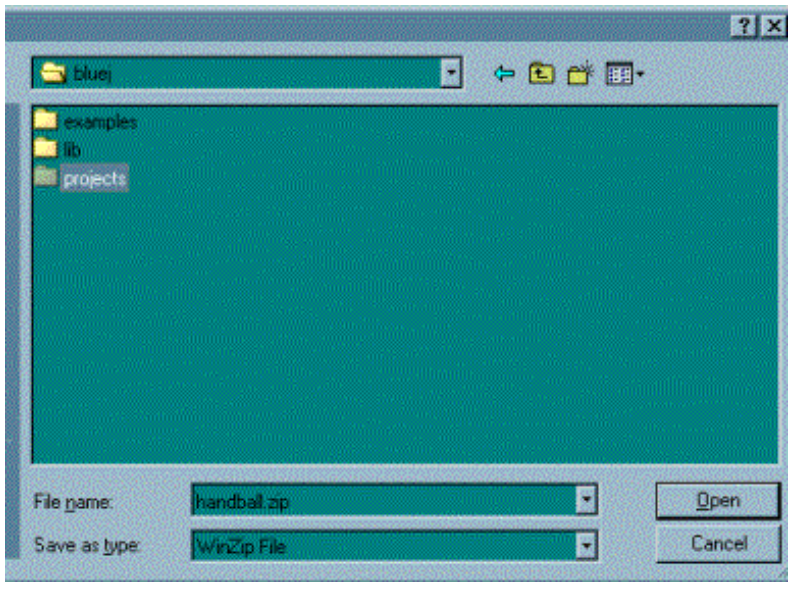


Figura 9 – Fazendo o download do arquivo "handball.zip"

O arquivo "handball.zip" é um arquivo compactado, que contém os arquivos do nosso projeto exemplo. Para descompactar esse arquivo, obtendo então o projeto original, clique sobre o mesmo e especifique o diretório "projetos" como o diretório no qual o

arquivo deve ser descompactado. Isso irá resultar na criação de um novo diretório – chamado Handball – como subdiretório de projetos. O diretório Handball deverá conter 11 arquivos e um subdiretório "docs". Se isso ocorrer, então você já terá tudo o que é necessário para compilar e executar seu primeiro projeto Java – um jogo semelhante ao jogo de ping-pong.

Retorne então ao BlueJ, selecione a opção Project , depois clique sobre a opção "Open Project"... (Abrir Projeto)... então, clique duas vezes, seguidamente, no diretório Handball:

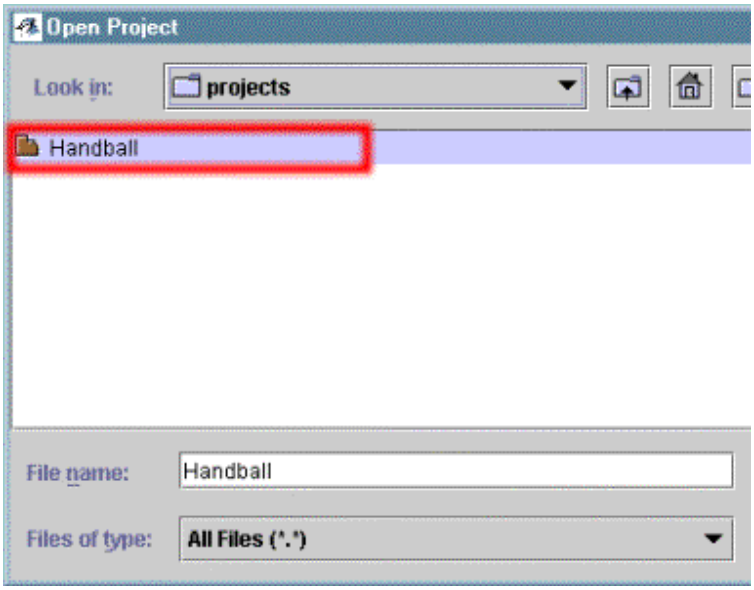


Figura 10 – Abrindo o projeto Handball.

O BlueJ então irá exibir o seu "project browser" (navegador de projeto), que mostra os relacionamentos entre os diferentes arquivos Java do projeto selecionado, como na figura a seguir:

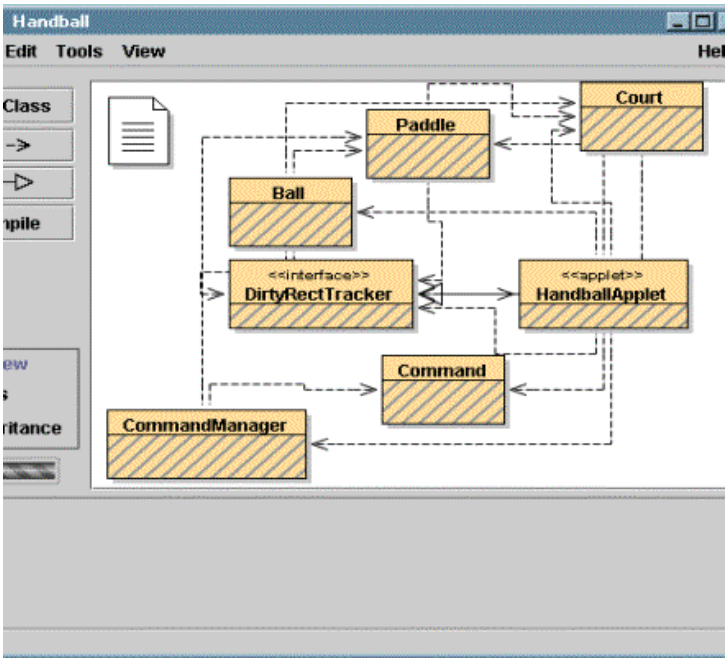


Figura 11 – O projeto Handball.

Agora você está pronto para compilar o projeto, o que será feito na próxima seção.

Compilando e executando um projeto

Nesta seção, você irá aprender como compilar um projeto e como eliminar alguns tipos de erros que podem ocorrer durante a compilação de programas. Você também irá aprender como executar um projeto que tenha sido compilado com sucesso – o que significa que você logo poderá jogar nosso jogo de ping-pong em seu computador. Sabemos que você já deve estar ansioso...

E então, como fazer para compilar um projeto? Se você for um bom observador, provavelmente terá notado a presença de um botão Compile (Compilar) na janela do navegador de projeto do BlueJ (repare no canto superior esquerdo da janela exibida na última figura da seção anterior). Não clique nesse botão ainda! Queremos que você observe algumas outras coisas antes disso.

Repare na janela do navegador de projeto do BlueJ. Você deverá ver umas caixinhas rotuladas com os nomes "Command", "Ball", etc. Essas caixinhas correspondem aos arquivos Java que constituem o projeto Handball. Você percebe as hachuras nessas caixinhas? Essas marcas indicam que o nosso projeto deve ser compilado e montado (em inglês, built). Sempre que existirem marcas desse tipo em algum dos arquivos de um projeto, temos que compilar o projeto (ou pelo menos o arquivo marcado), antes de executá-lo.

OK. Siga em frente e compile o projeto Handball, clicando sobre o botão Compile. Veja o que acontece.

Você deverá ter percebido que o compilador monta, um a um, os arquivos do projeto, isto é, você deverá ter visto desaparecerem as hachuras nas caixinhas que representam cada um desses arquivos. Enquanto o processo de compilação está em andamento, aparece no canto inferior da janela a mensagem "Compiling" (Compilando). Quando a compilação é concluída sem erros, aparece, nesse mesmo local, a mensagem "Compiling... Done" (Compilando... Concluído).

Agora que o projeto foi compilado, você pode ir em frente e executá-lo. Para fazer isso, clique com o botão direito do mouse sobre a caixa HandballApplet e selecione a opção Run Applet (Execute Applet) do menu que é mostrado na tela. Isso faz com que seja exibida a seguinte janela de diálogo:

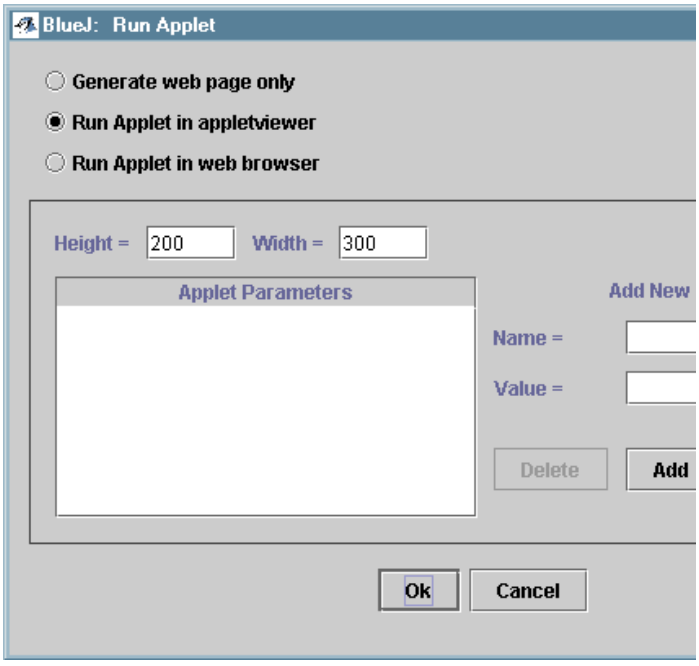


Figura 12 – Executando um applet no BlueJ.

Como os parâmetros já estão definidos corretamente, basta clicar no botão Ok para iniciar a execução de "Handball" pelo Applet Viewer:

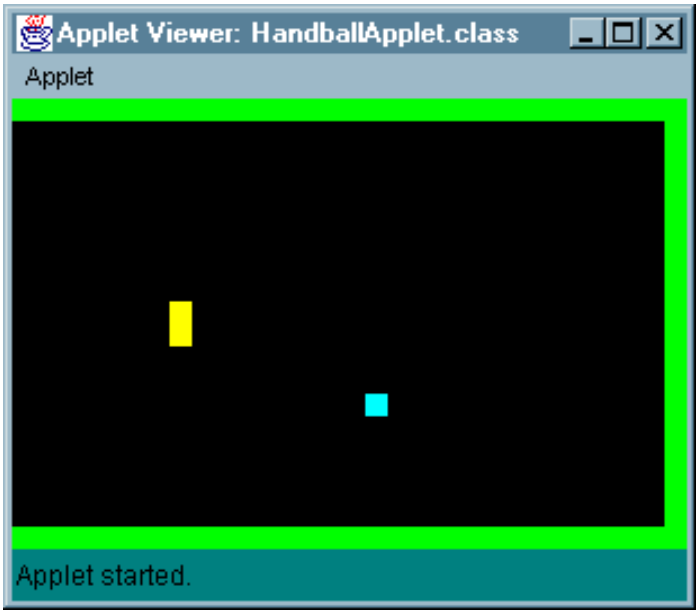


Figura 13 – O applet "Handball".

Para jogar, use os botões ? e ? do seu teclado, para mover a raquete amarela para cima e para baixo na tela do applet (poderá ser necessário clicar sobre a janela do applet para que ele comece a responder às teclas que você pressiona no teclado).

Agora que já conseguimos fazer com que o programa seja executado, vamos terminá-lo! Para isso, clique sobre o X que aparece no canto superior direito da janela do applet, fechando a janela e terminando o programa.

Agora vamos ver o que acontece quando ocorre um erro durante a compilação de um programa, para que você tenha uma idéia inicial sobre o processo de depuração de erros de compilação em programas.

Retorne então à janela do navegador de projeto do BlueJ e clique duas vezes, seguidamente, sobre a caixa que representa o arquivo "Ball" do projeto Handball. Isso fará com que seja exibida a janela de edição de programas, contendo o arquivo de código que implementa a bolinha do jogo. Uma visão parcial da janela exibida é mostrada a seguir:

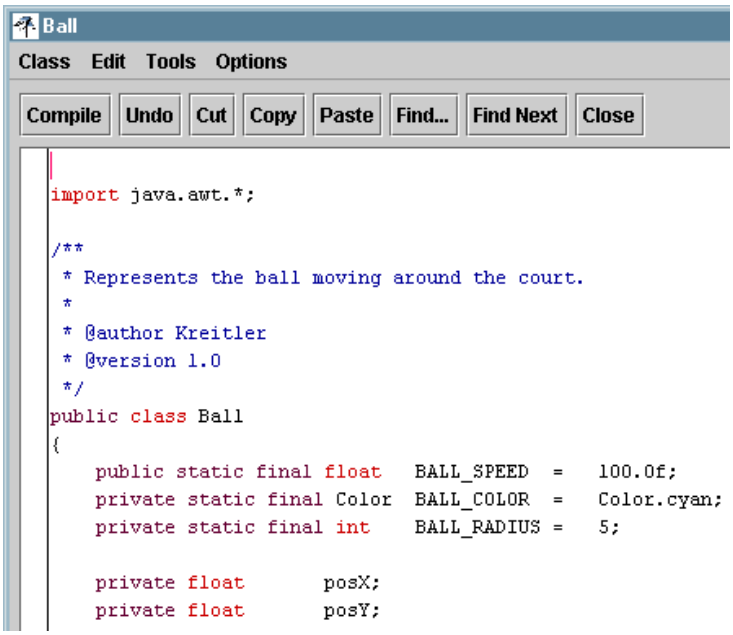


Figura 14 – Editando um arquivo de programa no BlueJ.

Vamos adicionar um texto ao programa, aleatoriamente, e compilar o programa novamente. Por exemplo, você poderá escrever "Chega de blah blah blah!" depois de "posX;":

```
private float      posX; No more blah blah blah!
private float      posY;
private float      velX;
private float      velY;
private Image      backBuffer;
```

Figura 15 – Introduzindo um erro.

Agora, pressione o botão "Compile" (Compilar), que aparece no canto superior esquerdo da janela de edição de programa.Você então verá aparecer um ponto de interrogação no canto inferior direito da janela do editor:

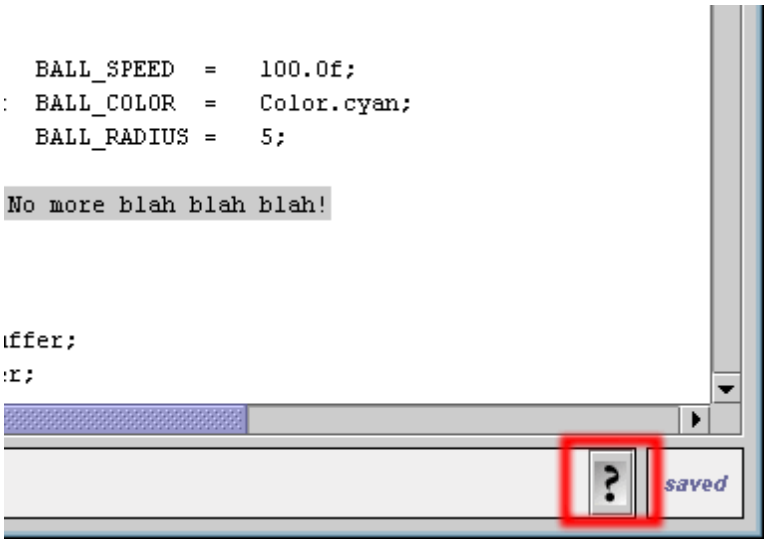


Figura 16 – Obtendo a descrição de erros de compilação, no BlueJ.

À esquerda desse ponto de interrogação deverá aparecer uma breve descrição do erro de compilação ocorrido. No do programas modificado como sugerido acima, aparecerá a mensagem "';' expected." Você poderá saber mais detalhes sobre o erro indicado, clicando sobre o ponto de interrogação.

Volte agora à janela principal do projeto. Note que a caixa correspondente ao arquivo Ball tem hachuras com linhas de cor cinza. Isso também ocorre com a caixa correspondente ao arquivo HandballApplet. Porque o arquivo HandballApplet estaria "quebrado" se apenas alteramos o conteúdo do arquivo Ball?

A resposta é que existe aí uma "dependência": HandballApplet faz referência a Ball (ou depende de Ball), como mostram as linhas pontilhadas do diagrama representativo do projeto HandBall, exibido pelo navegador de projeto do BlueJ:

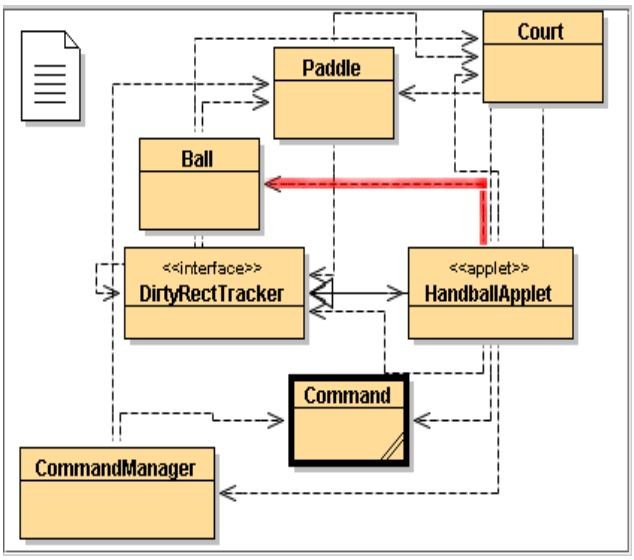


Figura 17 – Visualizando dependências entre classes no BlueJ.

OK. Já chega de código de programa, por enquanto! Vamos desfazer as alterações feitas no arquivo Ball, para então compilar e executar o applet mais uma vez, certificando-nos de que tudo está funcionando corretamente, como antes. Voltaremos a fazer referência a esse projeto posteriormente, neste tutorial.
Só mais uma observação final, para aqueles que desejarem explorar (e brincar com) o código desse programa (afinal, ele foi feito para isso mesmo). Em um programa Java, qualquer parte de uma linha de texto que é precedida por um par de barras inclinadas (//) é considerada um comentário. Também é considerado comentário um bloco de texto iniciado por /* e terminado por */ , como mostrado abaixo:

```
// Essa linha de texto é um comentário.

/**
 * Esse bloco de texto é um comentário.
 */
```

O compilador Java ignora comentários, ou seja, comentários não fazem parte do código do programa. Eles são usados para tornar o código do programa mais claro. Experimente ler apenas os comentários do texto do arquivo Ball. Isso irá ajudá-lo a entender o que acontece quando o código é executado.

Aqui vão algumas dicas para brincar um pouco com o código: troque a "frase" Color.green para Color.blue ou para Color.red. Salve o arquivo e então compile e execute o programa novamente e veja o que acontece. Você pode também procurar, no código do programa, toda ocorrência de "fillRect " e experimentar trocá-la por "drawRect " (por exemplo, fillRect(a, b, c, d) seria modificado para drawRect(a, b, c, d) – não altere o que está entre parênteses).

Não se preocupe se você danificar os arquivos do projeto – você sempre poderá extrair novamente os arquivos originais, a partir do arquivo Handball.zip.

Chega então de trabalhar com um projeto já pronto. No módulo a seguir, você vai criar um novo projeto, desde o início. Antes, porém, vamos rever o que já aprendemos.

Conclusão

Você acaba de concluir o primeiro módulo deste tutorial – isso já é um bocado! Você não apenas instalou o ambiente de programação Java e o ambiente integrado de desenvolvimento de programas BlueJ, mas também aprendeu a compilar e executar programas Java.

Nos módulos seguintes, você ira'começar criando seu primeiro projeto de programação e irá, pouco a pouco, aprendendo a escrever programas em Java. Vamos sempre trabalhar com dois tipos de tarefas – escrever pequenos programas, a partir do zero, ou modificar e adicionar código a um projeto maior, já existente.

Esperamos que você tenha gostado, até agora. Nos veremos de novo no próximo módulo.

Módulo 2

Conceitos Básicos da Linguagem Java

Introdução

Bem vindo ao segundo módulo do nosso tutorial. Imaginamos que você já deve estar ansioso para criar o seu primeiro projeto de programação. É isso o que vamos fazer neste módulo. Você vai saber um pouco sobre os diferentes tipos de projeto que você pode criar em Java e criar projetos de cada um desses tipos. Também vai aprender um pouco sobre alguns conceitos básicos de linguagens de programação orientadas a objetos (POO): classes, objetos, métodos...

Parece bastante coisa, não é? Vamos dividir isso em alguns tópicos:

1. Java básico - conceitos básicos de POO: classes, objetos e métodos.
2. Criando um primeiro projeto de programação – um applet
3. Segundo projeto de programação – um console simples.
4. Um pouco mais sobre Java – declarações de classes, variáveis e métodos.
5. Incrementando nosso segundo projeto de programação – escrevendo no console.

Vamos começar aprendendo um pouco sobre Java.

Introdução a classes, objetos, métodos...

Antes de criar um projeto e começar a escrever código de programa, precisamos entender alguns conceitos fundamentais de programação. Como Java é uma "linguagem de programação orientada a objetos" (LPOO), vamos começar procurando aprender um pouco sobre conceitos existentes em tais linguagens, como classes, objetos, métodos... Em seguida, será útil saber também sobre os diferentes tipos de programas que podemos criar em um ambiente de programação em linguagem Java.

Objetos, classes, métodos, interfaces...

No paradigma de programação orientado a objetos, a construção de um programa para implementar um determinado sistema baseia-se em uma correspondência natural e intuitiva entre esse sistema e a simulação do comportamento do mesmo: a cada entidade do sistema corresponde, durante a execução do programa, um objeto. Voltemos à nossa metáfora entre programas e receitas de culinária. No nosso exemplo, o cozinheiro usa ingredientes, como ovos e leite para criar o produto final – o bolo. Ele também usa ferramentas, tais como vasilhames, batedeira e forno. Podemos então nos perguntar: "Quais seriam as entidades correspondentes, quando um computador executa um programa Java?"

A resposta é objetos. Em outras palavras, em um programa em linguagem orientada a objetos, todos os dados, ferramentas usadas para manipular dados, ou qualquer informação usada no programa será representada como um objeto (NOTA: existem algumas exceções, que examinaremos mais adiante). Note que objetos do mundo real em geral possuem duas características: estado e comportamento. Por exemplo, uma bicicleta

pode estar parada, ou em movimento, ter uma determinada cor e marca, etc. Ele pode "executar" ações como acelerar e desacelerar, mudar de direção etc. Objetos em Java não são muito diferentes – são componentes da execução de programas que modelam objetos do mundo real, no sentido de que também possuem um estado e comportamento.

Um objeto de um programa mantém informação sobre seu estado por meio de variáveis e implementa seu comportamento (ou as ações que o objeto é capaz de executar) por meio de métodos. O diagrama a seguir é uma representação conceitual de um objeto, constituído de variáveis e métodos:

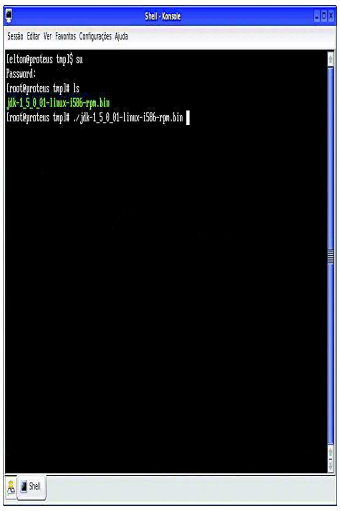


Figura 1 – Um objeto genérico.

Lembra-se do nosso programa Handball, do módulo anterior? A bola e a raquete, nesse programa, são exemplos de objetos. Também são objetos outros elementos de interfaces gráficas de programas, como janelas, botões e menus. Ao escrever um programa Java para implementar um determinado sistema do mundo real, estaremos, portanto, escrevendo código para descrever os objetos desse sistema, assim como comandos cuja execução resulta em criar esses objetos e fazê-los interagir de maneira a simular o comportamento do sistema.

Um componente de programa que descreve a estrutura e o comportamento de um grupo de objetos similares é chamado de uma classe. Em outras palavras, uma classe descreve informações que caracterizam o estado de objetos dessa classe, assim como as ações (ou operações) que eles podem realizar. Os objetos de uma classe – também chamados de instâncias da classe – são criados durante a execução do programa.
O exemplo abaixo ilustra um trecho do código da classe que descreve a bola do nosso programa Handball, do módulo anterior:

```
public class Ball {

    ... o código que descreve o comportamento da bola entraria aqui...
}
```

Não se preocupe, por enquanto, em entender o significado de 'public' e dos colchetes, no exemplo acima. O importante é que "class Ball" informa ao compilador Java que essa é a descrição de um tipo específico de objetos, que denominamos 'Ball' (bola, em inglês). Vamos examinar um pouco mais o código da classe 'Ball'?

```
public class Ball {
```

```
    ...
    private float      posX;
    private float      posY;
    ...

    // O método 'bounce'.
    public void bounce() {
        .... o código que descreve a maneira como a bola se movimenta viria aqui...
    }
}
```

A classe Ball contém variáveis posX e posY, que constituem parte da descrição do estado da bola – essas variáveis representam a posição da bola na tela, representada coordenadas cartesianas. A classe contém, também, um método – chamado 'bounce' – que contém o código que descreve o movimento da bola na janela do programa, ou seja, este método constitui parte da descrição do comportamento de objetos da classe Ball.

Não se preocupe, por enquanto, com outros detalhes do código desta classe que você ainda não compreende. Voltaremos a esses detalhes mais adiante, depois de criarmos nosso primeiro projeto. Antes disso, vale à pena saber um pouco sobre os diferentes tipos e programas que podemos criar na linguagem Java.

Aplicações console, aplicações gráficas "stand-alone" e Applets

Para executar um programa Java, o computador precisará ter alguma informação sobre a estrutura do programa. Mais especificamente, ele deverá conhecer os objetos que compõem o programa e o ponto do programa onde deve ser iniciada a execução das instruções. Com relação a esses aspectos, existem essencialmente três tipos de programas, ou aplicações, em Java: aplicações "console", aplicações gráficas "stand-alone" e "applets". Os dois primeiros tipos possuem várias características comuns e constituem a maioria dos programas exemplo que veremos neste tutorial. Entretanto, escolhemos trabalhar com applets, inicialmente, porque esse tipo de programa é mais fácil de ser criado no ambiente BlueJ.

Programas Java são, em geral, executados diretamente no computador onde estão instalados. Tais programas devem conter um objeto descrito por uma classe que contenha um método particular, chamado main. Quando programas desse tipo são executados, o usuário deve informar explicitamente qual é a classe que contém o método main – a execução do programa é iniciada a partir da primeira instrução desse método.

Programas desse tipo podem ser aplicações console ou aplicações gráficas – que oferecem uma interface gráfica (GUI – Graphical User Interface) para interação com o usuário, ou seja, utilizam, janelas, menus, botões, etc., para ler dados digitados pelo usuário e exibir informações na tela. Aplicações console, ao contrário, em geral apenas lêem e exibem dados, na forma de texto, em uma janela exibida na tela do computador, que denominamos console Java.

O terceiro tipo de programas Java é chamado de um applet. Falando de maneira simples, um applet é uma aplicação Java projetada para ser executada a partir de um navegador web (ou de um AppletViewer – um programa específico para visualização de applets). Muitos dos programas para jogos disponíveis na Internet são applets Java (veja, por exemplo, em http://www.cut-the-knot.org/nim_st.shtml).

Existem muitos detalhes importantes a respeito de applets — são programas que têm um comportamento especial, de maneira a garantir que sua execução possa ser feita de maneira segura através da Internet, sem permitir que possa danificar arquivos e programas do computador no qual é executado. Entretanto, não vamos nos preocupar com esses detalhes, nesse momento. Por enquanto, precisamos apenas saber que uma aplicação desse tipo constitui um objeto de uma determinada classe – denominada 'Applet'. O usuário não precisa especificar para o computador onde deve ser iniciada a execução de um applet – o navegador web é que irá iniciar a execução de diferentes métodos definidos na classe Applet, em resposta a eventos que podem ocorrer em janelas, tais como clicar o mouse sobre um botão, editar um texto em uma janela, fechar uma janela etc.

Agora que já aprendemos um pouco sobre Java, vamos praticar mais um pouco, criando nosso primeiro projeto de programação, na seção a seguir.

Criando um novo Projeto

Agora que você já trabalhou um pouco com o ambiente integrado de desenvolvimento de programas BlueJ e já aprendeu um pouco sobre a linguagem Java, podemos experimentar criar um primeiro projeto - um programa típico, que simplesmente imprime na tela uma mensagem: "João comeu pão".

Criando um Projeto

Primeiramente, temos que criar um novo projeto. Um projeto é simplesmente um diretório (ou pasta), no qual serão armazenados os arquivos java que compõem o programa que vamos desenvolver. Para criar um novo projeto, inicie a execução do BlueJ e vá para o menu Project (Projeto).

Figura 2 – Criando um novo projeto.

Selecione a opção New Project (Novo Projeto) e então aparecerá uma janela de diálogo, que mostra a estrutura de arquivos do seu computador. Procure o diretório projects (criado na seção anterior).

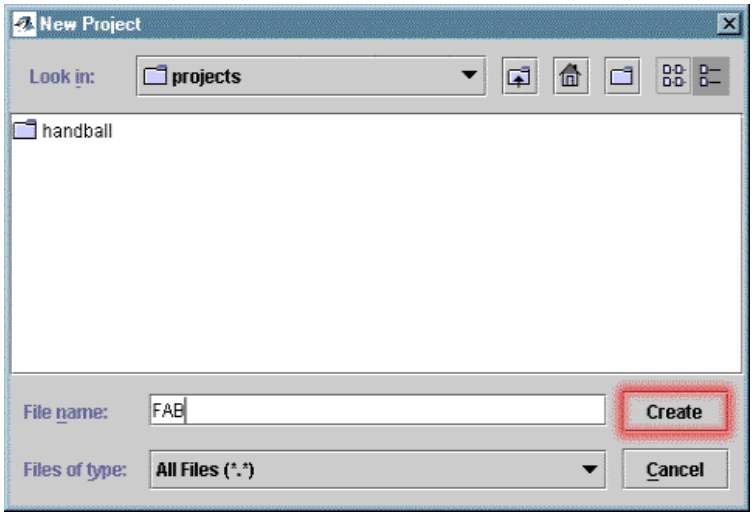


Figura 3 – Criando o diretório FAB.

Chame o novo diretório a ser criado de FAB (ou outro nome que você preferir). Clique então no botão botrr outro nome que voc vocquivos do seu computador. to. Create (Criar), e o BlueJ irá criar um novo diretório, com o nome especificado, e exibir uma nova janela:

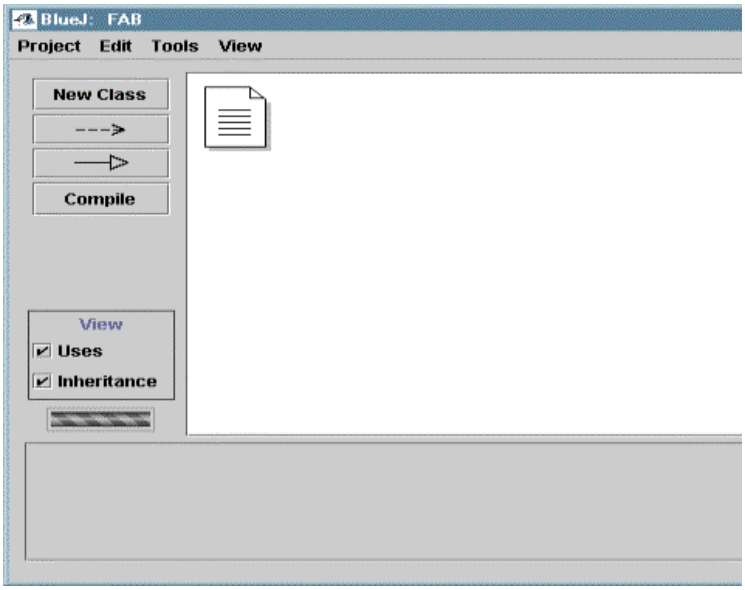


Figura 4 – Um novo projeto, ainda vazio.

Nesse ponto, pode ser instrutivo que você examine a estrutura de diretórios do seu computador, usando o gerenciador de arquivos do sistema, para verificar que o arquivo realmente foi criado. Você vê, no diretório FAB, um arquivo com o nome iretgerenciador de arquivos do sistema, para verificar que o arquivo realmente foi criado. README.TXT file? Esse arquivo é representado, na janela do BlueJ, pelo ícone que aparece no canto superior esquerdo. O arquivo README.TXT é simplesmente um arquivo de texto, que é criado para cada projeto pelo BlueJ, e serve para armazenar informações sobre o projeto, tais como: o nome, o objetivo, a data e os nomes dos autores do projeto, e instruções para os usuários do programa. Você não precisa se preocupar em preencher essas informações para os projetos que vamos desenvolver ao longo do curso.

Adicionando código de programa ao Projeto

Clique sobre o botão New Class (Nova Classe). Isso fará aparecer uma janela de diálogo,

solicitando informar o tipo de classe que se deseja criar:

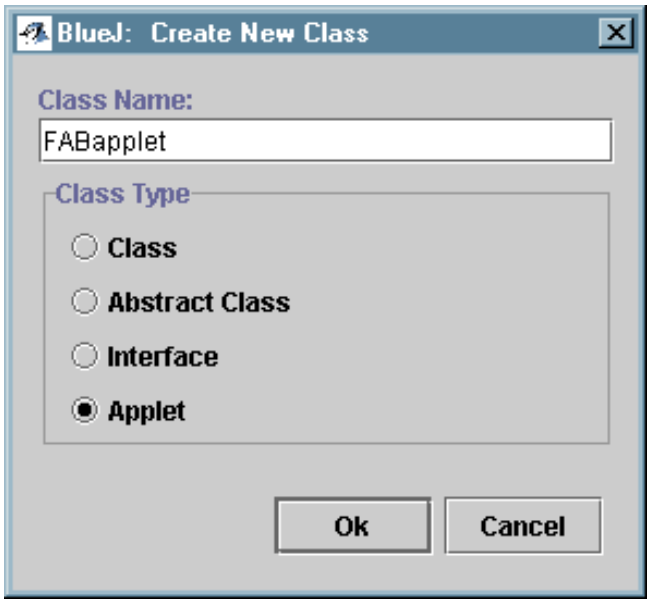


Figura 5 – Criando uma nova classe.

Selecione Applet e especifique FABapplet como o nome da classe a ser criada.

Nesse ponto, você provavelmente vai reclamar: "Opa, espere aí! Eu quero aprender a programar em Java, e não apenas seguir cegamente instruções!" Você está coberto de razão – o que temos feito até agora parece não ser muito educativo. Mas espere um pouco mais e vamos esclarecer tudo. Por enquanto, estamos apenas criando a "casca" do nosso programa. Uma vez que tenhamos terminado, vamos descrever o que é uma classe, um applet... e aprender alguns comandos básicos da linguagem Java.

Quando você clicar Ok na janela de diálogo mostrada anteriormente, o BlueJ irá gerar, automaticamente, o código para um applet básico, como conteúdo de um arquivo chamado "FABapplet.java". Além disso, um ícone correspondente a esse arquivo será mostrado na janela do nosso projeto FAB:(veja a figura a seguir).

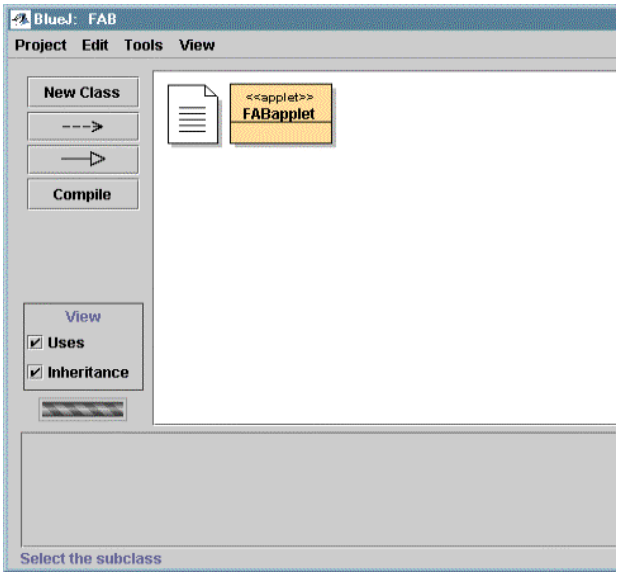


Figura 6 – O applet FABapplet do projeto FAB.

Para ver o código gerado pelo BlueJ, clique duas vezes, seguidamente, sobre o ícone de FABapplet.. Isso fará ser exibida uma janela contendo o código – a maior parte do qual consiste de comentários.

Vamos, agora, primeiramente, executar o programa. Isso é feito da mesma maneira como fizemos na seção anterior. Primeiro, compile o programa, usando o botão "Compile" (Compilar), depois clique sobre o ícone de FABapplet (com o botão direito do mouse) e selecione a opção Run Applet (Executar Applet). Aparecerá então a janela de diálogo de execução de applets. Certifique-se de que está selecionada, nesta janela, a opção Run in Applet Viewer e clique Ok.

Quando você fizer isso, aparecerá a janela do AppletViewer, contendo o seu novo programa. Você deverá ver algo como a janela mostrada a seguir, no canto superior esquerdo da tela do computador:

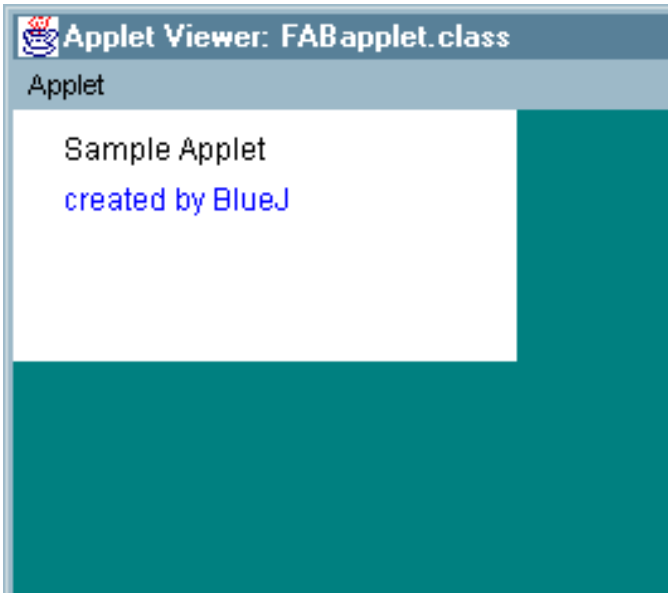


Figura 7 – Executando o FABapplet.

Finalmente – algum código!

Depois de tudo isso, vamos finalmente escrever algum código de programa. Você provavelmente gastaria horas apenas para escrever, em Java, o código e programa criado pelo BlueJ automaticamente. Por enquanto, não vamos nos preocupar em entender esse código. Vamos primeiramente focalizar alguns conceitos que permitirão a você ser capaz de modificar esse código para escrever na janela do applet a mensagem "João comeu pão". Você deverá estar se perguntando:

1. Como um Applet é capaz de exibir informação na tela do meu computador?

2. Como é o código que devo escrever no programa, para que a informação que eu desejo seja exibida na janela do Applet?

Vamos tratar disso passo a passo.

O método Paint –ou "Como um applet escreve uma mensagem em sua na janela"

Voltemos ao programa que acabamos de criar – FABapplet. Sabemos que um applet é um objeto. Isso significa que nosso programa deve conter uma definição de classe correspondente a esse objeto. Além disso, essa definição de classe deve conter métodos que descrevem as ações que um applet deve ser capaz de realizar. Sabemos que um applet deve ser capaz de exibir mensagens na sua janela. Como exibir uma mensagem é uma ação, deve existir um método que descreve como isso é feito. Se pudermos encon-

trar esse método, poderemos usá-lo, em nosso programa, para exibir na tela a mensagem "João comeu pão". E assim terminaremos nosso primeiro programa!

Bem, o método que procuramos é chamado paint. Todo applet usa seu método paint para exibir mensagens na tela, ou "pintar" a tela, como se diz mais comumente.

Abra o código de FABapplet no BlueJ, clicando duas vezes, seguidamente, no arquivo FABapplet . Clique então no botão Find... (Localizar) do editor de programas do BlueJ, como mostrado na figura a seguir:

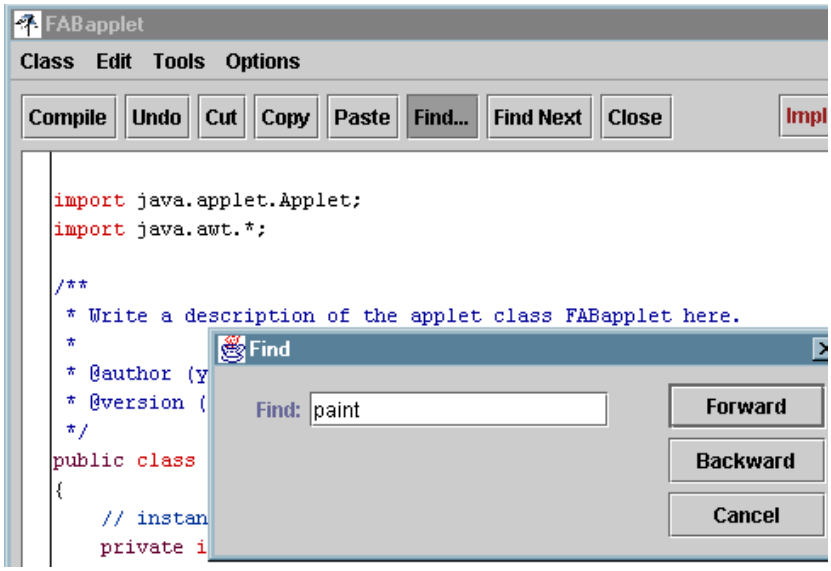


Figura 8 – Usando 'Find' no BlueJ.

Clique no botão forward (para frente). Repita esse processo (Find...forward), até encontrar a seguinte linha de código:

```
public void paint(Graphics g)
```

O método paint (pinte) é usado por um objeto da classe Applet para "pintar" a sua janela, isto é, para desenhar ou escrever mensagens na janela. Vamos então alterar o código deste método, de maneira que sua execução resulte em exibir a mensagem desejada.

Exibindo uma mensagem na janela do applet

Agora que já sabemos onde escrever nosso código, precisamos compreender como descrever a ação que desejamos que seja executada. Primeiramente, vamos dar uma olhada no código já existente, que foi gerado automaticamente pelo BlueJ:

```
/**
* This may be the most important method in your applet: Here, the
* drawing of the applet gets done. "paint" gets called everytime the
* applet should be drawn on the screen. So put the code here that
* shows the applet.
*
* @param g the Graphics object for this applet
*/
public void paint(Graphics g)
{
    // simple text displayed on applet
    g.setColor(Color.white);
```

```
    g.fillRect(0, 0, 200, 100);
     g.setColor(Color.black);
    g.drawString("Sample Applet", 20, 20);
     g.setColor(Color.blue);
    g.drawString("created by BlueJ", 20, 40);
}
```

Note que o código parece ter duas seções, destacadas, acima, em verde e vermelho. A primeira seção é apenas um comentário, escrito em inglês. Você deve estar lembrado – comentários são utilizados apenas para tornar o código do programa mais claro, sendo totalmente ignorados pelo computador. O comentário acima poderia ser traduzido do seguinte modo: "Esse pode ser o método mais importante do seu applet: ele é usado pelo applet para desenhar e escrever na tela. O método "paint" é executado sempre que a janela do applet deve ser exibida (ou "pintada") na tela. Portanto, devemos escrever, no o corpo desse método, o código correspondente ao que desejamos que seja exibido na janela do applet.". E a segunda seção do código? Você deve ter reparado, quando executamos nosso applet, que as seguintes mensagens foram exibidas na sua janela: "Sample Applet" ("Applet exemplo") e "created by BlueJ" ("criado pelo BlueJ"). Repare que, no corpo do método paint, existem duas linhas de código correspondentes a essas mensagens:

```
... ...
 g.drawString("Sample Applet", 20, 20);
... ...
 g.drawString("created by BlueJ", 20, 40);
```

Você já deve ter adivinhado que são estas as duas linhas de código cuja execução resulta na exibição das mensagens. Mas qual é exatamente, o significado do código acima? Vamos por partes...

```
public void paint(Graphics g)
```

Por enquanto, vamos ignorar as palavras public e void – o significado de cada uma será explicado mais adiante. Você já sabe que paint é o nome do método. Mas e esse misterioso "(Graphics g)"? O que isso significa?

Lembre-se que um applet é um programa que é executado a partir de uma página de um navegador web (ou por um AppletViewer). Suponha que a janela do navegador seja redimensionada, ocultando ou exibindo parte da janela do applet. Para garantir essa janela seja exibida corretamente, o navegador web realiza uma chamada do método paint do applet – isto é, inicia a execução do código desse método. É como se o navegador dissesse ao applet: "Ei, pinte sua janela!".
Entretanto, existem várias informações sobre a configuração do computador que o applet precisa conhecer, para pintar sua janela corretamente: o tamanho da tela do computador (640x480? 800x600? ou algum valor diferente?), o número de cores disponíveis (256? 16 mil? 24 milhões?) e várias outras informações dessa natureza. Portanto, o navegador de fato estaria dizendo: "Ei, pinte sua janela em uma tela de 800x600, com 16 mil cores, sendo vermelho a cor de fonte (letra) corrente." Esse misterioso g é que provê ao método paint a descrição da configuração corrente de características gráficas da máquina. A palavra Graphics indica que g contém informação sobre esse contexto gráfico (g é um objeto da classe Graphics). Mais formalmente, dizemos que o método paint tem como parâmetro um objeto g, da classe Graphics, o qual provê a esse método

a descrição de características gráficas do computador. Resumindo:

paint... — "Ei, Applet—pinte sua janela..."

(Graphics ... — "...usando a informação sobre a características gráficas da máquina ..."

...g) "...que são providas pelo objeto g."

Vamos agora rever o os comandos "drawString":

```
...   ...
 g.drawString("Sample Applet", 20, 20);
...   ...
 g.drawString("created by BlueJ", 20, 40);
```

A expressão g.drawString("Sample Applet", 20, 20) é uma chamada ao método drawString, dirigida ao objeto g. "Ei, espere aí!" – estou ouvindo você dizer. "Mas o navegador web não acabou de chamar o método paint do applet, para indicar que ele deve pintar sua janela?" Bem, é isso mesmo. A execução de uma determinada ação, por um dado objeto, envolve, muitas vezes, solicitar, a outros objetos, que executem uma ação – o paradigma de orientação por objetos é baseado na simulação de um sistema por meio da interação entre os objetos desse sistema, você se lembra? Pois então. No corpo do método paint, é feita uma chamada a um método do contexto gráfico – representado pelo objeto g, que é passado como parâmetro para esse método – solicitando a esse objeto que exiba a mensagem "Sample Applet".
Essa idéia de interação entre os objetos por meio de chamada de método é muito importante. Por isso, vamos ver mais um exemplo. Você se lembra do código da classe Ball? Se quisermos que um objeto dessa classe – chamado, digamos, ball – se movimente na tela, devemos escrever o seguinte:

```
ball.bounce();
```

Nessa chamada de método, o objeto ball é dito objeto alvo da chamada. O método chamado é o método bounce, definido na classe Ball. Nesse caso, o método chamado não possui parâmetros e, por isso, o nome do método é seguido pelos símbolos '( )', representando uma lista de parâmetros vazia.

Voltemos então aos mistérios de drawString. Qual é o significado dos números 20, 20 e 20, 40, que aparecem acima, nas chamadas a esse método?

Isso é fácil – eles definem a localização onde devem ser escritas as mensagens. Se você olhar bem para a imagem que aparece na tela do seu computador, verá que ela é constituída de pequenos pontos, que chamamos de pixels. Os números passados como argumentos para o método drawstring indicam quantos pixels devemos mover para frente e para baixo, antes de encontrar a posição onde deve ser escrita a mensagem, como mostra a figura a seguir. A posição inicial é o canto superior esquerdo da janela do applet.

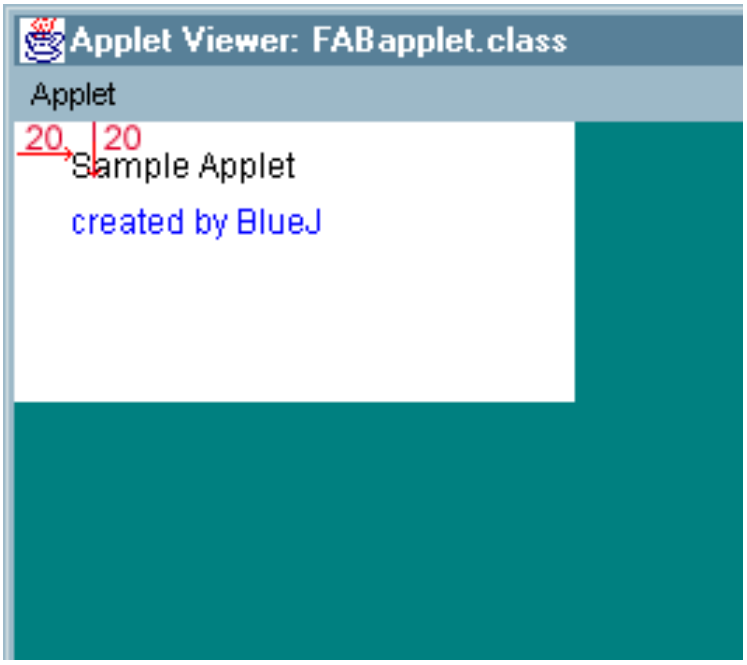


Figura 9 — Pixels e sistema de coordenadas na tela.

Restam ainda duas linhas de código a serem comentadas:

```
g.setColor(Color.white);
g.fillRect(0, 0, 200, 100);
```

Neste ponto, você não deverá ficar surpreso ao saber que

```
g.setColor(Color.white);
```

é uma chamada ao método setColor dirigida ao objeto g, da classe Graphics, que é usada para indicar a cor a ser usada como cor de fundo da janela onde são exibidas as mensagens. Essa cor é especificada pelo parâmetro usado na chamada – Color.white (cor branca). Além disso, você já deve ter adivinhado que:

```
g.fillRect(0, 0, 200, 100);
```

informa ao objeto g, da classe Graphics, para preencher o retângulo cujo canto superior esquerdo está localizado na posição (0, 0) e que tem largura e altura, respectivamente, de 200 e 100 pixels. Veja a figura anterior. Note que existe um retângulo branco na janela do applet.

E agora... o código

Depois de tudo isso, vamos finalmente escrever nosso tão esperado trecho de código para exibir na janela do applet a mensagem "João comeu pão". Tudo o que você tem a fazer é escrever o seu próprio comando drawString. Parece fácil, não é? Se você ainda não se sente seguro, leia as instruções a seguir.

Algumas dicas sobre a linguagem de Java

Aqui vão algumas dicas sobre a linguagem Java que irão ajudá-lo. Primeira

dica: todo comando de Java deve ser terminado com um ponto e vírgula (;;). Segunda dica: todo argumento de uma chamada ao método "drawString" deve uma expressão de tipo 'String'. Bem, mas o que é um String? Em computação, um string é qualquer seqüência de caracteres que representam letras, números ou outros símbolos, tais como ?, {, [, % etc. Por exemplo, "João comeu pão" é um String – atenção, para que o computador reconheça uma seqüência de caracteres como um String, ela deve ser escrita entre aspas duplas, tal como nesse caso.

Terceira dica: como você já deve ter percebido, o método drawString requer outros dois argumentos, cada um dos quais um número inteiro, que especificam a posição onde deve ser exibida a mensagem na janela do applet. Note que os argumentos usados na chamada de um método são separados por vírgula e que são especificados entre parênteses.

Finalmente, você pode adicionar o código para imprimir sua mensagem sua mensagem ao programa FABapplet. Uma vez que você tenha feito isso, compile novamente o programa, antes de executá-lo.

Se o seu programa estiver funcionando como esperado, podemos tentar mais algumas modificações. Altere o tamanho ou a localização do retângulo que é exibido na janela do applet, alterando os argumentos (0, 0, 200, 100) do método fillRect. Você pode experimentar, também, alterar a cor das mensagens e do retângulo. Experimente, por exemplo, Color.red (vermelho) Color.blue (azul) Color.green (verde) ou Color.yellow (amarelo).

Se você ainda tem muitas dúvidas a respeito de objetos, classes, métodos, chamada de método, parâmetros etc., não se afobe... Você ainda vai aprender muito sobre cada um desses conceitos. Para isso, vamos prosseguir criando nosso segundo projeto, na seção a seguir.

Criando uma aplicação console

Nesta seção, vamos aprender a criar uma aplicação console simples. Vamos primeiro seguir algumas instruções para fazer isso, passo a passo. Depois procuraremos entender todo o processo executado, com detalhes.

Criando mais um novo projeto

Finalmente, já sabemos como usar o BlueJ para criar um projeto. Execute o BlueJ e selecione a opção New Project do menu Project. Chame esse novo projeto de "consoleBlank" (ou outro nome que você preferir) e pressione o botão Create:

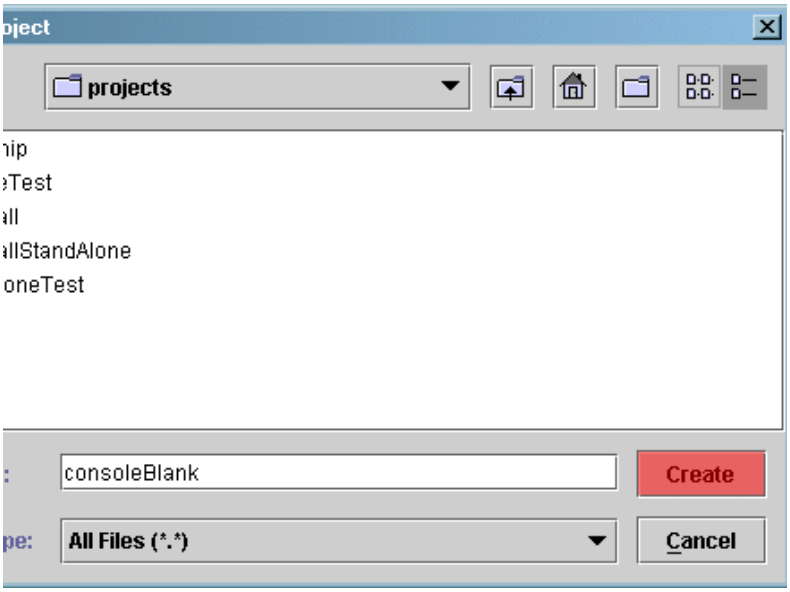


Figura 10 – Criando uma aplicação console simples.

O BlueJ então irá exibir o navegador de projeto, que deve conter um único ícone, representando um arquivo readme.txt, como vimos anteriormente. Pressione o botão New Class e selecione a opção Class na janela de diálogo Create New Class (Criar Nova Classe). Especifique o nome "ConsoleApp" para essa nova classe e pressione o botão Ok.

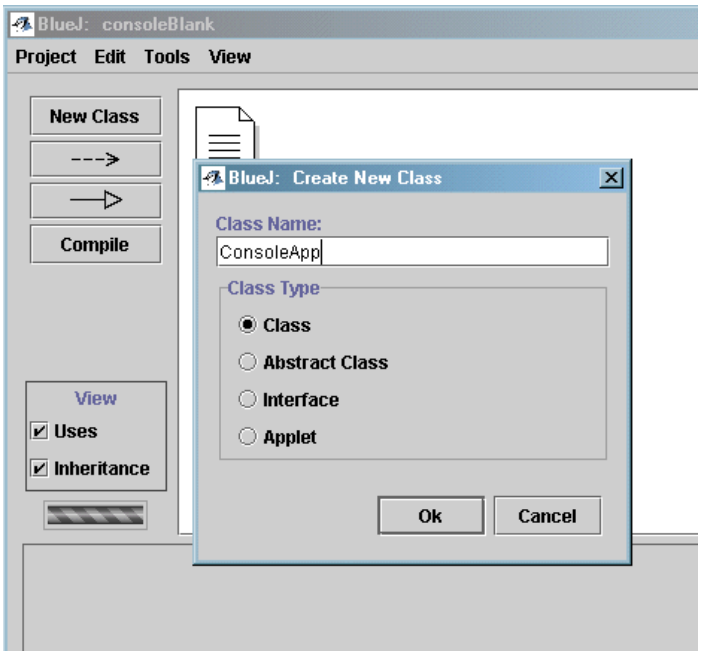


Figura 11 – Criando a classe "ConsoleApp".

O BlueJ deverá criar para você essa nova classe e exibir o ícone correspondente na janela do navegador de projeto. Note as marcas de hachura nesta classe, indiocando que ela deve ser compilada, antes de ser executada. Clique duas vezes, seguidamente, sobre esse ícone. Isso fará com que o código da classe, gerado automaticamente pelo BlueJ, seja exibido na tela. Esse código deve ser semelhante ao seguinte:

```
/**
 * Write a description of class ConsoleApp here.
 *
 * @author (your name)
```

```
 * @version (a version number or a date)
 */
public class ConsoleApp
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class ConsoleApp
    */
     public ConsoleApp()
    {
        // initialise instance variables
        x = 0;
    }

    /**
     * An example of a method - replace this comment with your own
     *
     * @param  y    a sample parameter for a method
     * @return      the sum of x and y
     */
     public int sampleMethod(int y)
    {
        // put your code here
        return x + y;
    }
}
```

Faça agora o seguinte:

1. Coloque seu nome onde está escrito (your name).

2. Coloque um número de versão, ou a data corrente, onde está escrito (a version number or date).

3. Apague o trecho de código existente desde a linha onde se lê private int x; até o penúltimo fecha chaves '}'.
Depois disso, seu código deve ser parecido com o seguinte:

```
/**
 * Uma aplicação console vazia.
 *
 * @author (seu nome aqui)
 * @version v0.1
 */
public class ConsoleApp
{
}
```

Agora, temos que adicionar à classe ConsoleApp o método main, como ilustrado a seguir:

```
/**
 * Uma aplicação console vazia.
```

```
 *
 * @author (seu nome aqui)
 * @version v0.1
 */
public class ConsoleApp
{
    /** Ponto inicial da execução do programa. */
    public static void main(String[] args) {
   }
}
```

Salve o arquivo e, em seguida, pressione o botão 'Compile' – seja na janela que mostra o código da classe ConsoleApp ou na janela do navegador de projeto. Depois que a classe ConsoleApp tiver sido compilada corretamente, o BlueJ deverá indicar que o programa está pronto para ser executado, removendo as linhas de hachura do ícone correspondente a essa classe:

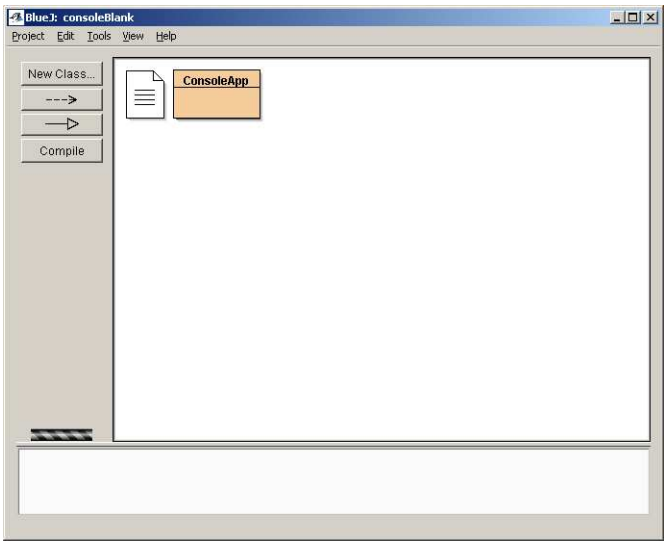


Figura 12 – ConsolApp pronto para ser executado (depois de compilado).

Humm... parece que você vai reclamar novamente: "Espere aí! Eu não estou aprendendo nada! Apenas seguindo instruções, tal como um computador!" VocÊ está certo, Agora que acabamos de ver como usar o ambiente BlueJ para criar uma aplicação console padrão, vamos procurar entender o que foi feito.

Isso envolve vários conceitos novos. Portanto, vamos tratar disso na próxima seção.

Declarações de classes, variáveis e métodos em Java

Na seção anterior, criamos uma nova classe e modificamos o código dessa classe para incluir o método main. Antes de continuar escrevendo o código do nosso programa, precisamos saber um pouco mais sobre classes, objetos e outros aspectos da linguagem Java.

Você deve estar lembrado da relação existente entre classes e objetos em programação – objetos são instâncias de classes. Em outras palavras, uma classe descreve um conjunto de objetos que possuem características e comportamento comuns e, com base nessa descrição, é possível criar, durante a execução do programa, objetos individuais que possuem as características e o comportamento descritos pela classe, podendo, entretanto, ter estados diferentes.

Reveja a estrutura básica de objetos mostrada na Figura 1 deste módulo. Objetos contêm variáveis e métodos. Como uma classe descreve (ou define) um conjunto de objetos, então ela deve consistir de definições de variáveis e de métodos. Se tentarmos imaginar a estrutura de uma classe, pensaríamos em algo semelhante a:

```
AQUI ESTÁ MINHA NOVA CLASSE, CHAMADA NomeDaClasse.
AQUI COMEÇA O CÓDIGO DA MINHA CLASSE.
    AQUI ESTÁ MINHA PRIMEIRA VARIÁVEL.
    AQUI ESTÁ A SEGUNDA VARIÁVEL.
   ...
    AQUI ESTÁ A ÚLTIMA VARIÁVEL.
    AQUI ESTÁ MEU PRIMEIRO MÉTODO.
    AQUI ESTÁ O SEGUNDO MÉTODO.
   ...
    AQUI ESTÁ O ÚLTIMO MÉTODO.
AQUI TERMINA O CÓDIGO DA MINHA CLASSE
```

Vamos ver agora como isso deveria ser escrito, usando as regras da linguagem Java:

1) A palavra reservada *class* deve ser usada para iniciar uma definição de classe, precedendo imediatamente o nome da classe. O nome da classe é, em geral, iniciado por letra maiúscula (isso não constitui uma regra, mas simplesmente uma boa prática de programação).

2) Um par de chaves (‘{‘ and ‘}’) é usado para indicar os limites da classe, isto é, o início e o fim do código da classe, respectivamente.
3) Seções podem conter outras seções, caso no qual os pares de chaves são aninhados:

```
{ // Este é o início da seção 1.
   { // Este é o início da seção 2.
   } // Este é o fim da seção 2.
} // Este é o fim da seção 1.
```

Aplicando essas regras à nossa tentativa anterior de escrever o código de uma classe, obtemos a seguinte versão:

```
class  NomeDaClasse
{
    AQUI ESTÁ MINHA PRIMEIRA VARIÁVEL.
    AQUI ESTÁ A SEGUNDA VARIÁVEL.
   ...
    AQUI ESTÁ A ÚLTIMA VARIAÁVEL.

    AQUI ESTÁ MEU PRIMEIRO MÉTODO.
    AQUI ESTÁ O SEGUNDO MÉTODO.
   ...
    AQUI ESTÁ O ÚLTIMO MÉTODO.
}
```

Vamos agora saber um pouco mais sobre o significado de variáveis em programas e sobre como variáveis são definidas em um programa Java.

Variáveis são usadas em programas para armazenar informação. Uma variável representa, de fato, um lugar na memória do computador, no qual a informação é armazenada. Toda variável possui um nome, que é utilizado no programa para referenciar esse lugar

de memória, ou o valor que está aí armazenado. Uma variável possui também um tipo, que determina os valores que podem ser armazenados nesse local da memória. Sabendo isso, é fácil imaginar como seria a definição (ou declaração) de uma variável, em Java:

1) A primeira parte da declaração de uma variável especifica o seu tipo. Alguns exemplos de tipos válidos em Java são: int – que representa números inteiros (tais como 1, -99, ou 0); float – que representa 'números de ponto flutuante', ou seja, números que possuem uma parte decimal (tais como 3.1415926 ou -0.3 – note que, em Java, a parte decimal é separada da parte inteira por um ponto, e não por vírgula); String – que representa seqüências de caracteres (isto é, seqüências de letras, algarismos ou outros símbolos), que devem ser escritos entre aspas duplas (tais como "Bom dia", "THX-1138", ou "5793"). Nos programas que você irá desenvolver ao longo deste tutorial, terá oportunidade de usar variáveis de diferentes tipos. Nesse ponto, é bom que você conheça os tipos primitivos da linguagem Java – clique aqui.

2) O nome da variável deve ser especificado imediatamente depois do seu tipo. Usualmente, o nome de uma variável é iniciado por uma letra minúscula (isso não é uma regra da linguagem Java, mas apenas uma prática usual em programas).

3) Uma declaração de variável deve terminar com um ponto e vírgula (;).
Aplicando essas novas regras ao nosso código, escreveríamos:

```
class  NomeDaClasse
{
    tipo nomeDaVariavel1;
    tipo nomeDaVariável2;
   ...
    tipo nomeDaVariavelN;

   AQUI ESTÁ MEU PRIMEIRO MÉTODO.
   AQUI ESTÁ O SEGUNDO MÉTODO.
  ...
   AQUI ESTÁ O ÚLTIMO MÉTODO.
}
```

Para ilustrar a idéia de definição de classes, vamos considerar a definição de uma classe "Conta", que descreve objetos que representam contas bancárias. Uma das principais características de uma conta bancária é o seu saldo. Pelo que acabamos de ver, a classe "Conta" certamente deveria conter uma declaração de variável para representar o saldo da conta. Essa declaração teria a forma: float saldo;

Quais seriam os métodos que deveriam ser definidos na classe "Conta"? Lembre-se que um método definido em uma determinada classe descreve uma ação (ou operação) que pode ser executada por objetos dessa classe. Ou seja, o código do método consiste de instruções que descrevem, passo a passo, como essa ação deve ser executada.

Você responderia, então, que, certamente, a classe "Conta" deve conter definições de métodos "depósito", "retirada" e "saldo", que descrevem as operações correspondentes sobre contas bancárias. Os métodos "depósito" e "retirada" deveriam ter como parâmetros, respectivamente, o valor a ser depositado e o valor a ser retirado da conta. A operação definida por esses métodos consistiria apenas em somar, ou subtrair ao saldo da conta, respectivamente, o valor especificado como argumento para o método (modificando assim o estado da conta). O método "saldo" não teria parâmetros e

deveria retornar, como resultado de sua execução, o valor corrente do saldo da conta.

Vamos então, finalmente conhecer as regras para definição de métodos da linguagem Java, que nos permitirão expressar operações como as descritas acima.

1) A primeira parte da definição de um método especifica o tipo do resultado produzido pela ação descrita por esse método.

2) Se o método não produz nenhum resultado, o tipo do resultado deve ser especificado como sendo 'void'.

3) O nome do método deve ser especificado imediatamente depois do tipo do resultado.

4) Logo depois do nome do método, deve ser especificada a lista de parâmetros do método, entre parênteses – '(' e ')'. Parâmetros provêm informação adicional necessária para a execução da ação descrita pelo método. Por exemplo, um objeto que representa uma conta bancária, deveria ter um método 'depósito', que teria como parâmetro o valor a ser depositado. Caso o método não possua parâmetros, o nome do método deverá ser seguido de uma lista de parâmetros vazia, representada como '()' (abre e fecha parênteses).

5) Um par de chaves – '{' e '}' - é usado para indicar, respectivamente, o início e o fim do corpo (ou código) do método.
Puxa, são regras demais! De fato, mas as regras são todas bastante simples. Aplicando essas regras, obteríamos a seguinte estrutura para uma declaração de classe em Java:

```
class NomeDaClasse
{
    type nomeDaVariavel1;
    type nomeDaVariavel2;
   ...
    type nomeDaVariavelN;

    type metodo1(type param1, type param2, ...) {
       // O corpo do método entra aqui.
   }
    type metodo2(type param1, type param2, ...) {
       // O corpo do método entra aqui.
   }
   ...
    type metodoN(type param1, type param2, ...) {
       // O corpo do método entra aqui.
   }
}
```

Voltando à nossa classe ConsoleApp

Agora que sabemos um pouco mais sobre a declaração de classes, variáveis e métodos em Java, podemos voltar à nossa classe ConsoleApp. A parte inicial do código dessa classe é mostrada a seguir:

```
/**
 * A blank console application, ripe with possibility.
 *
 * @author (seu nome aqui)
```

```
 * @version v0.1
 */
```

Essa parte inicial consiste apenas de um comentário. Lembre-se que, em um programa Java, todo o texto escrito entre /* e */ é considerado comentário, sendo, portanto, ignorado pelo computador. Também é considerado comentário todo o texto escrito depois de //, em uma linha do programa.
Removendo o comentário, obtemos:

```
public class ConsoleApp
{
    public static void main(String[] args) {
      // O corpo do método main entra aqui.
   }
}
```

Esse código define a classe ConsoleApp, a qual contém apenas uma definição de método – o método main. Mas o que significam as palavras 'public' e 'static', usadas neste programa?

As palavras reservadas 'public' e 'static' especificam atributos de classes, variáveis e métodos. Em termos simples, o atributo 'public' (público) indica que a classe ou método é visível em todo o programa: objetos de uma classe pública podem ser criados por quaisquer outros objetos do programa; métodos e variáveis públicos podem ser usados em qualquer parte do código do programa. Classes, métodos e variáveis podem também ser declarados 'private' (privado) ou 'protected' (protegido). Uma variável ou método protegido apenas pode ser usado dentro do corpo da classe na qual é declarado. O atributo 'protected' especifica uma visibilidade intermediária entre 'public' e 'protected'. O significado preciso desses atributos será visto mais adiante, quando também discutiremos a razão pela qual esses atributos são usados em programas. Por enquanto, você precisa apenas lembra-se que o método main e a classe que contém esse método devem ser declarados como públicos. Isso é necessário para que o computador possa encontrar o ponto a partir do qual deve iniciar a execução do programa.

Para não complicar ainda mais, vamos deixar para falar sobre o significado do atributo 'static' mais adiante. Porém, saiba que é sempre obrigatório declarar o método main como 'static'. De fato, a declaração do método main deve ter sempre a forma acima.

O que mais podemos saber sobre o método main, examinando a definição deste método? Note o tipo indicado para o valor de retorno do método - 'void'. Como vimos, isso indica que o método main não retorna nenhum valor. Note também que o método main possui um parâmetro – String[] args - especificado entre parênteses, logo em seguida ao nome do método. Hummm... o que significa "String[] args"?

Assim como no caso da definição de uma variável, a definição de um parâmetro requer duas informações: o tipo e o nome do parâmetro. Portanto, "String[] args" nos diz que o parâmetro do método main tem tipo String[] e nome args. Já sabemos o que significa o tipo String, você se lembra? Mas qual é o significado de String[]?

Java usa colchetes ('[ ]') para denotar um 'Array'. De modo simples, podemos dizer que uma Array é uma seqüência de valores, de um determinado tipo, tal que os elementos dessa seqüência podem ser identificados pela posição que ocupam nessa seqüência. Por exemplo, considere uma variável – 'compositores' – para armazenar informação sobre

nomes de compositores da MPB. Essa variável poderia ser um Array de elementos de tipo String, contendo os seguintes valores:

"Caetano"

"Gilberto Gil"

"Chico Buarque"

"Cazuza"

Para referenciar o segundo elemento desse Array (isto é, "Gilberto Gil"), escreveríamos, em Java, compositores[1]. Êpa, espere aí! Não deveria ser compositores[2]? Em Java, assim como em várias outra linguagens de programação, os elementos de um Array são numerados a partir de 0, e não de 1. Portanto, compositores[0] contém "Caetano", compositores[1] contém "Gilberto Gil" etc.

Vamos então resumir o que sabemos sobre o método main:

- Ele é public, o que significa que qualquer um pode tera cesso a ele.
- Ele é static – veremos o que isso significa, não se afobe...
- Ele retorna um resultado void, o que significa que, de fato, ele não retorna nenhum resultado.
- Ele é chamado main – como já sabemos.
- Ele tem um único parâmetro, chamado 'args', que consiste em um 'array' de Strings, o que é indicado pelo tipo String[].

Tudo isso está especificado no cabeçalho da declaração do método – a qual está contida em uma declaração de classe. Ah! Não se esqueça: todo programa deve ter uma única classe onde é declarado o método main – é pela primeira instrução do corpo desse método que é iniciada a execução do programa.

Na próxima seção, vamos de fato começar a escrever algum código. Não perca tempo!

Incrementando nossa aplicação console

Hora de escrever código! Primeiramente, vamos fazer uma cópia do nosso projeto ConsoleBlank - vamos guardar esse console simples para ser usado como base em projetos futuros.

Fazendo uma cópia do projeto ConsoleBlank

Na tela do BlueJ, selecione a opção Project e, em seguida, Open Project..., para abrir nosso projeto ConsoleBlank, como fizemos anteriormente. A tela do BlueJ deverá então apresentar-se assim:

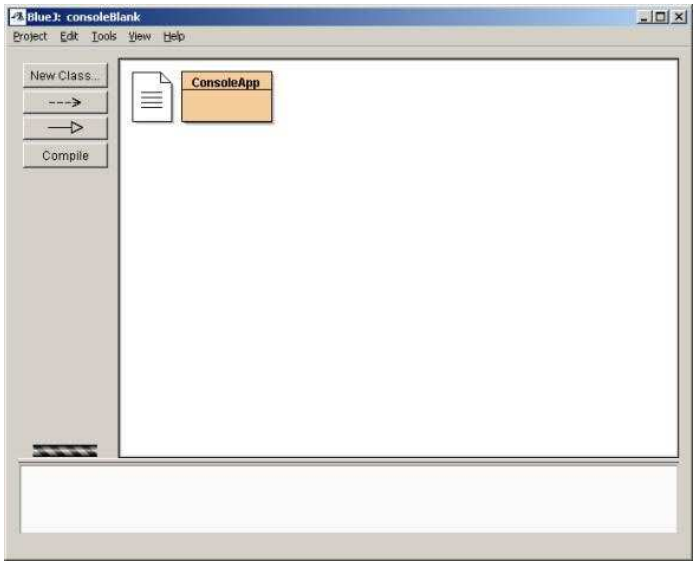


Figura 13 – Voltando à nossa aplicação console.

Selecione agora, novamente, a opção Project e, em seguida, Save As... (Salvar Como), para salvar uma cópia da nossa aplicação console, com um novo nome: especifique o nome FABConsole, como mostrado na figura a seguir, e, em seguida, clique no botão Save (Salvar).

Figura 14 – Salvando uma cópia da aplicação console.

Mais uma vez, o BlueJ abrirá seu navegador de projeto, mostrando na tela o ícone de um arquivo texto e o ícone correspondente ao arquivo que contém o código da classe ConsoleApp. Clique duas vezes, seguidamente, sobre esse ícone, para fazer abrir a janela do editor do BlueJ, mostrando o código da classe.

Escrevendo código...

Agora estamos prontos para modificar nossa classe ConsoleApp, de maneira a execução do método main, definido nessa classe, resulte em exibir na janela do nosso console a mensagem "João comeu pão.".

Você ainda se lembra da aplicação que criamos inicialmente neste módulo? Então deverá lembrar-se que usamos o método drawString para escrever um texto na janela de um applet. Nesse caso, tivemos que informar, explicitamente, as coordenadas da posição da janela onde a mensagem deveria ser escrita e a cor dos caracteres do texto. Dessa vez, vai ser tudo muito mais simples...

Como vimos na introdução deste módulo, aplicações console exibem suas mensagens

de saída no console Java. No console, o texto é exibido seqüencialmente, linha por linha, sempre com a mesma cor – tal como em uma máquina de escrever. Não podemos especificar a posição onde o texto será exibido ou a cor dos caracteres do texto. Em compensação, para exibir uma mensagem no console, precisamos apenas escrever o seguinte comando, bem mais simples:

```
System.out.println("insira sua mensagem aqui.");
```

Fácil, não é? Adicione o comando acima ao corpo do método main, isto é, entre os símbolos '{' e '}' que delimitam o início e o fim do corpo do método. Não se esqueça de modificar a mensagem para o que desejamos – "João comeu pão. " (bem, você pode pensar em uma mensagem mais interessante).

Pressione agora o botão 'compile', para fazer com que seu programa seja compilado. Caso seu programa seja compilado sem erros, aparecerá na barra inferior da janela do BlueJ a seguinte mensagem: "Class compiled – no syntax errors.". Caso contrário, isto é, se você obtiver algum erro de compilação, clique sobre o símbolo de ponto de interrogação ('? ') que aparecerá no canto inferior direito da janela do BlueJ – isso fará com que a linha do programa onde ocorreu o erro seja destacada, sendo também exibida uma explicação detalhada sobre o erro (peça ajuda ao seu professor para entender a mensagem de erro, que estará escrita em inglês). Para ajudá-lo, listamos alguns erros mais comuns, a seguir:

- Não escrever System com a primeira letra maiúscula.
- Esquecer do símbolo de ponto e virgule no final do comando.
- Esquecer de escrever a mensagem entre aspas duplas.
- Usar aspas simples, em vez de aspas duplas.
- Escrever alguma das palavras acima de forma errada.
- Escrever o comando for a do corpo do método main.
- You used single quotes instead of double quotes.

Se você definitivamente não conseguiu obter um programa sem erro, dê uma olhadinha no "código 1" disponível na seção de códigos, ao final do tutorial.

Depois de compilar seu projeto, você deverá testá-lo. Para isso, volte à janela do navegador de projeto do BlueJ e clique com o botão direito do mouse sobre o ícone que representa a classe ConsoleApp. Escolha então a opção void main(args) – isso instrui o sistema de execução de programas Java a procurar o método main, para iniciar a execução do programa da maneira adequada. Como resposta, o BlueJ irá exibir a seguinte janela de diálogo:

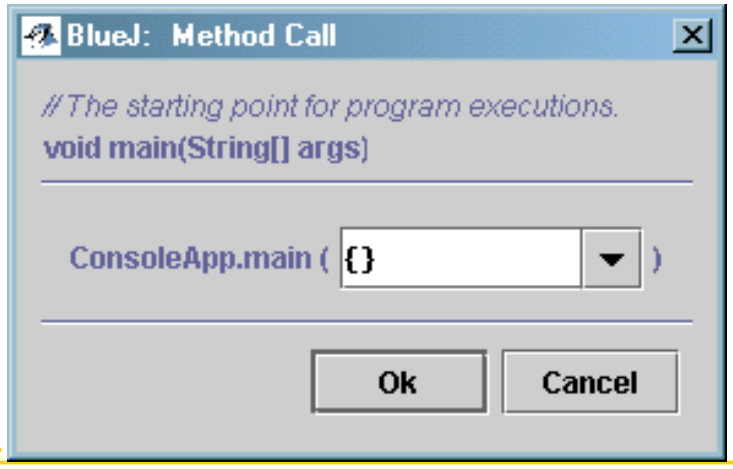


Figura 15 – Iniciando a execução de uma aplicação console.

Por enquanto, apenas pressione o botão Ok.

O BlueJ irá então iniciar a execução do seu programa. Isso resultará em que seja exibida a janela a seguir – cujo título é "Terminal Window" – na qual aparecerá escrita a sua mensagem.

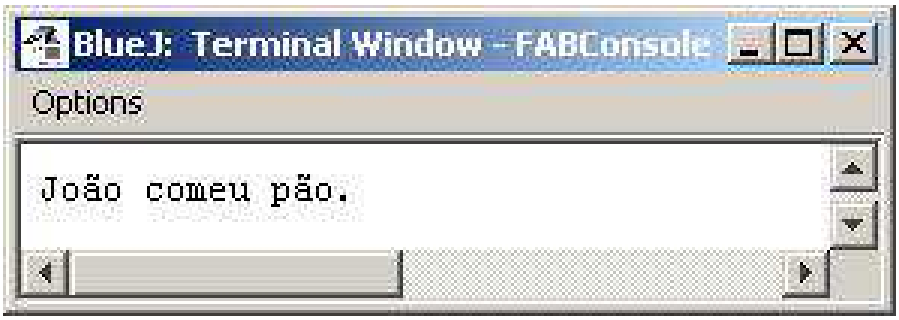


Figura 16 – Visão da janela do console.

Parabéns! Você acabou de concluir sua aplicação console! Entretanto, embora esse primeiro programa seja bastante simples, existe ainda um bocado de coisa que precisamos explicar, para que você possa compreender melhor o significado do único comando que incluímos no corpo do método main. Bem, aí vai um pouco mais de informação...

Chamada de métodos...

Vamos recordar um pouco do que já vimos anteriormente. Sabemos que a execução de um programa Java consiste, essencialmente, em criar objetos e processar interações entre esses objetos, por meio da execução de chamadas de métodos definidos nas classes que descrevem esses objetos. Mas essa é uma descrição um tanto vaga... Aposto como você gostaria de entender um pouco melhor o que acontece de fato.
Primeiramente, você precisará entender como expressar, em Java, o acesso a dados armazenados em objetos e a execução de ações por esses objetos. Suponha, por exemplo, que você definiu, em seu programa, uma classe 'Cão', na qual é declarada uma variável 'raça', do tipo String, e um método 'latir()'. O código da classe Cão teria a seguinte forma:

```
/** Uma classe que representa cães. */
public class Cão {
   /** Variáveis */
   String raça = "Poodle";
   ...

   /** Métodos */
   void latir() {
      // O código do método entra aqui.
   }
}
```

Considere agora o código para criar um objeto da classe Cão, em sua classe ConsoleApp, e referenciar a variável raça e o método latir() desse objeto:

```
/** Uma ConsoleApp que cria um objeto da classe Cão. */
public class ConsoleApp {
   /** Métodos */
   public static void main(String[] args) {
```

```
        // Criando uma variável para armazenar um objeto da classe Cão.
        // Essa declaração tem o formato tipo e nome da variável, como vimos anterior-
mente,
       // e, além disso, código para criar um objeto da classe Cão e atribuir esse objeto à
variável  meuCão.
       Cão meuCão = new Cão();

       // Imprimindo a raça de meuCão.
       System.out.println(meuCão.raça);

       // Fazendo meuCão latir.
       meuCão.latir();
   }
}
```

Para obter informação sobre a raça do cão, usamos o comando 'meuCão.raça'. Para "fazer o cão latir", usamos 'meuCão.latir()'. Em outras palavras: 'meuCão.raça' significa "obtenha a informação sobre a raça do objeto particular meuCão, da classe Cão"; 'meuCão.latir()' é uma chamada ao método latir(), dirigida ao objeto meuCão – a execução dessa chamada de método irá fazer com que sejam executadas as instruções do método latir(), definido na classe Cão (por exemplo, gerando um som correspondente ao latido do cão em questão). Note que, em ambos os casos, o nome do objeto que é referenciado precede o nome da variável ou método.

Você deve ter observado que a declaração da variável meuCão difere um pouco das que vimos anteriormente. Você deve estar lembrado de que a declaração de uma variável deve especificar o tipo e o nome dessa variável. O tipo da variável meuCão é Cão – toda declaração de classe em um programa introduz nesse programa um novo tipo, isto é, o nome dessa classe pode ser usado para especificar o tipo de variáveis que designam objetos dessa classe.
Você deve ter observado, também, que a declaração da variável meuCão inclui uma parte adicional. Uma declaração de variável pode indicar, opcionalmente, um valor que deve ser armazenado na variável. Por exemplo, a declaração de variável:

```
float posX =  0.0;
```

especifica o tipo da variável posX – ou seja float – e indica que o valor 0.0 deve ser armazenado nessa variável. Voltemos à declaração da variável meuCão:

```
Cão meuCão = new Cão();
```

Nessa declaração, é usada a expressão new Cão(); para criar um objeto da classe Cão. A avaliação dessa expressão resulta na criação de um objeto da classe Cão, retornando uma referência a esse objeto – essa referência ao objeto criado é então armazenada na variável meuCão. A expressão que indica a criação de um objeto, em Java, consiste da palavra reservada new (que significa novo), seguida por um construtor de objeto (ou simplesmente construtor). Um construtor pode ser visto como um método particular que define a criação de um objeto da classe. Um construtor deve ter sempre o mesmo nome da classe e pode, opcionalmente, incluir parâmetros, tal como qualquer outro método. Além disso, a declaração de um construtor difere da declaração de um método por não ser necessário especificar, nesse caso, o tipo do valor de retorno.

Se você for bastante atencioso, deve estar pensando agora que a nossa classe Cão não

inclui nenhuma declaração desse tal construtor Cão(), que é usado na expressão new Cão(); para criar um objeto da classe. Felizmente, em Java, toda classe tem implicitamente definido um construtor, de mesmo nome da classe, sem parâmetros, que pode ser usado para a criação de objetos da classe. Portanto, você apenas precisará declarar um construtor, em uma classe, quando desejar especificar alguma operação adicional que deve ser realizada ao ser criado um objeto da classe, tal como, por exemplo, atribuir valores iniciais a variáveis do objeto que está sendo criado, determinando o estado do objeto. Veremos, mais tarde, outros exemplos de criação de objetos e de definição de construtores de objetos.

Por enquanto, vamos voltar ao código do método main da nossa classe ConsoleApp.

A classe System e o objeto out

Você deve lembra-se que o corpo do método main da nossa classe ConsoleApp inclui um único comando:

```
System.out.println("João comeu pão.");
```

A execução desse comando resulta em exibir, na janela do console, a mensagem "João comeu pão". Como isso é feito? Bem, o comando acima é uma chamada ao método println, dirigida ao objeto referenciado pela variável out, a qual é declarada na classe System. Sabemos que out é uma variável, e não um método, uma vez que, no comando acima, out não é imediatamente seguido de uma lista de parâmetros (possivelmente vazia), como é o caso de println, que é um método. Você já'deve saber que a mensagem "João comeu pão." é o argumento (ou parâmetro) dessa chamada ao método println.

Em outras palavras, ao executar o comando System.out.println("João comeu pão."); o sistema de execução de programas Java busca pela classe System, encontra a variável out, declarada nessa classe, obtém o objeto referenciado por essa variável e dirige, a esse objeto, uma chamada ao método println, passando como argumento a mensagem a ser exibida na tela.

Hummm.... algumas questões devem estar te incomodando, neste momento... No exemplo discutido anteriormente, a chamada ao método latir() foi dirigida ao objeto meuCão, o qual teve que ser previamente criado, por meio da expressão new Cão(). Nesse comando da nossa classe ConsoleApp, o objeto out é referenciado sem que tenha sido previamente criado. Como isso é possível? E a classe System, onde ela está declarada? Além disso, diferentemente dos exemplos anteriores, estamos referenciando uma variável – out – de uma determinada classe – System – e não de um objeto. Qual é a diferença? Aqui vão as respostas.

A classe System pertence à biblioteca de classes padrão da linguagem Java – um conjunto de coleções de classes pré-definidas, que podem ser usadas em qualquer programa Java. Essa classe contém declarações de variáveis que representam informações sobre o sistema operacional, assim como métodos que provêem uma interface acesso às operações do sistema. Quando é iniciada a execução de qualquer programa Java, é criado um objeto da classe System.

Como dissemos anteriormente, a variável out é declarada na classe System. Ela referencia a saída de dados padrão do sistema operacional – ou seja, o console (como é chamada a tela do computador). Em outras palavras, no início da execução de todo programa

Java, é criado um objeto que representa o console e uma referência a esse objeto é atribuída à variável out. Esse objeto provê, ao programa, acesso às operações básicas de saída de dados do sistema operacional, tal como a operação de exibir um texto na tela, implementada pelo método println. Também é declarada, na classe System, uma variável – in – que referencia um objeto que representa a entrada de dados padrão, ou seja, o teclado.

A variável out é declarada com o atributo 'static' – dizemos que out é uma variável estática. O uso do atributo static em uma declaração de variável indica que o valor dessa variável é o mesmo, para todos os objetos da classe. Por esse motivo, variáveis estáticas são também chamadas de variáveis de classe, em contraposição a variáveis de instância (declaradas sem esse atributo), cujo valor pode ser diferente para cada objeto da classe que é criado no programa – o estado de cada objeto é representado pelos valores de suas variáveis de instância. Como o valor de uma variável estática é o mesmo para todos os objetos da classe, ela pode ser referenciada por meio do nome da classe – tal como em System.out – ao invés de ser referenciada por meio de um objeto, tal como, por exemplo, em meuCão.raça.

Vimos que o atributo 'static' também pode ser usado em uma declaração de método – lembre-se da declaração do método main. O significado, aí, é similar: indica que o método não pode ser redefinido em nenhuma outra classe do programa (veremos, mais tarde, qual é a utilidade da redefinição de métodos em programas). Assim como no caso de variáveis estáticas, uma chamada a um método estático pode ser feita indicando-se o nome da classe em o método é declarado, ao invés ser dirigida a um objeto dessa classe.

Parece que tudo ficou bem mais claro agora, não é? Se você ainda tem dúvidas, não se afobe. Ainda temos muito pela frente. No próximo módulo, já vamos começar a implementar o arcabouço da interface gráfica do nosso jogo de Batalha Naval. Mas, antes disso, vamos resumir o que aprendemos neste módulo.

Conclusão

Uau! De fato aprendemos um bocado de coisas novas neste módulo:

O significado dos conceitos de classes, objetos, variáveis e métodos em programas orientados a objetos.

A sintaxe de declarações de classes, variáveis e métodos em Java.

A sintaxe e significado de uma chamada de método.

Os diferentes tipos de tipos de programas Java: applets, consoles e programas standalone, com interface gráfica.

Já tivemos oportunidade de criar um programa de cada um dos dois primeiros tipos acima e sabemos como usar métodos, pré-definidos na biblioteca padrão de classes de Java, para exibir uma mensagem na janela de um applet ou na janela do console Java. No próximo módulo vamos construir uma aplicação semelhante ao nosso console simples, a qual usa, entretanto, uma interface gráfica para ler e exibir dados. Vamos também aprender mais sobre Java e finalmente começar a implementar o arcabouço do nosso jogo de Batalha Naval.

Vejo você então, no próximo módulo.

APÊNDICE – TIPOS PRIMITIVOS EM JAVA

A linguagem Java oferece diversos tipos primitivos, que representam tipos de valores comumente usados em programas, tais como valores numéricos, valore booleanos e caractereres. Fornecemos abaixo uma breve descrição sucinta dos tipos primitivos disponíveis em Java. Se você quiser saber mais, consulte a página web http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html.

Tipos numéricos

O tipo int representa valores numéricos inteiros. Existem, de fato, 4 diferentes tipos primitivos, em Java, para representar números inteiros, dependendo da quantidade de espaço usada para representar tais números na memória do computador. Um valor do tipo int é representado com 32 bits, ou binary digits (dígitos binários). Um valor do tipo int deve estar compreendido entre -2147483648 e 2147483647 – isto é, de -231 a 231-1 – o que, em geral, é uma faixa de valores suficientemente grande para os nossos propósitos.

Se você necessitar de uma maior faixa de valores inteiros, poderá usar o tipo primitivo long, que representa valores de -263 a 263-1. Um literal de tipo long, por exemplo 80951151051778, pode também ser escrito na forma 80951151051778L (isto é, com o caractere L (ou l) no final do número).

Existem também dois tipos para inteiros de menor valor: o tipo short para valore inteiros representados com 16 bits e o tipo byte, para valores inteiros representados com 8 bits.

IMPORTANTE: Como tipos inteiros representam números compreendidos em uma faixa limitada de valores, cálculos envolvendo valores desses tipos podem causar erros, caso o resultado de uma operação seja um valor fora da respectiva faixa de valores.

Números reais são representados de forma aproximada em um computador, usando o que chamamos de notação de ponto flutuante. Existem dois diferentes tipos para representação de números de ponto flutuante: os tipos float e double (número de ponto flutuante de precisão dupla). Valores de tipo float são representados com 32 bits. O maior valor desse tipo é aproximadamente 3.4x1038; o menor é aproximadamente -3.4x1038. O tipo float pode representar números com uma precisão de 8 algarismos decimais.

Valores de tipo double são representados com 64 bits. O maior valor de tipo double é aproximadamente 1.8x10308; o menor é aproximadamente -1.8x10308. O tipo double pode representar números com uma precisão de 16 algarismos decimais. O tipo double oferece, portanto, uma representação mais precisa para valores numéricos, assim como uma maior faixa de valores, sendo, em geral, mais adequado para cálculos científicos.

Booleanos

O tipo boolean representa valores do tipo booleano, isto é, o conjunto dos valores true (verdadeiro) e false (falso).

Caracteres e Seqüências de Caracteres

Caracteres são valores do tipo char. Literais desse tipo são escritos entre aspas simples, tal como: ‘x’.

Caracteres em Java são representados usando uma codificação denominada unicode, a qual é uma extensão da codificação mais comumente usada anteriormente, a codificação ascii. A codificação ascii codifica cada caractere usando 8 bits. A codificação unicode usa 16 bits para a representação de cada caractere.

Caracteres que não podem ser diretamente digitados no teclado, podem ser representados em um programa utilizando-se o caractere de escape. Por exemplo, o caractere de tabulação horizontal é representado como '\t'; o newline (nova linha) é representado como '\n'; o caracteres de aspas simples é representado por '\'', o caractere de aspas duplas é representado por '\"', o caractere de fim de linha é representado como '\\'.

Caracteres também podem ser especificados em um programa por meio do seu código numérico unicode, precedidos pelo prefixo \u.

Seqüências de caracteres são representadas por valores do tipo String. Por exemplo, "um String!" é um literal do tipo String. O tipo String de fato não é um tipo primitivo em Java, mas sim um tipo-classe (note que o nome do tipo é iniciado com letra maiúscula, ao contrário dos nomes de tipos primitivos, que começam com minúscula). Um literal do tipo Stringl é uma seqüência de caracteres escrita entre aspas duplas: "Oba, mais um String!". A seqüência pode conter caracteres escritos com escape, tal como em "Alô, mundo!\n" – nesse caso, o string termina com o caracteres newline ('\n').

Os nomes de tipos primitivos são palavras reservadas em Java. Isso signiofica que são palavras com significado especial, que não podem ser usadas com significado diferente em um programa.

MÓDULO 3

ESCREVENDO LOOPS E DEPURANDO PROGRAMAS

INTRODUÇÃO

Neste módulo vamos aprender mais um bocado sobre programação, em particular, como escrever, em Java, comandos de repetição – ou seja, comandos usados para especificar a execução de uma determinada ação repetidas vezes. Isso é importante porque o uso de comandos de repetição é extremamente comum em programas: diversas ações complexas consistem na repetição de um conjunto de ações mais simples.

Vamos também aprender a utilizar um depurador de programa – uma ferramenta que nos permita acompanhar a execução do código do programa, mostrando informações que são úteis para nos ajudar a encontrar erros no programa.

Finalmente, vamos começar a programar nosso jogo de Batalha Naval, escrevendo o código que implementa o arcabouço da interface gráfica do jogo.

Resumindo, os tópicos abordados neste módulo são os seguintes:

- Como passar parâmetros para programas Java.
- Comandos de repetição, ou loops, em Java
- Como usar o BlueJ como ferramenta para depuração de programas Java.
- Primeiros passos na implementação do jogo de Batalha Naval.

PASSANDO ARGUMENTOS PARA PROGRAMAS JAVA

Você se lembra da nossa definição de programa? Então saberá que um programa envolve três coisas: obter dados de entrada, manipular esses dados e produzir, como resultado, dados de saída. Nossa primeira aplicação – ConsoleApp – realiza apenas uma dessas funções: ela produz uma saída, isto é, exibe uma mensagem na tela. Programas podem obter dados de entrada de diversas maneiras: a partir de arquivos em disco, ou do teclado, ou de uma conexão de rede etc. Veremos, nesta seção, uma maneira simples de fornecer dados como entrada para um programa: especificando valores para os parâmetros do método main.

O conteúdo desta seção não é essencial para a compreensão do restante deste tutorial. Entretanto, decidimos incluí-la porque você pode estar curioso para saber o significado do conteúdo da janela de diálogo exibida pelo BlueJ, quando solicitarmos a execução do método main de um programa:

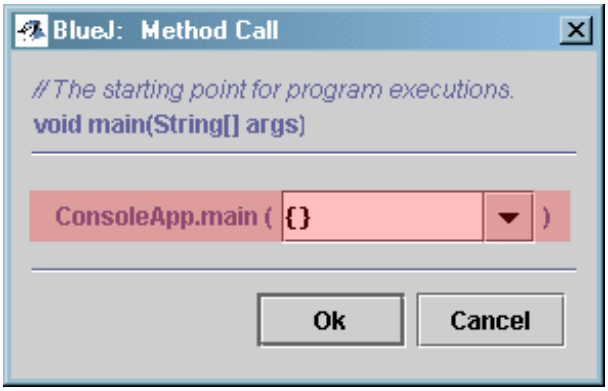


Figura 1 – Chamando o método main de um programa.

Note o título da janela de diálogo: "BlueJ: Method Call" – isso significa que será feita uma chamada de método. Observe agora a primeira linha: 'void main (String[] args)' – o método a ser chamado é main. Passemos então à terceira linha:

```
ConsoleApp.main({})
```

Note que os símbolos '{}' aparecem em combo box, isto é, um campo para entrada de texto. O BlueJ sabe que o método main tem, como parâmetro, um array de strings. Essa janela de diálogo nos oferece oportunidade de especificar os elementos desse array, para que esses valores sejam passados para o método main, antes que sua execução seja iniciada. Precisamos apenas digitar essa informação, e o BlueJ garantirá que ela será recebida pelo método main. Vamos experimentar?
Podemos atribuir valores iniciais a uma variável do tipo String[] (ou seja, um array de strings), em sua declaração, do seguinte modo:

```
String[] nomes = {"João", "Maria", "Paulo", "Pedro"};
```

Observe novamente a Figura 1: veja o par de chaves exibido no campo de texto. Você já deve ter adivinhado que devemos especificar, entre essas chaves, os elementos do array de strings a ser passado para o método main. Portanto, experimente digitar aí o valores "João", "Maria", "Paulo", "Pedro", como mostrado na janela a seguir:

Figura 2 – Passando argumentos para o método main.

Clique agora no botão 'Ok', para iniciar a execução do programa. Bem, ele continua funcionando exatamente como anteriormente, não é? Embora tenhamos passado argumentos para o programa, esses valores não são referenciados em nosso código, sendo, portanto, ignorados. Vamos então modificar o código no nosso programa, e maneira a usar o valores passados como argumento para o método main.
Clique duas vezes, seguidamente, sobre o ícone da classe 'ConsoleApp', para que o código dessa classe seja exibido pelo editor de textos do BlueJ. Encontre o comando "System.out.println...", no corpo do método main. Vamos substituir esse comando por outros similares, de maneira a imprimir argumentos passados para o método main. Tente o seguinte:

```
System.out.println(args[0]);
System.out.println("Alô " + args[1]);
```

Ok, já vamos explicar tudinho... Antes, porém, compile a classe ConsoleApp, corrigindo erros que eventualmente sejam cometidos. Em seguida, execute o programa, como anteriormente, clicando sobre o ícone da classe ConsoleApp e selecionando, em seguida,

a opção "void main(args)". Digite então "João", "Maria", "Paulo", "Pedro", no campo de texto da janela de diálogo 'method call', e depois clique 'Ok'. Se seu programa não funciona corretamente, consulte o código 2, na seção de códigos ao final deste tutorial, para ver a solução. Ao ser executado, programa deverá exibir o seguinte:

Figura 3 — ConsoleApp exibindo argumentos.

Agora, vamos entender melhor os comandos introduzidos. Lembre-se que um elemento de um Array pode ser obtido indicando-se sua posição no array (sabendo que as posições no array são numeradas a partir de '0'). Portanto, args[0] corresponde a "Pedro", args[1] corresponde a "Maria", etc.

Sabendo isso, fica fácil saber que a execução do comando System.out.println(args[0]) simplesmente imprime, no console, o primeiro elemento do array passado como argumento par ao método main, ou seja, esse comando imprime "João" e então muda para a linha seguinte.

Vejamos agora o comando System.out.println("Alô " + args[1]). Esse comando, como o anterior, imprime um String na janela do console: o String obtido como resultado da avaliação da expressão "Alô " + args[1]. O símbolo '+' indica, nessa expressão, uma operação de concatenação de strings – por exemplo, o resultado da expressão "Bom " + "dia" seria "Bom dia.". Portanto, o resultado de "Alô " + args[1] é "Alô Maria", uma vez que args[1] denota o segundo elemento do array passado como argumento para o método main, ou seja "Maria".

Bem, agora você já sabe como escrever um programa que recebe dados de entrada (via parâmetros do programa), manipula esses dados (por meio da operação de concatenação de strings) e exibe o resultado na tela (por meio de uma chamada ao método println, como acima). Que tal então brincar um pouco?

Por exemplo, experimente escrever um programa que recebe diferentes argumentos, tais como "pão", "queijo", "maçã" etc. e imprime a mensagem "João comeu pão, queijo e maçã". Certamente, você terá outras idéias mais interessantes.

Quando terminar, siga para a próxima seção, para aprender um pouco sobre loops em Java – isto é, sobre comandos que especificam a execução de uma determinada ação, repetidas vezes.

Loops em Java

Ao escrever um programa, freqüentemente desejamos especificar que uma tarefa deve ser realizada repetidas vezes. Nesta seção, vamos discutir comandos da linguagem Java que podem ser usados para expressar essa repetição – comumente chamados, em computação, de loops.

Comandos de repetição

Considere uma tarefa que envolve a execução, repetidas vezes, de uma determinada ação. Por exemplo, suponha que você tem uma lista dos nomes dos jogadores, em um jogo, e deseja imprimir essa lista na tela. Suponha que os nomes dos jogadores estão armazenados em array de strings, referenciado pela variável 'jogadores'. Se o jogo tem 10 jogadores, seus nomes poderiam ser impressos da seguinte maneira:

```
System.out.println(jogadores[0]);
System.out.println(jogadores[1]);
...
System.out.println(jogadores[9]);
```

Note que omitimos 7 linhas do programa, indicando apenas '...' no lugar delas. É claro que escrever todas essas linhas, praticamente iguais, seria muito tedioso. Imagine se o jogo tivesse cem(!!) ou mil(!!!) jogadores. Felizmente, existem maneiras mais fáceis e concisas de expressar tal repetição em um programa.

Um loop básico...Um loop básico...um loop básico...

Existem diversos comandos para expressar um loop em Java, mas todos envolvem os seguintes passos:

1. Definir a porção de código cuja execução deve ser repetida.
2. Definir as condições em que a repetição deve iniciar.
3. Definir as condições de término da repetição.

Como seriam esses passos, no caso do nosso problema de imprimir os nomes dos jogadores?

```
1. AQUI ESTÁ O CÓDIGO A SER REPETIDO: IMPRIMIR O NOME E A PONTUAÇÃO DO JOGADOR.
2. QUEREMO QUE A REPETIÇÃO COMECE PELO PRIMEIRO JOGADOR.
3. QUEREMOS QUE A REPETIÇÃO TERMINE NO ÚLTIMO JOGADOR.
```

Examinemos, agora, em detalhes, uma das possíveis maneiras de escrever esse loop, em Java.

Comando while

O comando while é um comando de repetição que tem, em Java, a seguinte forma genérica:

```
// Definição das condições iniciais.
while (uma determinada condição é verdadeira) {
    // Execute o código escrito aqui.
}
```

O comentário que precede o comando while indica onde devem ser especificadas as condições de início da repetição. O comando while propriamente dito começa com a palavra reservada while, que é seguida por uma condição de terminação, especificada entre parênteses. Em seguida, vem o corpo do comando while – um bloco de comandos, delimitado ('{' e '}') (Vimos anteriormente, que o par de chaves ('{' e '}') é também

usado, em Java, para delimitar o corpo de um método, ou de uma classe). A execução do comando while consiste nos seguintes passos: antes de ser executado o corpo do comando, a condição de terminação é avaliada; se o resultado da avaliação for verdadeiro, ou seja, se a condição é satisfeita, o corpo do comando é executado, e esse processo se repete; senão (isto é, se o resultado da avaliação da condição de terminação for falso), então a execução do comando while termina.
Vamos agora aplicar, ao nosso exemplo de imprimir os nomes dos jogadores de um jogo:

```
// Comece com o primeiro nome da lista.
while (ainda existirem nomes a serem impressos) {
   // Imprima o próximo nome.
}
```

Hummm... É claro que o comando acima ainda não poderia ser executado pelo computador: falta escrever as instruções acima na sintaxe da linguagem Java. Isso significa que precisamos definir cada detalhe do nosso problema e "explicá-lo" ao computador explicitamente. Portanto, vamos procurar reformular nosso problema mais detalhadamente.

Considere a seguinte lista de 10 nomes, referenciada pela variável jogadores, de tipo String[] (ou seja, um array de strings):

```
String[] jogadores = {"Batman", "Robin", "Superman", "Homem Aranha", "Zorro",
                      "Tarzã", "Fantomas", "Capitão Marvel", "Mandrake","Hulck"};
```

Suponha, agora, que desejamos imprimir esses nomes na ordem em que ocorrem no array. Sem um comando de repetição, escreveríamos:

```
System.out.println(jogadores[0]);
System.out.println(jogadores[1]);
System.out.println(jogadores[2]);
System.out.println(jogadores[3]);
System.out.println(jogadores[4]);
System.out.println(jogadores[5]);
System.out.println(jogadores[6]);
System.out.println(jogadores[7]);
System.out.println(jogadores[8]);
System.out.println(jogadores[9]);
```

Note que nada muda de uma linha para outra desse código, exceto o número entre colchetes – isto é, o índice do jogador. Além disso, esse índice varia de uma maneira ordenada: começa com '0' e incrementa de um até chegar a '9'. Hummm... Parece que já temos informação suficiente para escrever nosso comando em Java. Já definimos as condições iniciais do loop (o índice deve começar com o valor '0'); as condições de término (o último nome impresso é o do jogador de índice '9'); e o comando a ser repetido (System.out.println(jogadores[um determinado número];).
Vamos então inserir essa informação em nosso "modelo" de comando while:

```
Comece com o jogador de índice 0.
while (o índice do jogador é menor que 10) {
    System.out.println(jogadores[índice do jogador]);
}
```

Repare na condição de terminação que escrevemos acima. Lembre-se que, no comando while, o corpo do comando é executado sempre que essa condição é satisfeita. Por isso, o correto é que o corpo do comando seja executado sempre que o índice for menor que 10, isto é, para os valores do índice de 0 a 9.

O que escrevemos acima já é quase um código Java... mas falta ainda alguma coisa! O código ainda não está correto, porque menciona o "índice do jogador", o qual não é precisamente definido. Bem, vamos então defini-lo.
Sabemos que o índice dos jogadores consiste de valores inteiros, variando de 0 a 9. Isso significa que devemos representar esse índice por uma variável armazena valores inteiros – em Java, uma variável de tipo int. Vamos então escrever isso no nosso código:

```
int numJogador = 0;     // Define a variável que representa o número do jogador
                        // e atribui a essa variável o valor inicial '0'.
while (o índice do jogador é diferente de 9) {
    System.out.println(jogadores[numJogador]);
}
```

Note que o nome de uma variável deve ser uma seqüência de caracteres, sem espaços em branco – por isso, não poderíamos, por exemplo, dar à nossa variável o nome 'número do jogador'. Se você preferir, poderá também escrever, como é usual, num_jogador (com o caractere underscore ('_') em lugar do espaço em branco).

Falta ainda transcrever para Java a nossa condição de terminação. Isso é simples: numJogador < 10, tal como em matemática. O operador '<' é o que chamamos de um operador relacional, isto é, que expressa uma relação entre dois valores (nesse caso, o valor da variável numJogador e 10).
Vamos então escrever isso no nosso código:

```
int numJogador = 0;     // Define a variável que representa o número do jogador
                        // e atribui a essa variável o valor inicial '0'.
while (numJogador < 10) {
    System.out.println(jogadores[numJogador]);
}
```

Ótimo – isso parece um código Java. Vamos ver se ele funciona. Comece fazendo uma cópia da sua aplicação console, dando o nome "ConsoleLoop" (ou algo parecido) ao projeto correspondente a essa cópia. Abra a classe ConsoleApp desse novo projeto, no editor de textos do BlueJ (você já sabe como fazer isso). Vamos agora modificar o corpo do método main, definido nessa classe. No corpo desse método, insira a sua lista de jogadores:

```
String[] jogadores = {"Batman", "Robin", "Superman", "Homem Aranha", "Zorro",
                      "Tarzã", "Fantomas", "Capitão Marvel", "Mandrake","Hulck"};
```

Em seguida, adicione o código do loop:

```
int numJogador = 0;     // Define a variável que representa o número do jogador
                        // e atribui a essa variável o valor inicial '0'.
while (numJogador < 10) {
```

```
    System.out.println(jogadores[numJogador]);
}
```

Ah! Adicione também uma última linha, apenas para sabermos que terminou a repetição do loop:

```
System.out.println("Terminou!");
```

Compile seu programa e corrija algum erro que você possa, eventualmente, ter cometido. Como sempre, se você tiver problemas, dê uma olhadinha no código 3 da seção de códigos, incluída no final deste tutorial.

Antes que você execute seu programa, vamos limpar a saída impressa pelas nossas aplicações anteriores. Para isso, primeiramente selecione, no menu, a opção 'Views' (Visões), e então selecione a opção 'Show Terminal' (Mostrar Terminal) e ative o checkbox:

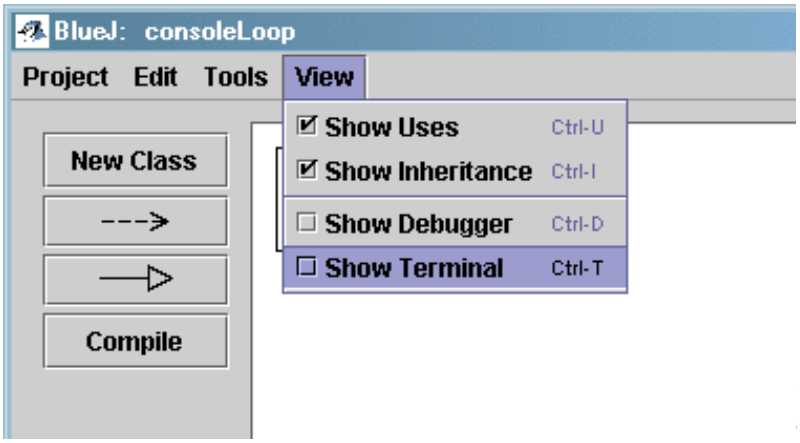


Figura 4 – Mostrando a janela do terminal.

(Opções) e, em seguida janela.

Agora, execute a aplicação ConsoleLoop (clicando com o botão direito do mouse sobre o ícone da classe 'ConsoleApp', e selecionando, em seguida, 'void main(args)', etc). Chiii.... Seu programa "está em loop"! – isto é, está executando sempre o mesmo comando, sem nunca parar! Você deve estar vendo algo como:

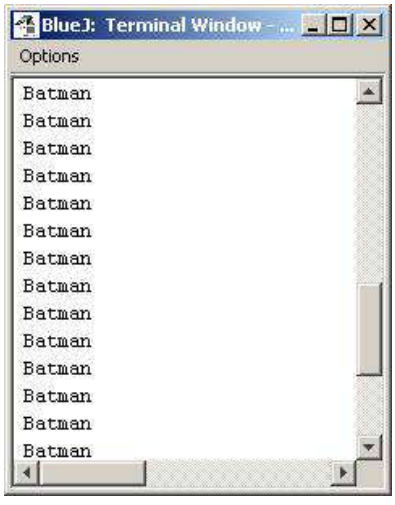


Figura 5 – Aplicação ConsoleLoop, primeira tentativa.

Para forçar o término da execução do programa, clique, com o botão direito do mouse, sobre a barra horizontal com listras vermelhas, que aparece no canto inferior esquerdo da janela do BlueJ e selecione a opção 'Reset Machine', para reiniciar o sistema de execução de programas Java.

Bem, agora vamos ver o que deu errado...

Depurando programas

Pois é... Acabamos escrevendo um programa que não funciona como esperávamos. Isso acontece com freqüência. Felizmente, existem ferramentas e técnicas para nos ajudam a lidar com esse tipo de problema. Nesta seção, você vai aprender a usar a mais importante dessas ferramentas – um depurador de programas.

O ambiente BlueJ provê um depurador de programas, bastante fácil de usar e que possibilita examinar, passo a passo, a execução do código do programa. Você verá como isso é útil para nos ajudar a descobrir erros em programas. Sem esse tipo de ferramenta, simplesmente teríamos que nos debruçar sobre o código, horas e horas, até, eventualmente, descobrir uma linha de código que originou um erro. Com o depurador de programa, é possível acompanhar a execução do código e ver exatamente o que está acontecendo de errado.

Conhecendo seu melhor amigo

Bem, o que é um depurador de programa, e como podemos usá-lo? Como já dissemos, ele é uma ferramenta que nos permite acompanhar, passo a passo, a execução dos comandos de um programa. No BlueJ, é muito fácil fazer isso.

Primeiramente, é necessário indicar ao BlueJ em que ponto do código desejamos começar a acompanhar a execução do programa. Como a nossa aplicação é bastante simples, essa escolha é fácil: vamos começar a acompanhar sua execução exatamente no ponto onde se inicia o loop.

Certifique-se de que a classe ConsoleApp foi previamente compilada e abra o código dessa aplicação no editor de texto do BlueJ. Observe a margem esquerda na janela do editor, mostrada na figura abaixo:

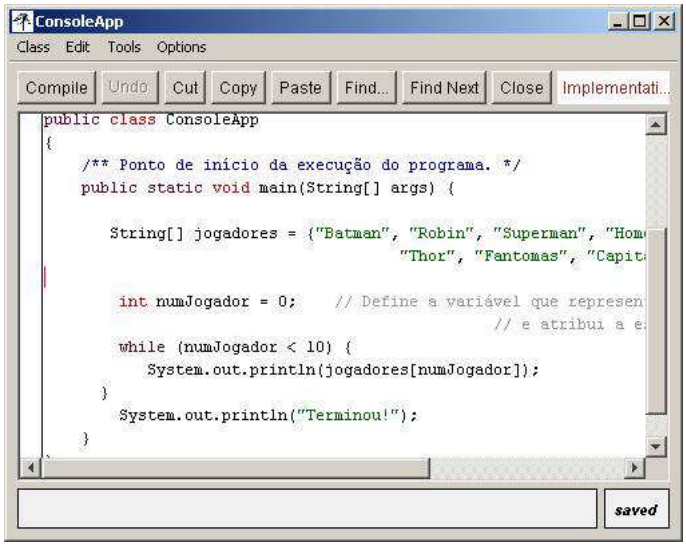


Figura 6 – margem de breakpoint.

Esta é a margem de breakpoint. Breakpoints indicam para o depurador onde parar a execução do programa, para que você possa examinar o que está acontecendo. Você pode especificar um breakpoint no código do programa simplesmente clicando sobre a margem de breakpoint, na posição correspondente ao comando em que a execução deve parar. Faça isso – clique sobre a margem de breakpoint, na linha do comando while. O BlueJ irá então colocar um pequeno sinal de parada na margem de breakpoint:

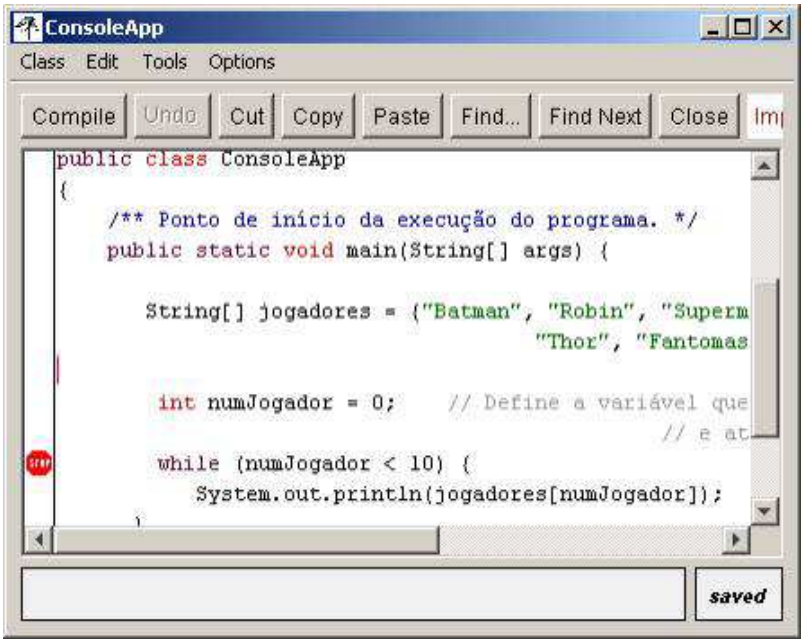


Figura 7 – Colocando um breakpoint.

Agora vá em frente, e execute a aplicação ConsoleLoop como anteriormente. A execução irá parar exatamente na linha do programa onde foi colocado breakpoint, antes de ser executado o comando correspondente. Quando isso ocorrer, sera exibida a janela do depurador, que mostra informação sobre o estado do programa, nesse ponto da execução:

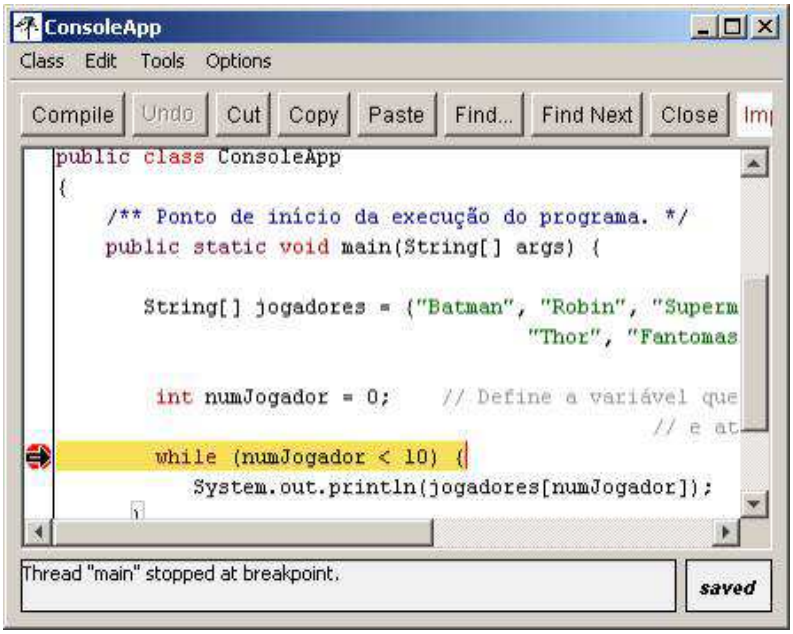


Figura 8 — BlueJ interrompe a execução no breakpoint.

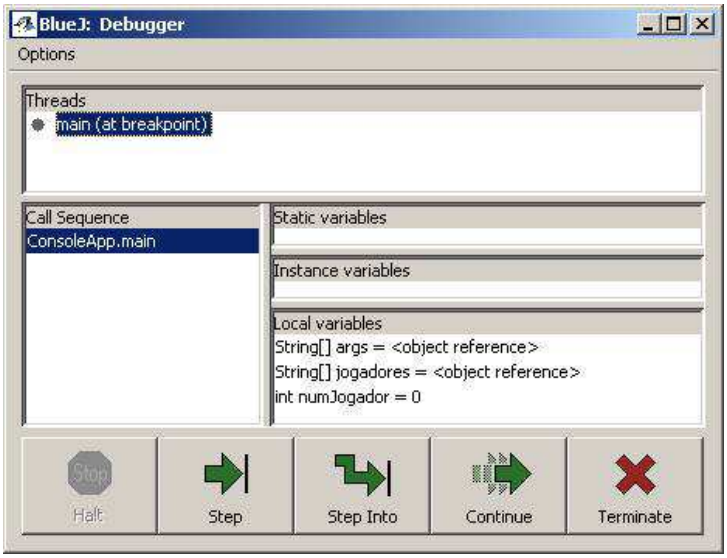


Figura 9 – a janela do depurador.

Para continuar mais um passo da execução do programa, selecione o botão step. Se desejássemos continuar a execução do programa normalmente, bastaria selecionar o botão continue. Para terminá-lo, bastaria selecionar 'terminate' (procure não usar essa opção – terminar o programa "à força" pode confundir o sistema de execução de programas Java, fazendo com que o comportamento do programa, em uma próxima execução, possa ser "estranho" (se isso acontecer, reinicie o sistema de execução de programas, clicando sobre a barra listrada, localizada no canto inferior direito da janela do BlueJ).

Ignore, por enquanto o botão step info. Já já, vamos explicar para que serve o misterioso botão halt (no momento, desabilitado).

Observe agora, na janela do depurador, o campo "Local Variables" (Variáveis Locais). Você verá que a variável numJogador armazena o valor '0'. Muito bem...vamos continuar.

Certifique-se de que o campo que contém main, na janela do depurador, está visível, e então pressione step novamente. Note que o nome do primeiro jogador – Batman - é impresso na janela do console. Pressione novamente step. Mais uma vez, é impresso o nome do primeiro jogador. Se você pressionar step mais uma vez, tudo igual irá se repetir...

Será que você já percebeu o que está acontecendo? Então observe novamente, no campo "Local Variables", o valor da variável numJogador – esse valor é sempre 0! Ele não é alterado durante a execução do corpo do comando while – por isso, a condição de terminação do loop é sempre verdadeira (uma vez que 0 é menor que 10), sendo sempre repetida a impressão do nome do primeiro jogador, sem nunca terminar. Está duvidando? Então pressione 'continue' e veja o que acontece.

E agora? O que fazer para parar a execução do programa? Bem, pressione o botão halt (parar) e, em seguida, pressione terminate. Ok, nós dissemos para você evitar isso, mas em situações desesperadoras, a solução tem que ser drástica.
Bem, como podemos, então, corrigir nosso programa? O que precisamos é incrementar o valor da variável numJogador, a cada vez que o nome do jogador é impresso, não é isso? Para isso, basta usar o seguinte comando:

```
numJogador = numJogador + 1;
```

Esse comando também poderia ser escrito, de forma mais abreviada, como:

```
numJogador += 1;
```

Embora sendo bastante simples, esse comando faz um bocado de coisas: primeiro, obtém o valor armazenado na variável numJogador; em seguida adiciona 1 a esse valor; finalmente, armazena o valor resultante de volta na área de memória referenciada pela variável numJogador.

Note como isso faz com que o valor de numJogador, inicialmente igual a 0, aumente, de 1 em 1, até 9. A variável numJogador faz o papel de um contador – conta o número de iterações, ou repetições, ou do loop. Ok, isso tudo é apenas jargão, mas é bom se acostumar com esses termos, que são usados muito comumente, em computação.
Voltando ao nosso problema... para corrigir o programa, basta inserir o comando acima depois do comando println:

```
int numJogador = 0;     // Define a variável que representa o número do jogador
                        // e atribui a essa variável o valor inicial '0'.
while (numJogador < 10) {
    System.out.println(jogadores[numJogador]);
   numJogador += 1;
}
```

Agora, compile novamente o seu programa. Remova o breakpoint. Limpe a tela do console e execute novamente a aplicação. Dessa vez, ao executar o programa, você deverá ver os nomes dos jogadores impressos na janela do console.

Chega então de programas simples e não muito úteis. Vamos aplicar um pouco do que já aprendemos para dar início ao nosso projeto principal – vamos começar nosso jogo de Batalha Naval!

Um pouco sobre interfaces gráficas

Agora que você já aprendeu um bocado, podemos começar a trabalhar no nosso projeto de construir um programa que implementa o jogo de Batalha Naval. Uma parte fundamental desse projeto é escrever o código que implementa o tabuleiro do jogo – esse código é fundamentalmente baseado em loops. Para concluir este módulo do tutorial, vamos então escrever um método cujo código descreve como traçar as linhas do tabuleiro do jogo.

Antes, mais uma informação importante.

Como já sabemos, o console Java apenas pode ser usado para exibir texto. O tabuleiro do nosso jogo, entretanto, é uma interface gráfica – consiste de uma grade de linhas e quadros coloridos. Portanto, não podemos usar, para o nosso jogo, o arcabouço da aplicação console com o qual vínhamos trabalhando até agora.

A solução seria usar Applets ou, alternativamente, o terceiro tipo de programa Java, que não abordamos ainda: aplicações gráficas stand-alone, isto é, aplicações que residem na própria máquina onde são executadas e fazem usos de janelas, figuras, menus,

botões etc. para interação com o usuário (lembre-se que a execução de um applet, ao contrário, é iniciada a partir de uma página da Internet). Vamos preferir, por simplicidade, escrever nosso jogo de Batalha Naval, não como um applet, mas como uma aplicação gráfica stand-alone – em inglês, Stand-alone Graphical Application (SGA). Mais tarde você poderá tentar escrever uma nova versão do jogo, como um applet, se desejar.

É claro que é possível criar uma SGA, no ambiente BlueJ, a partir do zero, e você deverá fazer isso, em algum módulo mais adiante. Por enquanto, para facilitar seu trabalho, faça o download de um modelo de aplicação SGA clicando aqui. Como o arquivo que você baixou está compactado (em formato .zip), você vai precisar usar o descompactador de arquivos do seu computador, para obter os arquivos do projeto original – descompacte o projeto original no subdiretório 'projects ' do diretório do BlueJ. Em seguida, inicie o BlueJ e selecione a opção 'open project' (abrir projeto) do menu 'Projects' da janela principal do BlueJ. Isso fará com que o projeto 'StandaloneApp' seja exibido na janela do navegador de projeto do BlueJ.

Você deverá ver nessa janela um ícone correspondente a um arquivo texto e outro ícone, correspondente à única classe do projeto – também denominada StandaloneApp. Compile e execute essa aplicação. Para executá-la, faça como anteriormente: clique com o botão direito do mouse sobre o ícone da classe StandaloneApp e escolha a opção 'void main(args)', etc. Se tudo correr bem, deverá aparecer, na tela do computador, uma nova janela, com fundo cinza claro e um 'x' azul desenhado ao longo das suas diagonais.

Pronto... Podemos começar!

Podemos agora começar a escrever o código do nosso jogo de Batalha Naval. Primeiro, vamos criar um novo projeto para o jogo. Depois, vamos procurar entender um pouco o código do modelo já existente, para poder então escrever o código que traça o tabuleiro na janela da aplicação. Finalmente, é isso o que vamos fazer.

Comece fazendo uma cópia do projeto StandaloneApp – chame esse novo projeto de Battleship (Batalha Naval). Compile e execute esse novo projeto, como anteriormente, apenas para ter certeza de que tudo está funcionando corretamente.

Agora, o código...

Agora estamos prontos para procurar entender o código existente. Clique duas vezes, seguidamente, sobre o ícone da classe StandaloneApp, para abrir o código dessa classe no editor do BlueJ. Embora o código dessa classe não seja muito longo, há muito que explicar sobre ele – mais do que seria possível em todo o espaço disponível neste tutorial. Assim, vamos nos ater aos aspectos mais importantes, o que permitirá que você aprenda o suficiente para continuar a implementação do jogo.
Primeiramente, procure pelo método main, nessa classe (use "ctrl-f" ou o botão 'Find...' na barra de ferramentas). Você deverá encontrar esse método próximo do início do código da classe. O método main deverá te parecer familiar: 'public', 'static', com um parâmetro do tipo String[] etc. Note, entretanto, que ele cria um novo objeto – da classe StandaloneApp – e inicia a variável app com uma referência esse objeto, por meio do comando:

```
app.setVisible(true);
```

Em seguida, realiza a chamada de método:

```
app.setVisible(true);
```

É isso que faz com que seja exibida a janela que vimos ao ser executado o programa. Observe as seguintes linhas de código, logo acima do método main:

```
private static final int APP_WIDTH = 300;
private static final int APP_HEIGHT = 200;
```

Elas declaram duas variáveis estáticas, de tipo int. Lembre-se que o atributo 'static' significa que o valor da variável é o mesmo para todos os objetos da classe Note também que essas variáveis são privadas (declaradas com o atributo 'private'), significando que apenas podem ser usadas no corpo da classe 'StandaloneApp'.

A palavra reservada 'final' significa simplesmente que o valor dessas variáveis não pode ser modificado ao longo da execução do programa. Isso significa que o valor de APP_WIDTH será sempre 300 e o valor de APP_HEIGHT, sempre 200. De fato, o que estamos fazendo é apenas dar um nome a cada um desses valores, para usar esses nomes, no código do programa, em lugar dos respectivos valores. Porque isso? A resposta é que essa prática torna o código do programa mais legível e mais fácil de ser modificado: basta modificar o valor da variável, em um único ponto do programa, ao invés de ter que modificar toda ocorrência do valor ao longo do código do programa. Variáveis declaradas com o atributo 'final' são também chamadas de constantes.

Se você sabe falar inglês, deve ter adivinhado que APP_WIDTH especifica a largura e APP_HEIGHT especifica a sua altura da janela da aplicação. Mas uma janela com essas dimensões seria muito pequena para a interface do nosso jogo. Por isso, vamos começar modificando esses valores – mude ambos para 400. Você deve estar se perguntando: "400? 400 o que?!"

Lembre-se que estamos lidando com uma janela gráfica: usualmente expressamos dimensões nesse tipo de janela em pixels. Se você olhar para o monitor, bem de perto, verá que a imagem é composta por pequenos pontos – cada ponto é o que chamamos de um pixel. Dependendo da resolução do monitor, podemos ter 640x480 pontos, 800x600, 1024x768, ou valores ainda maiores. Esse conceito de pixels será importante logo mais.

Depois de alterar o valor dessas constantes, compile e execute o programa novamente e observe o que mudou.

Agora, procure, no código da classe StandaloneApp, pelo método 'paint'. Se você ainda se lembra do primeiro módulo deste tutorial, saberá que o método 'paint' era o responsável por "pintar" a janela do applet. O mesmo ocorre nesse caso: o método 'paint' é que "pinta" a janela da nossa aplicação na tela do computador.
Observe o código do método 'paint' da classe StandaloneApp:

```
public void paint(Graphics gfx) {
    super.paint(gfx);

    Container workArea = this.getContentPane();
    Graphics workAreaGfx = workArea.getGraphics();

     workAreaGfx.setColor(Color.blue);
    workAreaGfx.drawLine(0, 0,
                  workArea.getWidth(),
                  workArea.getHeight());
```

```
    workAreaGfx.drawLine(workArea.getWidth()-1,
                0, 0,
                 workArea.getHeight());
}
```

Como sempre, um bocado de código que você não entende, ainda... Vamos ignorar a maior parte, nos atendo apenas ao que é importante para o nosso propósito. Apague todas as linhas desse código, exceto a segunda e a terceira linhas. Isso deve resultar no seguinte:

```
public void paint(Graphics gfx) {
    Container workArea = this.getContentPane();
    Graphics workAreaGfx = workArea.getGraphics();
}
```

O que isso significa? A primeira linha declara uma variável do tipo 'Container' chamada 'workArea' (em português, área de trabalho). Além disso, atribui a essa variável o resultado retornado pela chamada de método 'getContentPane()', que é dirigida a um determinado objeto, referenciado por 'this'. Vejamos uma coisa de cada vez...

Um 'Container' é um objeto Java que representa uma parte de uma interface gráfica. Mais especificamente, um 'Container' pode conter componentes como botões, campos de texto, listas de opções etc. Você deve lembrar-se que dissemos, anteriormente, que o ambiente de desenvolvimento de programas Java – Java SDK – provê uma coleção de bibliotecas de classes, já prontas, que podemos usar para facilitar nossa tarefa de escrever programas. Pois bem, a classe Container faz parte de uma dessas bibliotecas – denominada Swing – que reúne classes que implementam componentes de interfaces gráficas. Mais adiante, voltaremos a falar um pouco mais sobre essas bibliotecas de classes de Java.

A palavra reservada this é uma maneira especial de se fazer referência ao objeto corrente. Como sabemos, a execução de um programa Java consiste, essencialmente, de interações entre objetos criados pelo programa, por meio de chamadas de métodos, dirigidas a esses objetos. Em cada ponto da execução do programa, o objeto corrente é aquele ao qual foi dirigida a chamada de método que está sendo correntemente executada. Na situação acima, em que this ocorre no corpo do método paint, definido na classe StandaloneApp, a referência é ao objeto desta classe que está correntemente "pintando" sua janela na tela (ou seja, o objeto ao qual foi dirigida a chamada do método paint).

Muito bem... mas qual é o resultado de uma chamada do método 'getContentPane()'? Esse método retorna uma referência ao objeto que representa a porção da janela da interface gráfica da aplicação, compreendida entre as bordas e a barra de título da janela. Em outras palavras, 'content pane' é aquela parte da janela onde normalmente é exibido algum 'conteúdo'—tal como uma mensagem de texto, um desenho, uma figura etc.

Resumindo, essa linha do código:

1. Declara uma variável de nome workArea

2. Atribui a essa variável uma referência a um objeto que representa a área de conteúdo da janela da aplicação.

A linha seguinte é mais fácil:

```
Graphics workAreaGfx = workArea.getGraphics();
```

Essa linha declara uma variável do tipo Graphics, chamada workAreaGfx, e atribui a essa variável o resultado retornado pela chamada do método getGraphics(), dirigida ao objeto workArea. Lembre-se que um objeto da classe Graphics contém toda a informação sobre a configuração de características gráficas do computador, que um programa Java necessita para poder desenhar na tela do computador. Sempre que desenhamos qualquer coisa na tela, precisamos fazer isso usando um objeto da classe Graphics.

Portanto, essa segunda linha do código do método paint obtém o objeto da classe Graphics associado à área de conteúdo da janela corrente e faz a variável workAreaGfx referenciar esse objeto.

Preto no Branco

Agora – finalmente – podemos escrever nosso próprio código. Vamos começar especificando a cor de fundo da janela da interface gráfica do nosso jogo: preto!
Isso é fácil: fazemos isso usando o objeto Graphics apropriado e dirigindo a esse objeto uma chamada do método 'setColor' (definir cor):

```
workAreaGfx.setColor(Color.black);
```

Em seguida, temos que desenhar um retângulo pintado com essa cor de fundo, tendo as dimensões que a "parte de conteúdo" da janela. Ôpa! E como vamos saber qual é a dimensão dessa parte da janela? Você possivelmente estará tentado a dizer 400 x 400 pixels, mas isso não está totalmente correto. Essas são as dimensões de toda a janela, mas precisamos preencher apenas a área de conteúdo, que é um pouco menor.
Felizmente, a classe 'Container' provê métodos que nos permitem obter o tamanho dessa área de conteúdo da janela – os métodos getWidth() e getHeight(), definidos nessa classe, retornam a largura e a altura dessa área, respectivamente. Como esses dados provavelmente serão usados em vários pontos do programa, vamos declarar duas variáveis inteiras (Istoé, de tipo int), para armazenar esses valores. Isso é feito do seguinte modo:

```
int width = workArea.getWidth();
int height = workArea.getHeight();
```

Podemos agora desenhar nosso retângulo. Para isso, vamos usar o método fillRect, da classe Graphics. Esse método tem como parâmetros as coordenadas cartesianas (x e y) do canto superior esquerdo do retângulo e também a largura e a altura do mesmo. Ok, mas quais são as coordenadas do canto superior esquerdo do retângulo?
Bem, a coordenada '0' do eixo horizontal (eixo-x) corresponde à margem esquerda do retângulo; à medida que o valor de x se torna maior, nos movemos para a direita. De maneira análoga, a coordenada '0' do eixo vertical corresponde ao lado superior do retângulo; valores maiores de y correspondem a mover para baixo, na direção do lado inferior do retângulo. Portanto, o canto superior esquerdo do retângulo tem coordenadas (0,0). Fácil!

```
workAreaGraphics.fillRect(0, 0, width, height);
```

Juntando tudo isso, temos o seguinte código do método 'paint' da nossa classe

StandaloneApp:

```
public void paint(Graphics gfx) {
    // Obtém a area de conteúdo da janela.
    Container workArea = this.getContentPane();
    // Obtém o objeto gráfico associado.
    Graphics workAreaGfx = workArea.getGraphics();
    // Pinta o fundo da área de conteúdo com cor preta.
     workAreaGfx.setColor(Color.black);
    int width = workArea.getWidth();    // largura da area de conteúdo
    int height = workArea.getHeight();  // altura da area de conteúdo
    workAreaGfx.fillRect(0, 0, width, height);
}
```

Vá em frente: compile e execute esse novo código. Ao desenvolver um programa, é aconselhável compilá-lo e executá-lo sempre que alguma alteração for efetuada, verificando se a execução produz o resultado esperado. Quando fazemos um grande número de alterações em um programa, para e só então compilá-lo e executá-lo novamente, freqüentemente temos dificuldade em detectar onde pode estar o erro, caso algum erro ocorra (o que acontece, na maioria das vezes).

Traçando as linhas do tabuleiro.

Agora que já temos o fundo com a cor desejada, vamos traçar o tabuleiro do nosso jogo. Como você deve saber, o jogo de Batalha Naval tem dois tabuleiros – um para o "time vermelho" e outro para o "time azul". Vamos ensinar como traçar o tabuleiro vermelho. Depois, você poderá facilmente escrever sozinho, o código para traçar o tabuleiro azul.

Vamos supor que o tabuleiro do jogo tem dimensões 10 por 10. Para desenhar um tabuleiro com essas dimensões, teremos que traçar 11 linhas verticais e 11 linhas horizontais, como mostra a figura a seguir.

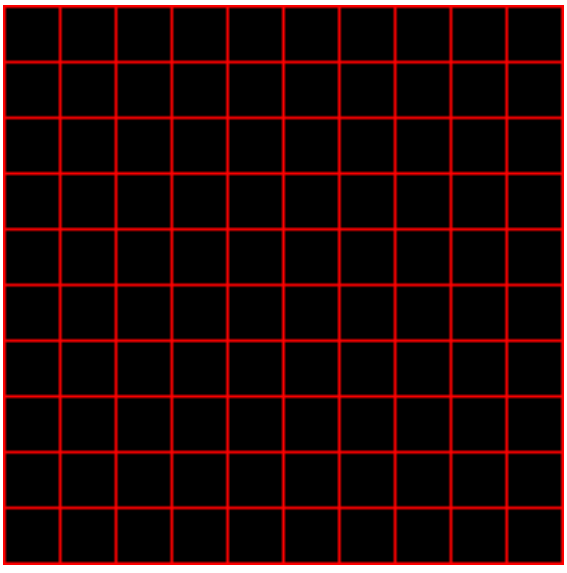


Figura 10 – O tabuleiro vermelho.

Como você já deve ter adivinhado, a maneira mais fácil de traçar essas linhas, é usando um comando de repetição. Antes de escrever esse comando, vamos primeiro saber como

traçar uma única linha.
Felizmente já existe um método para fazer isso: o método drawline, definido na classe Graphics. Uma chamada a esse método, na forma:

```
drawLine(x1, y1, x2, y2);
```

desenha uma linha que tem extremidades nas coordenadas (x1, y1) e (x2, y2). No nosso caso, quais devem ser esses valores? Lembre-se que o canto superior esquerdo do nosso tabuleiro tem coordenadas (0,0). Portanto, um comando na forma

```
drawLine(10, 50, 70, 50);
```

desenharia uma linha horizontal, começando 10 pixels à esquerda da margem esquerda e terminando à distância de 70 pixels da margem esquerda, posicionada à distância de 50 pixels do topo.

O que precisamos, então, é determinar onde queremos começar a traçar as linhas da grade e a distância entre elas. Vamos começar a traçar nossa grade no ponto (5,5), deixando uma margem no tabuleiro. Para a largura de cada linha e cada coluna, vamos usar 20 pixels (ou seja, separar as linhas da grade por 20 pixels, na direção x e na direção y).

Estamos prontos, agora, para escrever o trecho de código para traçar as linhas da grade. O que precisamos é de um comando de repetição para traçar as linhas horizontais e outro para traçar as linhas verticais. Para definir cada um desses comandos, precisamos determinar, em cada caso, quais são as condições iniciais, as condições de término e as operações a serem repetidas. Para facilitar, vamos proceder como no exemplo anterior, escrevendo essas condições e ações em português, como se fossemos usá-las no loop.

Comecemos pelas linhas verticais. A condição inicial é simples:

Comece com uma linha vertical a 5 pixels da margem esquerda.

A condição de término também é fácil:

Termine o loop depois de ter traçado 11 linhas.

A ação que queremos repetir também não é difícil:

Desenhe uma linha vertical, começando em y = 5 com comprimento ertical de 200 pixels, e então mova 20 pixels para a direita (na direção x). (Porque a linha teria comprimento de 200? Bem, ela cobre 10 linhas, cada uma com largura de 20 pixels).
Vamos precisar de um contador, para contra as linhas desenhadas (ou o número de repetições do loop). Vamos chamá-lo de lineCounter, e iniciá-lo com o valor 0.

```
int lineCounter = 0;
```

Também vamos precisar de variáveis para armazenar as coordenadas da posição inicial da nossa linha vertical:

```
int startX = 5;
int startY = 5;
```

Esses três comandos definem as condições iniciais do nosso loop. E as condições de término? Bom, queremos terminar o loop depois de ter traçado 11 linhas. Se incrementarmos lineCounter de um, cada vez que é traçada uma linha, devemos termi-

nar com lineCounter == 11. Em outras palavras, queremos que o comando seja repetido enquanto lineCounter < 11.
Se isso parece confuso, dê uma olhada no código abaixo e responda: "o que acontece quando o valor de lineCounter atinge 11?" Acompanhe a execução do código passo a passo, como se você fosse o depurador de programa, e você verá como ele funciona.

```
while (lineCounter < 11) {
    // Trace a linha e mova para frente.

    // Incremente o contador do loop.
    lineCounter += 1;
}
```

Falta ainda o código para traçar a linha e mover para frente. Traçar a linha não é muito difícil:

```
workAreaGfx.drawLine(startX, startY,startX, startY + 200);
```

Mover para frente é mais fácil ainda:

```
startX += 20;    // Mover para frente 20 pixels.
```

Vamos então colocar tudo isso junto:

```
// Código para desenhar as linhas da grade.
// Primeiro, definir a cor da linha.
workAreaGfx.setColor(Color.red);

// Desenhar as linhas verticais da grade.
// Condições iniciais do loop.
int lineCounter = 0;
int startX = 5;
int startY = 5;
// Usando um loop para traçar as linhas.
while (lineCounter < 11) {
    // Traçar a linha.
    workAreaGfx.drawLine(startX, startY,
                         startX, startY + 200);
    // Mover para frente 20 pixels.
    startX += 20;
    // Incrementar o contador do loop.
    lineCounter += 1;
}
```

Já temos um bocado de código novo. Compile e execute seu programa, para certificar-se de que ele funciona como vocÊ espera.
Agora, vamos traçar as linhas horizontais. Ao invés de escrever o código passo a passo, novamente, vamos apresentar a solução final e discutir em que ela difere do código anterior, para traçar as linhas verticais.

```
// Desenhar as linhas horizontais da grade.
// Condições iniciais do loop.
lineCounter = 0;
startX = 5;
startY = 5;
```

```
// Usando um loop para traçar as linhas.
while (lineCounter < 11) {
   // Traçar a linha.
    workAreaGfx.drawLine(startX, startY,
                  startX + 200, startY);
    // Mover para baixo 20 pixels.
   startY += 20;
    // Incrementar o contador do loop.
    lineCounter += 1;
}
```

Comecemos pelas condições iniciais do loop. Note que não é necessário declarar novamente as variáveis startX, and startY! Isso já foi feito, anteriormente, neste método. Se você declará-las de novo, isso será indicado como erro, durante a compilação do programa.

Agora observe a chamada do método drawLine. Note o que difere da chamada anterior. Agora, é a coordenada horizontal (x) que é acrescida de 200 pixels, na extremidade final da linha, em relação ao seu valor na extremidade inicial; a coordenada vertical (y) permanece a mesma – isso é exatamente o que queremos, no caso de uma linha horizontal. É isso! Não existem outras diferenças entre os dois códigos.

Compile e execute o programa, outra vez. Você deverá ver uma grade vermelha no canto superior esquerdo da janela. De fato, ela é grande demais – parece que não haverá espaço suficiente para outro tabuleiro do mesmo tamanho, para o time azul.

Agora é sua vez de programar.

Tente escrever o código para a grade azul. Ou você terá que aumentar o tamanho da janela da interface do jogo (lembre-se que APP_WIDTH e APP_HEIGHT determinam a largura e altura da janela, respectivamente), ou terá que diminuir o tamanho da grade vermelha (tente incrementar startX e startY de valores menores que 20). Mova as grades na janela – tente diferentes posições. Altere suas cores. Modifique o número de linhas da grade. Em resumo – agora é sua vez de brincar.

Antes de fazer isso, vale a pena conferir o que aprendemos neste módulo.

Conclusão

Uau – este módulo deu um bocado de trabalho!

Primeiro, aprendemos como passar dados de entrada para programas, por meio de parâmetros.

Depois nos concentramos nos aspectos fundamentais de ações que são descritas como repetições de ações mais simples. Aprendemos então como isso pode ser expresso em Java, por meio de um comando de repetição while.

Aproveitando um modelo básico de uma aplicação com interface gráfica, aprendemos a traçar linhas na janela da aplicação e, juntando tudo isso, começamos a programar parte do nosso jogo de Batalha Naval: o código que desenha o tabuleiro do jogo na janela. .

É provável que você tenha ainda muitas dúvidas sobre o vimos neste módulo. Isso não

tem importância. Você terá oportunidade de esclarecer essas dúvidas e aprender muito mais, até o final do jogo. O mais importante é: não se preocupe com detalhes – tente começar a programar, sem ter receio de "estragar" o código do programa que construímos até agora. Experimente. Divirta-se. Aproveite.

Nos vemos no próximo módulo.

APÊNDICE – Operadores em Java

Operadores são símbolos usados para indicar uma determinada operação. Alguns exemplos comuns são:

Operadores Aritméticos — Denotam operações aritméticas, tais como adição, subtração, multiplicação, divisão etc.

Operadores Lógicos — Denotam operações lógicas, usadas, por exemplo, para expressar condições de decisão mais complexas, pela combinação de condições mais simples, por meio de conceitos como 'e', 'ou', e 'não'.

Operadores relacionais — Denotam relações entre valores, tais como 'é menor que' ou 'é igual a'.

As tabelas a seguir relacionam os operadores, de cada tipo, usados mais comumente, exemplificando como eles podem ser usados em um programa. Note que essa não é uma lista completa dos operadores existentes em Java.

Alguns Operadores Aritméticos

operador	descrição	exemplo
+	Adiciona dois números	3 + 5
-	Subtrai dois números	x - 7
*	Multiplica dois números	velocidade * tempo
/	Divide dois números	3.75 / 1.92

Alguns Operadores Relacionados

operador	descrição	exemplo
==	É igual a	if (x == 3)
!=	É diferente de	while (teste != false)
>	É maior que	if (4 > cat.numPatas())
<	É menor que	do {} until (countador < 7)
>=	É maior ou igual a	while (c >= 9)
<=	É menor ou igual a	if (x <= 4.5)

Observação: Note que é usado o símbolo '==' para o operador de comparação de igualdade, diferentemente do usual em matemática (onde é usado o símbolo '=').

Alguns Operadores Lógicos

OPERADOR	DESCRIÇÃO	EXEMPLO
&&	'E'	IF ((X<3) && (Y>7))
||	'OU'	IF ((X<1.5) || (G<1))
!	'NÃO'	WHILE (!PARELOOP)

Módulo 4

Introdução a Interfaces Gráficas

Introdução

Neste módulo, vamos cobrir vários tópicos importantes:

1. Como criar uma aplicação com interface gráfica a partir do zero.
2. O conceito de herança em linguagens orientadas a objetos.
3. Como organizar o código do programa em classes.
4. O conceito de programação dirigida por eventos.

Em cada caso, vamos aplicar o tópico na programação do nosso Jogo de Batalha Naval. Ao terminar este modulo, você terá desenvolvido o código para exibir a janela de abertura do jogo – uma janela simples, como o nome do jogo e a mensagem "clique o mouse para continuar".

No módulo seguinte, você também irá aprender como tratar o evento de o usuário clicar o mouse sobre essa janela: o que desejamos é que o programa então desenhe na janela o tabuleiro do jogo e deixe tudo pronto para que o jogo possa começar.

E aí, pronto para começar a programar? Então, siga em frente!

Criando uma interface gráfica e aprendendo sobre herança

No módulo anterior, utilizamos um modelo padrão de uma aplicação gráfica (SAG – Standalone Graphical Application), para criar, a partir desse modelo, o tabuleiro da interface do nosso jogo de Batalha Naval. Neste módulo, veremos como criar uma aplicação gráfica a partir do zero, para aprendermos sobre o modelo de programação de interfaces gráficas e sobre as bibliotecas de classes, disponíveis em Java, para facilitar a programação desse tipo de aplicação.

De modo geral, o processo de criar uma aplicação gráfica é semelhante ao de criar uma aplicação console, o que foi visto no módulo 2. Entretanto, existem alguns conceitos noves envolvidos, e um bocado de novas classes Java que você terá que conhecer. Vamos começar criando um novo projeto, no BlueJ, e instruindo, passo a passo, como programar parte do código desse tipo de aplicação, para depois explicar o código do programa, como temos feito até agora.

Inicie a execução do BlueJ e selecione a opção New Project (Projeto Novo) do menu Project (Projeto). Digite o nome "BattleshipSAG", quando for solicitado o nome do novo projeto (você já sabe que Batttleship quer dizer Batalha Naval), e clique então sobre o botão Create (Criar).

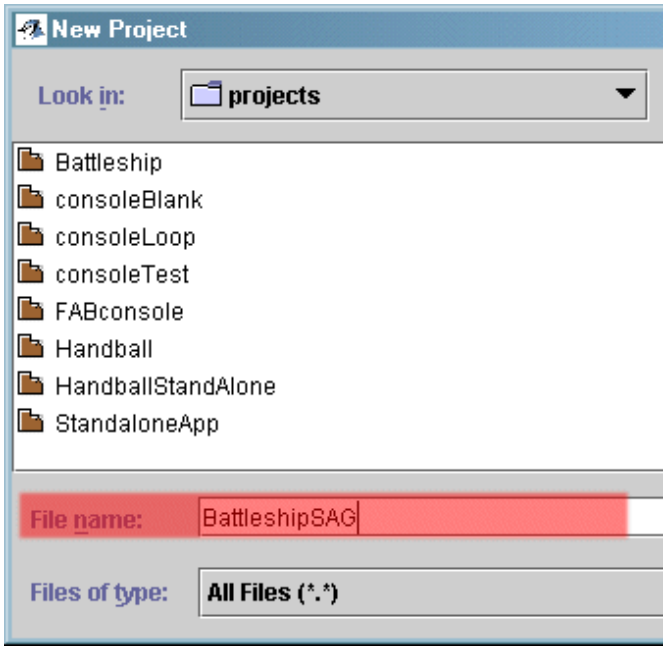


Figura 1— Criando uma aplicação SAG a partir do zero.

Quando o novo projeto for exibido na janela do navegador de projeto do BlueJ, selecione a opção New Class (Nova Classe) e digite "BattleshipApp" para o nome da nova classe.

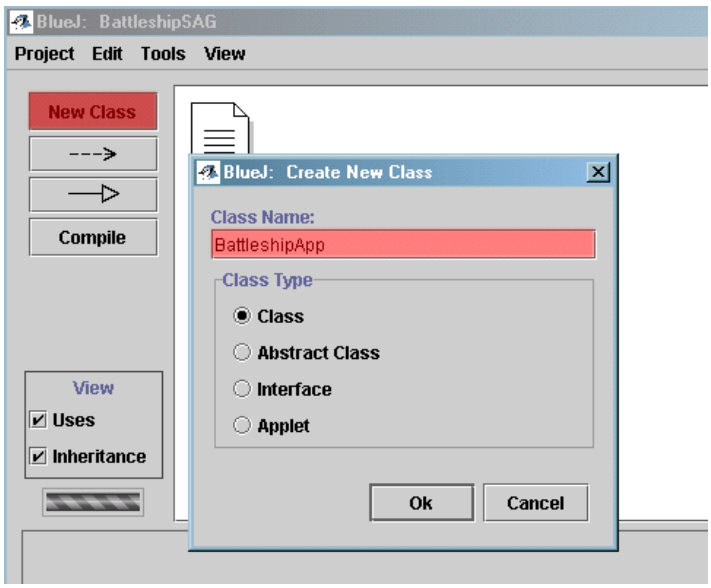


Figura 2— Criando a classe BattleshipApp.

O ícone correspondente à sua nova classe irá aparecer na janela do navegador de projeto (note que ele tem hachuras, indicando que o código da classe deve ser compilado, antes de ser executado). Clique sobre esse ícone duas vezes, seguidamente, de maneira a abrir o código da classe no editor de textos do BlueJ. E observe o código exibido. Elimine desse código os comando e comentários que não serão necessários, de maneira a obter o seguinte código:

```
/**
* Write a description of class BattleshipApp here.
*
* @author (your name)
* @version (a version number or a date)
```

```
*/
public class BattleshipApp
{
    // instance variables

    /**
     * Constructor for objects of class BattleshipApp
    */
     public BattleshipApp()
    {
    }
}
```

Substitua os comentários acima, gerados automaticamente pelo BlueJ, por suas respectivas traduções em português, ou seja:

```
/**
* Descricão da classe BattleshipApp.
*
* @author (seu nome aqui)
* @version (número de versão ou data)
*/
public class BattleshipApp
{
    // variáveis de instância

    /**
     * Construtor de objetos da classe BattleshipApp
    */
     public BattleshipApp()
    {
    }
}
```

Altere o comentário do cabeçalho do programa, de maneira a indicar o seu nome e a versão (v.01) ou a data, nos locais indicados. Além disso, substitua a linha onde está escrito Descricão da classe BattleshipApp por uma descrição dessa classe. Agora, assim como no caso da aplicação console, vamos ter que adicionar, ao corpo da classe, a definição do método main – o método que contém o código a partir do qual será iniciada a execução do programa, como você já sabe. Insira, então, o seguinte código, logo antes da última chave, que fecha o corpo da classeo, o ficaria entomo voceradicionar, ao c BlueJ, por suas respectivas traduJ, selecione a ope teremos que conhecer:

```
/**
* Ponto de início da execução do programa.
*/
public static void main(String[] args) {
}
```

Compile o código da classe, clicando sobre o botão 'compile' (compilar), para certificar-se de que tudo está OK. Uma vez que a classe tenha sido compilada corretamente, podemos começar a adicionar código que irá transformá-la em na aplicação SAG que desejamos.

Mas, antes, um pouco de teoria.

Janela para o Mundo

Uma interface gráfica é baseada em janelas que possuem botões, menus etc, recursos para modificar o tamanho da janela e reagem a comandos do usuário, como clicar o mouse sobre um determinado botão ou digitar uma seqüência de caracteres no teclado. Devido a essa funcionalidade extra, para que programas com interface gráfica possam ser executados, o sistema de execução de programas Java precisa conhecer mais informações sobre o programa, do que no caso de uma aplicação console, bem mais simples, como vimos anteriormente. Em termos gerais, essa é a relação das informações iniciais que devem ser fornecidas pelo programa:

1. Primeiro, deve informar que é um programa com interface baseada em janela.

2. Segundo, é necessário informar qual deve ser o tamanho da janela, quando a execução do programa é iniciada.

3. Terceiro, deve incluir um comando para exibir a janela na tela do computador.

Depois de adicionar essas informações à nossa classe BattleshipApp, deveremos poder compilar e executar essa classe, observando que a janela da aplicação será exibida na tela do computador.

Herança – Estendendo uma classe para programar a janela da aplicação

Satisfazer o primeiro requerimento acima é bastante simples – simplesmente, precisamos dizer "Ei,, Java—coloque esse programa em uma janela." Para fazer isso, simplesmente modifique a linha de código

```
public class BattleshipApp
```

to

```
public class BattleshipApp extends JFrame
```

Isso é muito fácil de escrever, mas o que significa?

Para responder a essa pergunta, precisamos falar sobre um conceito muito importante da programação orientada a objetos – o conceito de herança. Assim como a relação de 'herança' entre pais e filhos, no mundo real, em programação orientada a objetos, herança indica uma relação semelhante entre objetos pai e objetos filho. Por exemplo, na vida real, podemos dizer que "esse menino herdou os olhos do pai", no sentido de que seus olhos se parecem muito com os de seu pai. Em Java, podemos dizer que "BattleshipApp herda seu comportamento de janela da classe JFrame", significando que BattleshipApp se comporta como JFrame, no que se refere à aparência e propriedades da sua interface exibida na tela do computador. Se uma classe B herda de uma classe A, dizemos também que B é uma subclasse de A, ou A é uma superclasse de B. Você vais saber outros aspectos importantes sobre essa relação de subclasses em Java, mais adiante.

Muito bem. Mas o que é JFrame? Falando de maneira simples, é uma classe pré-definida em Java, que implementa as funcionalidades básicas de uma janela – isto é, define objetos que representam uma janela na tela do computador, que possui uma barra de título, um botão para fechar a janela, botões para aumentar ou diminuir o tamanho da janela, assim como o funcionamento desses botões. Portanto, quando dizemos que BattleshipApp herda de JFrame, o que realmente queremos dizer é que a janela do nosso jogo

terá um título, um botão para fechar a janela, botões de controle do tamanho da janela etc.

Para obter todas essas características, basta adicionar duas palavras simples ao cabeçalho da declaração da nossa classe BatleshipApp: extends JFrame. Você pode imaginar que 'extends' significa 'herda de'. Portanto, as palavras acima indicam que a classe BattleshipApp herda da classe JFrame. Isso significa que todas as variáveis e métodos definidas na classe JFrame são visíveis na classe Battleship, podendo ser usados no código dessa classe. Note que, portanto, uma grande vantagem do mecanismo de herança em linguagens orientadas a objetos é a possibilidade de reuso de código já existente, sem a necessidade de se ter que copiar todo esse código no programa, novamente.

Faça essa modificação no código da classe BattleshipApp e então compile novamente o código da classe. Você vai obter um erro de compilação. Não se preocupe com esse erro, por enquanto. Já vamos explicar qual é a razão do erro e como corrigi-lo.

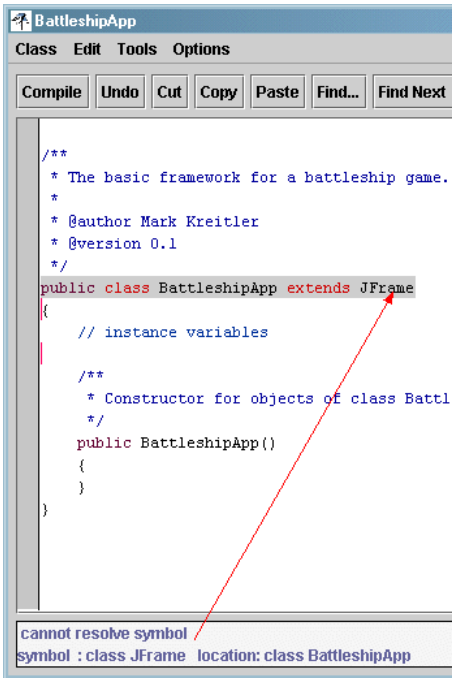


Figura 3— Problemas na compilação da classe BattleshipApp.

A linha em evidência indica onde ocorreu o erro, e a janela de estado, nap arte inferior da janela do editor de texto, descreve o erro ocorrido. Esse erro indica que o compilador desconhece a palavra JFrame, isto é, ele não consegue encontrar, em seu programa, a declaração de uma classe cujo nome é JFrame. Isso não é um grande problema. Como dissemos anteriormente, a classe JFrame é pré-definida em Java. Em outras palavras, essa classe é declarada em uma das bibliotecas de classes que fazem parte do ambiente de desenvolvimento e programas da linguagem – JDK (Java Development Toolkit).

Existe um grande conjunto de classes pré-definidas em Java, com o objetivo de facilitar a tarefa de desenvolver programas na linguagem. Essas classes estão organizadas em bibliotecas de classes, de acordo com as funcionalidades que implementa. Por exemplo:

§ A biblioteca java.Math agrupa classes que definem métodos que implementam operações matemáticas comumente usadas em programas, tais como, elevar um número a um determinado expoente, extrair a raiz quadrada de um número, calcular o seno, cosseno ou tangente de um ângulo etc.

§ A bliblioteca java.lang agrupa classes que implementam funcionalidades fundamen-

tais da linguagem, tais como operações sobre seqüências de caracteres (ou Strings), operações de comparação de igualdade entre objetos, de cópia de objetos etc.

§ A biblioteca java.io agrupa classes que implementam operações de entrada e saída de dados, a partir da tela, do teclado, de arquivos, de portas de comunicação etc.
As bibliotecas java.awt e javax.swing contêm um conjunto de definições de classes, que descrevem componentes de interfaces gráficas e suas funcionalidades: janelas, botões, menus, campos para entrada de texto etc. A classe JFrame é definida na biblioteca javax.swing. Para poder utilizar classes definidas em uma dessas bibliotecas, precisamos incluir, no início do programa, uma cláusula que indica o nome dessa biblioteca. Por exemplo, para utilizar classes da biblioteca javax.swing, em nossa classe BattleshipApp, precisamos incluir, no início do arquivo dessa classe, a seguinte cláusula de importação:

```
import javax.swing.*;
```

O símbolo '*' indica que estamos importando (isto é, tornando visíveis, para uso no código da nossa classe) todas as declarações de classes contidas na biblioteca "javax.swing". Se quiséssemos importar apenas a classe JFrame, poderíamos escrever

```
import javax.swing.JFrame;
```

As definições de classes da biblioteca javax.swing são baseadas em classes definidas em outra biblioteca – java.awt, a qual constitui a base para a implementação de interfaces gráficas (awt é uma abreviação de Abstract Window Tollkit, isto é, ferramentas para implementação de uma janela abstrata). Por esse motivo, além da cláusula de importação acima, precisaremos também incluir, no início da nossa classe BattleshipApp, a seguinte cláusula de importação:

```
import java.awt.*;
```

Depois de incluir, no início do arquivo, essas duas cláusulas de importação, seu código deverá ser semelhante ao seguinte:

```
/**
* Arcabouço básico do jogo de Batalha Naval.
*
* @author (seu nome aqui)
* @version 0.1
*/
import java.awt.*;
import javax.swing.*;

public class BattleshipApp extends JFrame
{
   // variáveis de instância

   /**
    * Ponto de início da execução do programa.
   */
    public static void main(String[] args) {
   }

   /**
    * Construtor de objetos da classe BattleshipApp
```

```
    */
     public BattleshipApp()
    {
    }
}
```

Tente compilar o código da classe novamente. Observe que, dessa vez, tudo irá correr bem: o erro que havíamos obtido anteriormente foi eliminado.

O TAMANHO DA JANELA

Agora que já informamos, em nosso programa, que nossa aplicação é baseada em janelas, precisamos informar o tamanho da janela e indicar que ela deve ser exibida na tela do computador. Em princípio, isso é fácil – simplesmente precisamos incluir, em nosso programa, uma chamada ao método setSize (ou define tamanho), o qual é herdado pela nossa classe BattleshipApp, da sua superclasse JFrame. De fato, o método setSize também é herdado pela classe JFrame, da sua superclasse Component, na qual o método é definido.

```
this.setSize(600, 500);
```

Fazendo isso, indicamos que a nossa janela tem largura de 600 pixels e altura de 500 pixels. Mas... onde essa linha de código deve ser incluída? Bem, queremos que esse comando seja executado quando a janela do nosso programa é criada. Quando isso acontece?

Essa é uma questão importante. De fato, temos que indicar, explicitamente, no nosso programa, que um objeto da classe BattleshipApp deve ser criado. Se não fizermos isso, nunca existirá um objeto, que representa essa janela, ao qual poderá ser dirigida a chamada do método 'setSize'. Experimente executar o seu programa (clicando duas vezes, seguidamente, sobre o ícone de 'BattleshipApp' no navegador de projeto, e então selecionando a opção 'main(args[])'. Você verá que não aparecerá nenhuma janela na tela do computador, uma vez que o código do programa não inclui um comando para criar essa janela.
Portanto, a primeira coisa que precisamos fazer é criar um objeto da classe BattleshipApp, ou, em outras palavras, criar uma instância dessa classe. Mas onde devemos incluir o código para fazer isso? Lembre-se que Java inicia a execução do programa pelo método main. Você já deve ter adivinhado que é no corpo desse método que devemos incluir o comando para criar nossa janela – você está certo! O código requerido é bastante simples – adicione, no início do corpo do método main, o seguinte comando:

```
new BattleshipApp();
```

A execução desse comando cria uma instância da classe BattleshipApp, que é um objeto que representa uma janela (uma vez que a classe BattleshipApp herda da classe JFrame)

Agora, temos que definir o tamanho da janela. Você pode estar tentado a fazer isso imediatamente depois do comando 'new BattleshipApp()'. Isso seria lógico, mas existe um local melhor: no construtor da classe BattleshipApp. Para entender porque, vamos gastar alguns minutos explicando sobre esse tipo de método especial, e importante, existente em toda classe.

Construtores de objetos

Você provavelmente deve ter notado que, quando o BlueJ cria automaticamente o esqueleto do código de uma classe, ele sempre inclui um método, cujo nome é igual ao dessa classe. Por exemplo, na classe BattleshipApp, existe um método público chamado BattleshipApp(). Como dissemos anteriormente, Java reconhece um método cujo nome é igual ao da classe como um construtor de objetos dessa classe. Quando queremos criar um objeto de uma determinada classe, digamos C, devemos usar o comando new, seguido de um construtor de objetos dessa classe – tal como em new C(); .Quando esse comendo é executado, o construtor C() é chamado pelo sistema de execução de programas Java, imediatamente após o objeto ser criado. Isso é bastante útil, porque podemos inserir, no corpo do construtor, o código necessário para estabelecer o estado inicial do objeto, atribuindo valores às suas variáveis de instância – o que é usualmente chamado de "inicialização" do objeto.
Por exemplo, considere uma classe simples, Círculo, que descreve objetos que representam círculos:

```
public class Círculo {
    public float raio;
    public float circunferência;
    // Construtor de objetos da classe Círculo.
    public Círculo(float r) {
       raio = r;
        circunferência = 2 * 3.14159 * r;
   }
}
```

Observe que classe Círculo declara duas variáveis de instância – raio e circunferência. Podemos criar um objeto dessa classe por meio do seguinte comando:

```
Circle c1 = new Circle(3);
```

Quando esse comando é executado, primeiramente é criado um objeto da classe Círculo (mais precisamente, é alocada uma área da memória do computador para cada uma das variáveis de instância definidas na classe) e, em seguida, é executado o código do construtor da classe Círculo, passando, como argumento para esse construtor, o valor 3 (o que determina, então, o valor de 'r', no corpo desse métodod) . Observe a definição do construtor da classe Círculo. No corpo desse método, o valor de 'r' é atribuído à variável 'raio' e o valor da variável 'circunferência' é calculado, por meio da avaliação da expressão 2 * 3.14159 * r. Portanto, depois e executado o comando acima, se tentarmos obter o valor do raio e da circunferência do objeto criado, por meio das chamadas de métodos c1.raio e c1.circunferência, obteríamos os valores '3.0' e '18.9' (igual a 2 * 3.14159 * 3.0), respectivamente.
Observe, no código da classe Círculo, que não é declarado o tipo do valor retornado pelo construtor definido na classe, o que não é necessário, no caso de construtores, como já dissemos anteriormente. Em outras palavras, a maioria das definições de métodos tem a forma:

```
tipo nomeDoMétodo (tipo1 arg1, tipo2 arg2, ...) {
    // Corpo do Método aqui.
}
```

onde tipo indica o tipo do valor retornado pelo método (tal como int, float, etc). Construtores, por outro lado, têm a forma:

```
nomeDaClasse (tipo1arg1, tipo2 arg2, ...) {
    // Corpo do Construtor aqui.
}
```

Não é necessário especificar o tipo do valor retornado por um construtor, porque Java sabe que um construtor sempre produz, como resultado, uma referência a um novo objeto da classe correspondente.

Voltando ao nosso código

Voltando ao nosso problema, devemos incluir a chamada ao método setSize no corpo do construtor da classe BattleshipApp, uma vez que o corpo desse método será executado sempre que for criada uma nova instância de BattleshipApp. Esse é exatamente o comportamento que desejamos. Portanto, vamos tentar o seguinte código:

```
/**
* Constructor for objects of class BattleshipApp
*/
public BattleshipApp()
{
    this.setSize(600, 500);
}
```

Aqui, a palavra reservada this é opcional: ela indica que a chamada do método setSize é direcionada a 'esta' instância da classe BattleshipApp, que está sendo correntemente criada pela execução do corpo do construtor. Caso a palavra this seja omitida, ou seja, se a chamada for feita simplesmente na forma setSize(600, 500); o efeito será o mesmo.

Compile e execute o seu código, após ter modificado o construtor BattleshipApp(), como indicado acima. Se você obtiver algum erro durante a compilação, confira o seu código com o código 4, apresentado na seção de códigos incluída no final deste tutorial.

Quando você executa essa versão do programa, ainda não acontece nada. O que fizemos foi apenas criar uma instância da classe BattleshipApp, que representa uma janela, uma vez que essa classe herda de JFrame, mas ainda não introduzimos nenhum comando que resulte em exibir essa janela na tela do computador. Precisamos, então, dizer, explicitamente, que essa janela deve ser exibida.

Exibindo a janela

Felizmente, comandar isso é muito fácil. Basta adicionar a seguinte linha de código ao construtor da classe BattleshipApp (imediatamente depois da chamada do método setSize):

```
this.setVisible(true);
```

Assim como o método setSize, o método setVisible também é definido na classe Component, sendo herdado dessa classe, pela nossa classe BattleshipApp, via sua superclasse JFrame. O método setVisible pode ser usado, tanto para exibir como para

ocultar uma janela, dependendo do valor que é passado como argumento na chamada ao método: o argumento true indica que a janela deve ser exibida, e false indica que ela deve ser ocultada. Se você não sabe o que significam os valores true e false, reveja a descrição dos valores do tipo boolean – um dos tipos primitivos da linguagem Java – no apêndice do módulo 2.

Compile e execute novamente o seu programa. Finalmente, uma janela!

Agora que temos o arcabouço básico do nosso jogo de Batalha Naval, podemos começar a projetar o restante do nosso projeto. Aqui começa a parte mais interessante. Vamos prosseguir aprendendo a estruturar os nosso projeto em classes.

Estruturando o código em classes

Agora que temos o arcabouço básico de uma aplicação gráfica, precisamos adicionar ao projeto novas classes, que de fato implementam o nosso jogo de Batalha Naval. Nessa seção, vamos considerar uma versão de "mundo real" do jogo de Batalha Naval, para depois tentar converter suas partes em classes Java. O que vamos fazer é escrever "cascas" vazias para as nossas classes e adicioná-las ao projeto, especificando, assim, a interface de comunicação entre as diversas classes do projeto.

Parte por parte

Imagine o jogo de Batalha Naval. Sabemos exatamente quais são os objetos físicos que compõem o jogo:

Primeiro, precisamos de dois tabuleiros:

1. Um tabuleiro vermelho.
2. Um tabuleiro azul.

Além disso, cada tabuleiro contém:

1. 5 navios.
2. Um conjunto de pinos brancos.
3. Um conjunto de pinos vermelhos.

Também precisamos de dois jogadores:

1. O jogador "aliado".
2. O jogador "inimigo".

Isso pode ser resumido no seguinte diagrama:

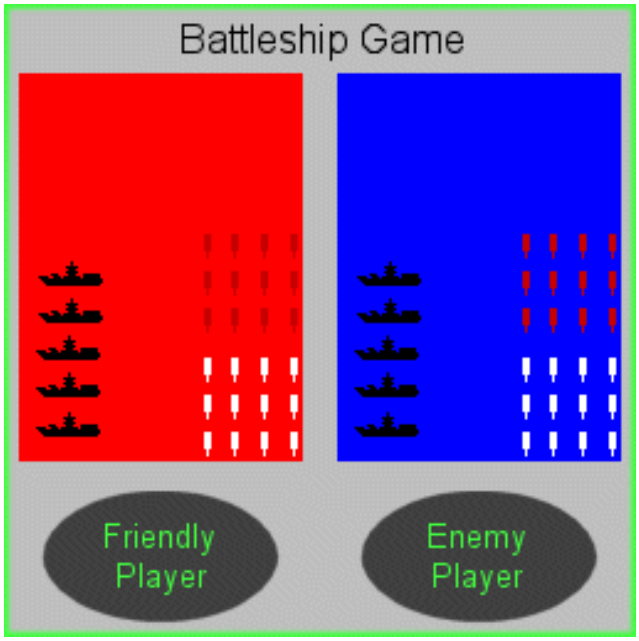


Figura 4— Objetos do jogo de Batalha Naval

Vamos supor que queremos criar uma classe Java para representar cada um desses tipos de objetos do "mundo real". Nesse caso, teríamos a seguinte lista de classes para o nosso projeto:

§ Classe dos objetos que representam os pinos vermelhos (vários, para cada tabuleiro).

§ Classe dos objetos que representam os pinos brancos (vários, para cada tabuleiro).

§ Classe dos objetos que representam os navios (vários, para cada tabuleiro).

§ Classe do objeto que representa o tabuleiro vermelho.

§ Classe do objeto que representa o tabuleiro vermelho.

§ Classe do objeto que representa o jogador aliado.

§ Classe do objeto que representa o jogador inimigo

§ Classe do objeto que representa o jogo como um todo.

Agora, o que precisamos fazer é criar algum código correspondente a cada um dos itens dessa lista.

Abstração do mundo concreto

Estamos agora prontos para passar do mundo concreto do jogo de Batalha Naval para o mundo abstrato do código Java que representa esse jogo. Para conectar esses dois mundos, o que precisamos é de um velho conhecido: a definição de classe.

Lembre-se de como é uma definição de classe, dos exemplos que vimos anteriormente. De modo geral, uma definição de classe tem a seguinte forma:
class nomeDaClasse

```
{
   type nomeDaVariável1;
   type nomeDaVariável2;
  ...
```

```
    type nomeDaVariávelN;
    type método1(type arg1, type arg2, ...) {
       // Corpo do método entra aqui.
    }
    type método2(type arg1, type arg2, ...) {
       // Corpo do método entra aqui.
    }
    ...
    type métodoM(type arg1, type arg2, ...) {
       // Corpo do método entra aqui.
    }
}
```

Imagine que vamos organizar as classes do no nosso jogo exatamente como o diagrama da figura acima. Vamos começar pelas classes dos objetos mais simples: aqueles que não contêm nenhum outro objeto. Existem cinco deles: pinos vermelhos (RedPeg), pinos brancos (WhitePeg), navios (Ship) e os dois jogadores (FriendPlayer e EnemyPlayer).
Portanto, nosso código poderia começar assim:

```
class RedPeg {
    // Dados de pinos vermelhos entram aqui.
    // Métodos de pinos vermelhos entram aqui.
}
class WhitePeg {
    // Dados de pinos brancos entram aqui.
    // Métodos de pinos brancos entram aqui.
}
class Ship {
    // Dados de navios entram aqui.
    // Métodos de navios entram aqui.
}
class FriendlyPlayer {
    // Dados do jogador aliado entram aqui.
    // Métodos adicionais do jogador aliado entram aqui.
}
class EnemyPlayer {
    // Dados do jogador inimigo entram aqui.
    // Métodos adicionais do jogador inimigo entram aqui.
}
```

Muito bem – criamos cinco "cascas" vazias para as nossas classes, que eventualmente irão conter as definições de variáveis de instância e métodos requeridos. Por enquanto, vamos deixar essas classes desse modo e considerar as demais classes do jogo, mais complexas.
Primeiro, temos os tabuleiros (representados pela classe Board, a seguir). Tabuleiros contêm pinos e navios e, portanto, a classe correspondente irá incluir esses objetos como dados. Os nomes das variáveis usadas para armazenar esses dados foram escolhidos, em inglês, de acordo com o que cada um desses objetos representa: pinos vermelhos representam acertos – hitMarkers – e pinos brancos representam erros – missMarkers; o conjunto de navios é uma esquadra – fleet:

```
class Board {
    private RedPeg[]    hitMarkers;
    private WhitePeg[]  missMarkers;
    private Ship[]      fleet;
    // Additional board data goes here.
    // Métodos da classe Board entram aqui.
}
```

Não se esqueça que uso dos símbolos '[]' em seguida a um nome de tipo indica um Array de elementos desse tipo, ou seja, uma seqüência de valores desse tipo, em que cada elemento da seqüência pode ser obtido por meio da sua posição na mesma. Observe que o conjunto de pinos vermelhos do tabuleiro é representado como um array de pinos vermelhos – do tipo RedPeg[]. O conjunto de pinos brancos e o conjunto de navios são representados de maneira análoga.
Por ultimo, temos o jogo propriamente ditto – BattleshipApp – que representa o tabuleiro e os jogadores. Note que a classe BattleshiApp está sendo usada para representar o jogo como um todo. Desse modo, ela pode ser vista como a caixa cinza da Figura 4 apresentada acima, que contém tudo mais. Novamente, os nomes das variáveis que representam os dados do jogo foram escolhidos em ingles.

```
class BattleshipApp {
    private Board         redBoard;          // tabuleiro vermelho
    private Board         blueBoard;         // tabuleiro vermelho
    private FriendlyPlayer  friendlyPlayer;   // jogador aliado
    private EnemyPlayer     enemyPlayer;  // jogador inimigo
    // Dados adicionais do jogo entram aqui.
    // Métodos da classe BattleshipApp entram aqui.
}
```

Note nenhuma das classes definidas acima inclui definições de métodos. Como resultado, objetos dessas classes ainda não podem executar nenhuma ação. Isso não importa, por enquanto. O objetivo ao criar as "cascas" das classes do nosso jogo é estruturar o código do programa. Daqui em diante, saberemos exatamente onde adicionar uma nova porção de código e, quando for necessário, onde procurar por um erro. Isso pode não parecer muito importante agora, mas você verá a importância dessa estrutura mais tarde, eu garanto!

Vamos agora adicionar novos arquivos ao nosso projeto, cada um contendo a definição de uma das classes acima.

Um toque de classe

Vamos adicionar os novos arquivos na mesma ordem em que definimos as classes anteriormente: pinos, navios, jogadores, tabuleiros e, finalmente, o jogo propriamente dito.

Clique sobre o botão "New Class" (Nova Classe) e digite o nome "RedPeg" para essa classe.

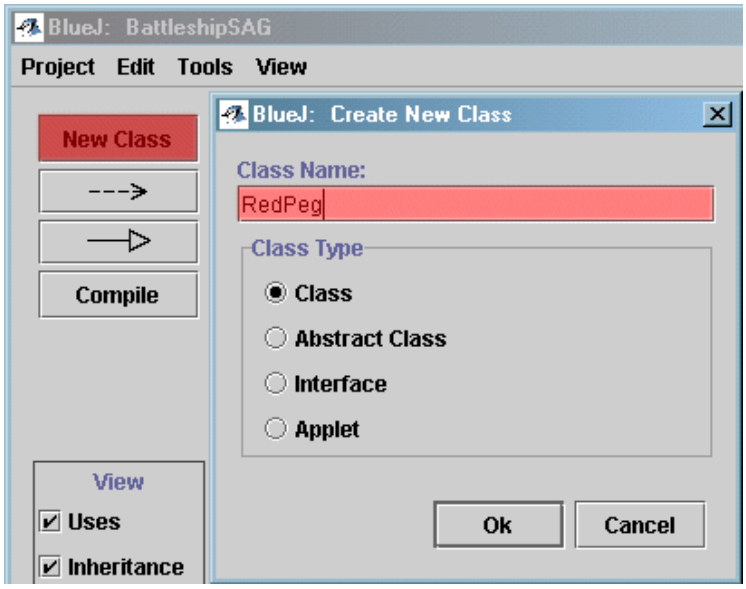


Figura 5— Criando a casca da classe RedPeg

Abra a classe "RedPeg" no editor de texto do BlueJ e dê uma olhada no código, gerado automaticamente. Como sempre, ele contém uma porção de dados extras, que nós não vamos necessitar. Elimine esse código extra, tal como fizemos anteriormente, para a classe BattleshipApp. Quando você tiver feito isso, o código da sua classe RedPeg deverá ser semelhante ao seguinte:

```
/**
* Pinos para marcar acertos no tabuleiro do jogo.
*
* @author (seu nome aqui)
* @version 0.1
*/

public class RedPeg
{
    // Variáveis de Instância
    // Mátodos
    /**
     * Construtor de objetos da classe RedPeg
     */
    public RedPeg()
   {
   }
}
```

Repita esse processo para as demais classes: WhitePeg, Ship, FriendlyPlayer, EnemyPlayer, Board e BattleshipApp. Certifique-se de eliminar, em cada classe, o código extra que não será necessário, e modificar os comentários em cada uma delas, adequadamente, tal como fizemos para a classe RedPeg.

Quando você terminar, seu projeto deverá ter a seguinte aparência:

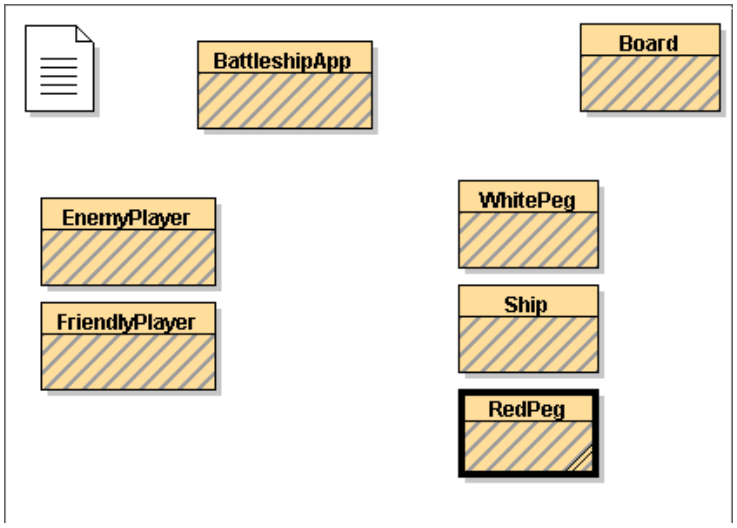


Figura 6 – As classes do jogo de Batalha Naval.

Agora que já criamos as "cascas" das classes do jogo, podemos ire m frente e adicionar as declarações de variáveis de instância nas classes Board e BattleshipApp, como vimos anteriormente. Clique duas vezes, seguidamente, sobre o ícone que representa a classe Board, para abrir o código dessa classe no editor de texto do BlueJ. O código da classe Board deverá ser semelhante ao seguinte:

```
/**
* Representa o tabuleiro de um jogador.
*
* @author (seu nome aqui)
* @version 0.1
*/
public class Board
{
    // Variáveis de Instância.

    // Métodos.
    /**
     * Construtor de objetos da classe Board
     */
    public Board()
   {
   }
}
```

Defina agora as variáveis de instância que representam o conjunto de pinos vermelhos, o conjunto de pinos broncos e o conjunto de navios. Quando você terminar, seu código deverá estar assim:

```
/**
* Representa o tabuleiro de um jogador.
*
* @author (seu nome aqui)
* @version 0.1
*/
public class Board
{
```

```
    // Variáveis de Instância.

    private RedPeg[]    hitMarkers;
    private WhitePeg[]  missMarkers;
    private Ship[]      fleet;

   // Métodos.
    /**
     * Construtor de objetos da classe Board
     */
    public Board()
   {
   }
}
```

Lembre-se do significado dos atributos usados em declarações de classes, variáveis e métodos. O atributo "private" significa que apenas o próprio objeto ao qual pertencem as variáveis pode ter acesso às mesmas – nesse caso, apenas um objeto da classe Board pode ter acesso às suas variáveis hitMarkers, missMarkers e fleet).

Uma vez que você tenha feito essas modificações no seu código, você verá uma mudança na janela do navegador de projeto do BlueJ: ela indica dependências entre as classes do seu projeto, por meio de setas que conectam essas classes. Bem, mas o que é uma dependência? É apenas uma maneira de dizer que existe um relacionamento no qual uma classe precisa de outra. O BlueJ irá mostrar, nesse caso, setas que indicam que a classe Board depende das classes RedPeg, WhitePeg e Ship. Isso é claro, pois Board faz referência a essas classes, quando declara suas variáveis de instância.

Existe agora apenas uma classe cujo código precisa ser modificado. Essa será uma tarefa sua. Abra a classe 'BattleshipApp' no editor do BlueJ e adicione as declarações de variáveis de instância apropriadamente, conforme o que apresentamos acima. Como sempre, você poderá verificar sua tarefa com a resposta com o código 5, apresentado na seção de códigos incluída no final deste tutorial.

Depois de concluir todas as modificações propostas acima, compile o seu projeto. Se tudo estiver OK, a janela do navegador de projeto do BlueJ deverá exibir o seguinte diagrama:

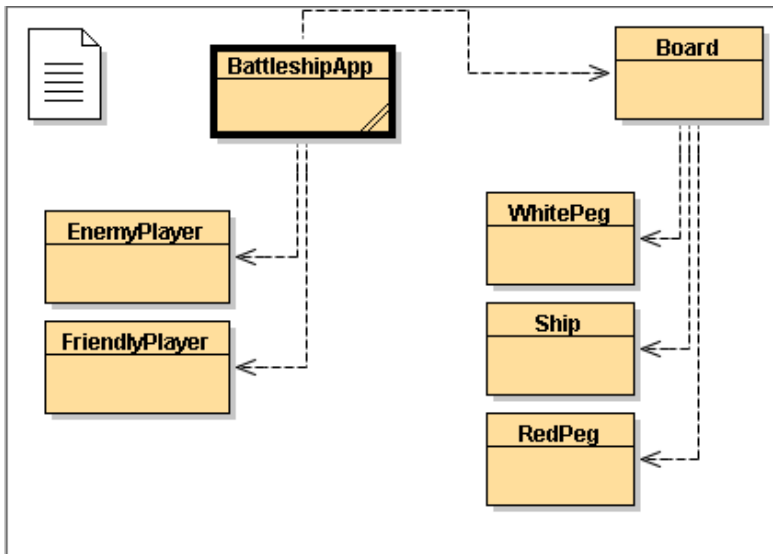


Figura 7 – Relacionamento entre todas as classes do projeto Battleship.

Se você obtiver algum erro de compilação, será necessário examinar o código, para identificar onde o erro ocorreu – mas isso não deverá ser muito difícil, uma vez que as classes ainda estão praticamente vazias.

Depois que tudo estiver compilado corretamente, poderemos continuar a adicionar código às classes acima, o que será feito na próxima seção

Programando a janela de abertura

Podemos, agora, começar a escrever o código que mencionamos na introdução deste módulo. Começaremos programando uma janela de abertura do jogo e, em seguida, veremos como adicionar código para tratar o evento do usuário clicar o mouse, quando o cursor está posicionado sobre a janela da aplicação.

Programando a janela de abertura

Primeiramente, vamos focalizar o problema de escrever o código da "janela de abertura" do programa. Essa janela é bastante simples: apenas exibe o nome do jogo – "Batalha Naval" – e a mensagem "clique o mouse para continuar", sobre um fundo de cor preta. O código para fazer isso não é difícil – já escrevemos algo semelhante, anteriormente. Sabemos como usar o método fillRect para preencher o fundo da janela em preto e como usar o método drawString para exibir uma mensagem na janela. Tudo o que precisamos fazer é criar um novo método, adicionar esses comandos ao corpo desse método, e incluir uma chamada a esse método em algum lugar do código da nossa aplicação. Chamaremos esse novo método de drawTitleScreen (o que significa, em português, desenha a janela de abertura). A definição desse método será inserida na classe BattleshipApp, e ele será um método privado – ou seja, declarado com o atributo private – uma vez que nenhuma outra classe, exceto BattleshipApp, deverá usá-lo. Como o método drawTitleScreen não produz nenhum valor, o tipo do valor retornado será void. O esqueleto da definição desse método seria, portanto, o seguinte:

```
/**
 * Draws the 'Welcome to Battleship' screen.
 */
private void drawTitleScreen() {
}
```

Agora, só falta escrever o código do corpo do método. Primeiro, queremos criar um fundo em preto – mais precisamente, queremos preencher a área de conteúdo da janela da aplicação com a cor preta. O que é a área de conteúdo (denominada, em Java, contentPane)? Você já sabe: é a parte da janela limitada pelas bordas e pela barra de título, também chamada de "área de cliente". Nosso primeiro passo é, então, obter o objeto que representa a área de conteúdo da janela, e armazenar uma referência a esse objeto em uma variável local – isto é, uma variável declarada no corpo do método. Chamaremos essa variável de clientArea (em português, área de cliente). Em Java, a área de conteúdo de uma janela é representada por um objeto da classe Container, que é uma classe pré-definida da biblioteca java.awt. Portanto, temos que incluir a seguinte declaração de variável, no corpo do método drawTitleScreen:

```
Container clientArea = this.getContentPane();
```

Neste ponto, você deverá estar se perguntando: "Ei, mas não temos que definir o método 'getContentPane'? Normalmente sim, mas, graças ao mecanismo de herança, isso não é necessário. Esse método já está definido na classe JFrame. Lembre-se que nossa classe BattleshipApp herda de JFrame – o que é especificado, no programa, como BattleshipApp extends JFrame. Portanto, todos as variáveis e métodos (públicos e protegidos), definidos na classe JFrame, são herdados pela classe Battleship, incluindo getContentPane.

Uma chamada do método getContentPane, dirigida a um objeto que representa uma janela – isto é, um objeto da classe JFrame, ou de qualquer classe que herda de JFrame – retorna uma referência ao objeto que representa a área de conteúdo dessa janela. No comando acima, o objeto alvo da chamada é representado pela palavra reservada this – você se lembra o que isso significa? Vimos que this representa, em cada ponto da execução do programa, o objeto corrente – ou seja, o objeto alvo da chamada que está sendo executada. Quem seria o objeto corrente, na chamada acima? Note que essa chamada ocorre no corpo do método drawTitleScreen, que estamos programando. Portanto, o objeto corrente, no momento da chamada do método getContentPane, é o objeto ao qual foi dirigida a chamada do método drawTitleScreen. Como esse método é um método privado da classe Battleship, o objeto corrente só pode ser um objeto dessa classe. De fato, na nossa aplicação, será criado um único objeto dessa classe, o qual irá representar a janela do jogo. Portanto, no comando acima, this representa a janela do jogo, e this.gertContentPane retorna a área de conteúdo dessa janela.
Precisamos, agora, preencher com a cor preta essa área de conteúdo da janela – representada pela variável clientArea. Para fazer isso, será preciso definir a cor corrente como preta. Precisamos, então, obter o objeto da classe Graphics associado à área de conteúdo da janela – lembre-se que um objeto dessa classe armazena as informações sobre a configuração gráfica corrente da janela, incluindo a cor usada para pintá-la. Para obter esse objeto, usamos uma chamada do método getGraphics, que é definido na classe Container. Como vamos precisar desse objeto por várias vezes, no código do programa, é conveniente armazenar uma referência a esse objeto em uma variável local. Para isso, incluiremos no corpo do método drawTitleScreen, a seguinte declaração:

```
Graphics gfx = clientArea.getGraphics();
```

Também vamos precisar saber as dimensões da área de conteúdo da janela (isso é fácil, como já vimos anteriormente):

```
int width = clientArea.getWidth();
int height = clientArea.getHeight();
```

Para preencher o fundo da área de conteúdo da janela, em preto, basta agora adicionar o seguinte:

```
gfx.fillRect(0, 0, width, height);
```

Lembre-se que os dois primeiros argumentos de uma chamada do método fillRect, ambos iguais a zero, na chamada acima, definem as coordenadas do canto superior esquerdo do retângulo a ser preenchido. Os dois últimos argumentos definem a largura e a altura do retângulo.
Para imprimir as mensagens, precisamos usar duas chamadas ao método drawString (lembre-se que esse método também é definido na classe Graphics). Como já sabemos,

esse método requer, como argumentos, as coordenadas cartesianas x e y da posição onde será escrito o texto. Uma vez que nossa janela tem 600 pixels de largura e 500 de altura, para centralizar o texto das mensagens, vamos definir posições próximas de (300,250), que são as coordenadas do centro:

```
gfx.setColor(Color.green);
gfx.drawString("BATALHA NAVAL", 260, 225);
gfx.setColor(Color.gray);
gfx.drawString("(clique o mouse para continuar)", 228, 275);
```

Uma vez que você tenha declarado o método drawTitleScreen na classe BattleshipApp, incluindo, no corpo desse método, as declarações de variáveis e os comandos descritos acima, compile seu programa novamente. Se você tiver algum problema, verifique seu código comparando-o com o código 6, apresentado na seção de códigos, incluída no final deste tutorial.

Quando seu programa tenha sido compilado corretamente, siga em frente e execute-o. Não acontece nada demais, certo? Você provavelmente já sabe porque: nós apenas adicionamos uma nova definição de método à classe BattleshipApp, mas não incluímos, no programa, nenhuma chamada a esse método. Portanto, o corpo desse método nunca é executado. Para que o método drawTitleScreen seja executado, devemos incluir uma chamada desse método em algum ponto do programa. Existe um ponto do programa que é particularmente adequado para isso. Você já vai saber qual é esse ponto, e porque ele é o mais adequado. Mas, para isso, precisamos ver um pouco mais de teoria.

Programação dirigida por Eventos

A tela do computador é um ambiente bastante movimentado – nela podem ser exibidas, simultaneamente, janelas de diversos programas que estão sendo executados. O usuário pode mover o mouse sobre essas janelas, pode digitar um texto em um campo de uma janela, pode fechar, minimizar ou maximizar janelas – ações desse tipo são chamadas de eventos. Um programa baseado em janelas – tal como o nosso jogo de Batalha Naval – em geral precisa estar preparado para responder a esses eventos que ocorrem no ambiente.

Por exemplo, você certamente terá notado que toda janela de programa tem um ícone em formato de X, no seu canto superior direito. O que acontece quando você clica o botão do mouse, com o cursor posicionado sobre esse ícone? Usualmente, a janela é fechada e o programa termina. Essa é a resposta padrão de um programa ao evento de clicar o mouse sobre esse ícone da janela.

Mas como um programa pode responder a um evento ocorrido no ambiente? Como ele saberá que o evento ocorreu?

Você deve saber que a execução de programas, em um computador, é controlada pelo sistema operacional – um programa especial, que gerencia todos recursos do computador: o processador, a memória e os dispositivos de entrada e saída de dados, tais como a tela, o teclado, o mouse etc. O sistema operacional é capaz de detectar a ocorrência de eventos no ambiente e enviar essa informação aos programas, por meio de mensagens como:

"Ei! O usuário acabou de clicar no botão close (fechar) da sua janela!"

"Atenção – o usuário moveu a janela de uma outra aplicação sobre a sua!"

"Alguém pressionou o botão 'd' enquanto seu programa estava ativo – o que você vai

fazer a respeito?"

Naturalmente, as mensagens enviadas pelo sistema operacional não são assim tão prosaicas – mas você deve ter compreendido a idéia.

Aplicações projetadas para lidar com eventos – tais como aplicações baseadas em janelas – são chamadas de "programas dirigidos por eventos" (em inglês, event-driven programs). O fato de ter que lidar com esses eventos introduz no código de aplicações baseadas em janelas, mesmo as mais simples, uma certa complexidade. Felizmente, em programas Java, grande parte dessa complexidade já é tratada pelas classes que implementam componentes de interfaces gráficas, pré-definidas nas bibliotecas java.awt e java.swing. Por exemplo, eventos como fechar, minimizar ou maximizar, já são tratados por janelas representadas por objetos da classe JFrame. Por isso, não é necessário tratar esses eventos em nossa classe BattleshipApp: ela herda esse comportamento da sua superclasse JFrame. Existem, entretanto, alguns eventos com os quais teremos que lidar no código da nossa classe, como o que descrevemos a seguir.

O evento de "requisição de pintura" da janela

Quando a execução de uma aplicação gráfica é iniciada, sua janela deve ser exibida na tela. O mesmo acontece quando a janela é maximizada, depois de ter sido minimizada, ou quando a janela volta ser ativa, depois de ter sido sobreposta pela janela de outro programa. Em cada um desses casos, o sistema operacional informa ao programa a ocorrência de um evento, que chamamos de "requisição de pintura" da janela (em inglês, "draw request"). Em uma janela representada por um objeto da classe JFrame, o tratamento desse evento origina uma chamada ao método paint, que "pinta" a janela na tela, novamente. Esse comportamento é herdado por toda subclasse de JFrame, tal como nossa classe BattleshipApp.

Portanto, fica fácil acomodar o código que responde à ocorrência desse evento, em uma classe que herda da classe JFrame: se queremos que nossa aplicação exiba algo em sua janela – como, por exemplo, a mensagem de abertura –, devemos redefinir o método paint, inserindo, no corpo desse método, o código correspondente ao que desejamos que seja exibido na janela.

Mas o que significa redefinir, nesse contexto? Simplesmente substituir o código herdado por outro, que definimos em nossa nova classe.

Redefinindo o método paint

Redefinir o método paint é muito simples. Tudo o que precisamos fazer é acrescentar as seguintes linhas de código à classe BattleshipApp:

```
/**
* Desenha a janela do jogo.
*/
public void paint(Graphics gfx) {
}
```

Antes de ser inserido esse trecho de código na classe BattleshipApp, toda ocorrência do evento "requisição de pintura" ocasionava uma chamada ao método paint, definido na classe Component e herdado por JFrame. Depois de inseridas essas linhas de código, o tratamento desse evento será feito pelo código contido no corpo dessa nova definição do método. Essa é uma característica chave do mecanismo de herança: Java usa o méto-

do herdado por uma classe somente quando esse método não for (re)definido na própria classe. Uma vez que você defina o seu próprio método paint, na classe BattleshipApp, Java irá usar essa definição, em uma chamada desse método dirigida a objetos da classe BattleshipApp, e não a definição pré-existente na classe JFrame (Obs: de fato, o método paint é definido na classe Component, e herdado dessa classe por JFrame).
Você já deve ter adivinhado qual é o código que devemos inserir no corpo do método paint – simplesmente uma chamada do método drawTitleScreen:

```
/**
* Desenha a janela do jogo.
*/
public void paint(Graphics gfx) {
    this.drawTitleScreen();
}
```

A partir de agora, quando nossa classe BattleshipApp receber uma requisição de pintura, esse método paint será executado, resultando em que a janela de abertura do jogo seja novamente exibida na tela. Fácil!

Tente adicionar esse código à classe BattelshipApp e, em seguida, compilar e executar o programa. Você deverá ver, na tela, a janela de abertura do jogo. Se isso não ocorrer, verifique o código do seu programa com o código 7, da seção de códigos, incluída no final deste tutorial.

Muito bem. Agora temos que aprender como tratar o evento do usuário clicar sobre essa janela. Desejamos que, quando isso ocorrer, o programa exiba então o tabuleiro do jogo, deixando tudo pronto para que os jogadores possam começar a jogar. É o código correspondente a isso que vamos programar no próximo módulo deste tutorial. Mas, antes disso, vamos rever o que aprendemos neste módulo.

Conclusão

Parece que cumprimos as tarefas propostas no início deste módulo:

5. Aprendemos a criar uma aplicação com interface gráfica a partir do zero.

6. Aprendemos sobre o conceito de herança em Java e usamos este conceito para programar nossa classe BattleshipApp, como subclasse de JFrame, facilitando assim, a programação de várias ações da janela do nosso jogo, além de evitar escrever código para diversas outras, tais como fechar, minimizar ou maximizar a janela.

7. Definimos a estrutura no nosso programa, organizando-o em classes que correspondem aos objetos da aplicação, existentes no mundo real.

8. Aprendemos um pouquinho sobre o conceito de programação dirigida por eventos. Mas há ainda um bocado a aprender sobre isso, e é o que faremos no próximo módulo.

Como dissemos na introdução deste tutorial, agora precisamos aprender como tratar o evento de o usuário clicar o mouse sobre uma janela, para passar da janela de abertura do jogo para a sua janela principal, onde é exibido o tabuleiro do jogo. Então vamos já tratar disso, seguindo para o próximo módulo.

Módulo 5

Mais sobre Interfaces Gráficas e Tratamento de Eventos

Introdução

Neste módulo, vamos continuar a construção do nosso jogo de Batalha Naval, passando da janela de abertura do programa, para a janela, onde é exibido o tabuleiro do jogo. Vamos seguir os seguintes passos:

Aprender um pouco mais sobre o tratamento de eventos em programas, em particular, o evento de o usuário clicar o mouse sobre uma janela.

Criar o código para tratamento desse evento, em nossa aplicação, o qual deverá fazer com que seja exibida a janela principal do jogo.

Refazer o código que exibe o tabuleiro do jogo, tratando detalhes adicionais da interface.

Para fazer isso, você também irá também aprender um novo tipo de comando, existente em toda linguagem de programação – o comando condicional.

Pronto para recomeçar? Então, siga em frente!

Tratando o mouse

Vamos agora adicionar, ao nosso programa, código para tratar o evento de o usuário clicar o mouse sobre a janela de abertura do jogo. Isso deve fazer com que seja exibida a janela principal do jogo, a qual contém o tabuleiro. Isso não é difícil, mas requer, primeiro, que você entenda o que são eventos do mouse, como eles podem ser detectados pelo programa e, finalmente, como fazer com que nosso jogo realize alguma ação, quando detecta esses eventos.

Eventos do mouse

Como usuários de um computador, esperamos que, ao clicar o mouse sobre a janela da interface de um programa, uma ação correspondente irá ocorrer. Clicamos sobre botões da janela, "arrastamos" suas bordas, marcamos trechos de textos... nunca precisamos nos preocupar com o mecanismo que faz com que a ação correspondente seja executada.

Como programadores, é nossa responsabilidade garantir que todo evento de clicar ou movimentar o mouse seja tratado da maneira apropriada. A nossa sorte é que fazer isso não é difícil, em programas Java. Cada pequeno movimento do mouse, cada ação de pressionar ou soltar um botão do mouse, ocasiona o envio de uma mensagem correspondente ao programa, à qual ele pode responder. Essas mensagens são chamadas de MouseEvents (Eventos do Mouse), e contêm informações úteis para o tratamento de cada evento particular. Mais precisamente, MouseEvents são objetos Java, que contêm variáveis e métodos, úteis para que o programa possa saber o que o mouse está fazendo e qual é a sua posição relativa dentro da janela.

Mais adiante, vamos usar esses métodos e variáveis para obter informação sobre a posição corrente do mouse quando usuário clicar sobre a janela do jogo. Por enquanto, vamos nos preocupar apenas com o fato de o usuário clicar o mouse, em qualquer posição da janela, indicando que ele deseja sair da janela de abertura e passar para a janela principal do jogo.

"Ouvindo" um evento

Antes de podermos processar eventos do mouse, precisamos informar ao sistema de execução de programas Java que esses eventos interessam ao programa, ou, em outras palavras, que o programa deseja executar alguma ação, como resposta à ocorrência de um evento dessa natureza. Fazemos isso designando um MouseListener (em português, Ouvinte do Mouse). Um MouseListener é um objeto especial do programa, ao qual o sistema operacional irá comunicar diretamente a ocorrência de qualquer evento do mouse. Se nenhum ouvinte for especificado, todos esses eventos simplesmente serão ignorados.

Essa idéia de Listener (Ouvinte) é usada com freqüência em Java. Existem ouvintes para eventos do teclado, cliques sobre botões, eventos de janelas (como minimizar e maximizar) e qualquer outro que você queira identificar. De fato, em algum ponto, vamos criar nossos próprios eventos, particulares do nosso jogo, a ouvintes para os mesmos. Esse mecanismo de evento/ouvinte funciona muito bem para diversas situações – estamos vendo apenas um caso particular.
Bem, como designamos um ouvinte para eventos do mouse? Primeiro, temos que escolher um objeto, em nosso projeto, que será o responsável por receber notificação da ocorrência de um evento do mouse. No nosso acaso, vamos usar o objeto da classe BattleshipApp, uma vez que ele representa a janela do jogo. Em seguida, temos que modificar a declaração da classe BattleshipApp, do seguinte modo:

```
public class BattleshipApp extends JFrame implements MouseListener
```

Note as novas palavras implements MouseListener, adicionadas à declaração. Vamos entender o que isso significa.
A palavra reservada implements indica que a nova classe deve incluir determinados métodos. MouseListener especifica quais são esses métodos. Nesse caso, toda classe que quer ser um MouseListener deve definir os seguintes métodos:

```
public void mouseClicked(MouseEvent event)
public void mouseEntered(MouseEvent event)
public void mouseExited(MouseEvent event)
public void mousePressed(MouseEvent event)
public void mouseReleased(MouseEvent event)
```

Implementando esses métodos, uma classe se torna apta a receber informação sobre eventos do mouse, e responder a esses eventos. O método mouseEntered será chamado sempre que o cursor indicador da posição do mouse mover-se para dentro da janela, e mouseExited, quando ele mover-se para fora da janela. O método mousePressed será chamado quando o usuário pressionar o botão, e, mouseReleased, quando ele soltar o botão. O método mouseClicked será chamado quando o usuário clicar o mouse, isto é, pressionar e soltar o botão rapidamente.

Uma observação importante: Quando a declaração de uma classe especifica que ela

implements MouseListener, o compilador Java exige que esses métodos sejam definidos na classe, mas o corpo de cada um desses métodos é especificado pelo programador, da maneira como achar conveniente, para tratar cada evento. O corpo do método pode, até mesmo, ser vazio, indicando, nesse caso, que nenhuma ação é executada, no caso de ocorrência do evento.

Mas, porque isso? Uma resposta simples é a seguinte: A máquina de execução de programas Java irá chamar esses métodos em determinadas circunstâncias (ou seja, quando o evento correspondente ocorrer). Se o ouvinte especificado não define esses métodos, então não haverá o que chamar, o que ocasionaria um erro, durante a execução do programa. Em outras palavras, quando se usa a palavra implements em uma declaração de classe, isso significa um compromisso de definir um determinado conjunto de métodos nessa classe.

MouseListener é um exemplo do que é chamado, em Java, uma interface. Uma interface é semelhante a uma classe, mas contém apenas definições de cabeçalhos de métodos – os métodos que se requer que sejam definidos em classes que implementam essa interface. Você pode pensar em uma interface como um contrato – uma vez que você adere a esse contrato, você obtém alguns recursos, mas também se compromete com algumas obrigações. Nesse caso, aderir ao contrato significa ser informado sobre eventos do mouse e, em contrapartida, implementar os métodos acima.

Tratando o mouse no nosso programa

OK – vamos traduzir essa idéias em nosso programa. Comece alterando a antiga declaração da classe BattleshipApp:

```
public class BattleshipApp extends JFrame
```

para a nova declaração:

```
public class BattleshipApp extends JFrame implements MouseListener
```

e adicione a seguinte cláusulas de importação, no topo do programa:

```
import java.awt.event.*;
```

Isso é necessário porque a interface MouseListener e a classe MouseEvent são definidas em um subpacote do pacote AWT, e Java exige que subpacotres sejam incluídos explicitamente.
Vá em frente: compile seu programa. Você verá que o BlueJ indicar um erro de compilação, dizendo que você deveria declarar sua classe BattleshipApp como abstrata, porque você não definiu nenhum dos cinco métodos da interface MouseListener. Em outras palavras, você quebrou a promessa de implementar MouseListener, feita na declaração da classe BattleshipApp. Para resolver esse problema, basta adicionar os métodos faltantes no final do corpo da classe:

```
/**
 * MouseListener methods.
 */
public void mouseClicked(MouseEvent event) {// Empty}
public void mouseEntered(MouseEvent event) {// Empty}
public void mouseExited(MouseEvent event) {// Empty}
```

```
public void mousePressed(MouseEvent event) {// Empty}
public void mouseReleased(MouseEvent event) {
    Graphics gfx = this.getGraphics();
     gfx.setColor(Color.yellow);
     gfx.drawString("Clique!", event.getX(), event.getY());
}
```

Note inserimos algum código ao corpo do método mouseReleased. A maior parte desse código deverá te parecer familiar, portanto, vamos explicar apenas um pouquinho. O argumento event armazena informação sobre as condições em que o evento ocorreu. Os métodos getX e getY retornam, respectivamente, as coordenadas x e y da posição do mouse, quando o evento ocorreu (mais precisamente, da posição relativa, na janela, do cursor que representa o mouse).
Falta apenas uma linha de código! Agora que preparamos a classe BattleshipApp, de maneira que objetos dessa classe possam ser ouvintes de eventos do mouse, temos que especificar que esses eventos devem ser informados ao objeto dessa classe que é criado. Fazemos isso adicionando a seguinte linha de código ao construtor de da classe BattleshipApp:

```
this.addMouseListener(this);
```

O método addMouseListener é herdado, por BattleshipApp, da classe JFrame. Quando chamamos esse método, devemos passar como argumento o novo ouvinte do mouse (um MouseListener). O argumento especificado é this, uma vez que o próprio objeto da classe BattleshipApp, que está sendo criado, é esse ouvinte. Isso pode parecer um pouco confuso, mas se você refletir por alguns minutos, temos certeza de que vai entender.

Tente compilar o programa novamente. Se não funcionar, confira o seu programa com o código 8, apresentado na seção de códigos incluída no final deste tutorial.

Execute o programa. Clique o mouse. Veja o que acontece!

Quando você tiver terminado de brincar, podemos continuar nosso projeto

Revendo o código do tabuleiro do jogo

Estamos quase lá! Tudo o que precisamos fazer agora é ligar o código que trata o evento de clicar o mouse com nosso código para traçar a grade do tabuleiro do jogo, desenvolvido no módulo 3. Depois disso, temos apenas que traçar a segunda grade do tabuleiro, e pronto! Temos (quase) pronta a interface da nossa Batalha Naval.

Reduzir, Reusar, Reciclar

Ligar o tratamento de eventos do mouse ao código para traçar o tabuleiro é fácil. Sabemos que o método mouseReleased será chamado sempre que o usuário soltar o botão do mouse. Portanto, tudo o que precisamos fazer é inserir o código para traçar a grade do tabuleiro no corpo desse método.

Ao invés de escrever esse código novamente, vamos recuperá-lo, do módulo 3. Abra seu antigo projeto Battleship no BlueJ e clique sobre o ícone que representa a classe StandaloneApp. Copie o código para traçar a grade, do corpo do método paint, de

StandaloneApp, e, em seguida, feche o antigo projeto Battleship (selecionando a opção 'Project' e, em seguida, a opção 'Close').
Adicione um novo método à sua classe BattleshipApp atual:

```
/*
 * Desenhar grade.
 */
private void drawGrid() {
}
```

Uma vez que você tenha feito isso, cole, no corpo desse método, o antigo código para desenhar a grade, que você acabou de copiar:

```
/*
 * Desenhar grade.
 */
private void drawGrid() {
    // Obtém informação sobre a area de conteúdo da janela.
    Container workArea = this.getContentPane();
    Graphics workAreaGfx = workArea.getGraphics();

    // Preenche o fundo da janela em preto.
     workAreaGfx.setColor(Color.black);
    int width = workArea.getWidth();
    int height = workArea.getHeight();
    workAreaGfx.fillRect(0, 0, width, height);

    // Código para traçar as linhas da grade.
    // Primeiro, define a cor do traço.
     workAreaGfx.setColor(Color.red);

    // Traça as linhas vertcais.
    // Especifica condições iniciais do loop.
    int lineCounter = 0;
   int startX = 5;
   int startY = 5;

    // Usa um loop para traçar as linhas.
    while (lineCounter < 11) {
       // Desenha linha.
        workAreaGfx.drawLine(startX, startY,
       startX, startY + 200);

        // Move para frente 20 pixels.
       startX += 20;
        // Incrementa o contador do loop.

       lineCounter += 1;
   }

    // Traça linhas verticais.
    // Especifica condições iniciais do loop.
    lineCounter = 0;
   startX = 5;
   startY = 5;
```

```
    // Usa um loop para traçar as linhas.
    while (lineCounter < 11) {
       // Desenha linha.
        workAreaGfx.drawLine(startX, startY, startX + 200, startY);

        // Move para baixo 20 pixels.
       startY += 20;
        // Incrementa o contador do loop.

        lineCounter += 1;
    }
}
```

Agora, apague o antigo código do corpo do método mouseReleased e inclua aí uma chamada do método drawGrid.

Compile e execute o seu programa. Clique o mouse (sobre a janela da aplicação). Tudo funciona corretamente, não é?

Nem tudo funciona corretamente...

Bem, nem tudo funciona corretamente... Para ver porque, minimize a janela do jogo de Batalha Naval e restaure a janela em seguida. Ou então, mova uma outra janela, sobrepondo-a parcialmente à do jogo, de maneira que cubra a grade vermelha, e então remova essa janela novamente. A grade do tabuleiro desaparece e a janela de abertura do jogo aparece novamente! Que diabos está acontecendo?

O problema está relacionado com a idéia de programação dirigida por eventos e o método paint. Lembre-se que programamos paint para exibir a janela de abertura. E lembre-se também que esse método é chamado sempre que ocorre um evento de "requisição de pintura" da janela, como acontece nas situações que descrevemos acima.

Será que encontramos um problema insolúvel? É claro que não – a solução é descrita a seguir.

A maioria dos programas – em particular, jogos – tem diversos modos de operação. Por exemplo, em seu programa processador de textos, você pode estar no modo de "salvar" o arquivo, no modo de "inserir" uma figura, ou de "localizar" uma palavra, ou no modo normal de "editar", entre outros. Programadores usualmente chamam esses modos de estados, algo com o qual você tem familiaridade, no mundo real.

Considere, por exemplo, um jogo de Batalha Naval "real". Você e seu amigo estão sentados, organizando as peças para começar o jogo - estão no estado de "introdução" ao jogo. Em seguida, vocês pegam o tabuleiro e posicionam nele os seus navios – vocês estão no estado de "preparação". Finalmente, por um longo tempo, vocês ficarão no estado "jogando" (playing) – cada jogador atirando, em sua vez, para tentar acertar um navio do oponente. Finalmente, quando todos os navios de um dos jogadores forem atingidos, o estado será de "fim do jogo" (endgame) – aí será determinado o vencedor.

Bem, programas funcionam exatamente do mesmo modo. No estado de introdução ao jogo – intro state – nosso programa deverá exibir a janela de abertura. Depois disso, ele entra no estado de preparação – setup state – no qual ele exibe o tabuleiro e deixa que os jogadores posicionem seus navios. Então, vem o estado em que os jogadores podem fazer suas jogadas – playing state – que simula o estado correspondente no do

jogo real. Por último, no estado final – game over – o programa exibe a mensagem "Excelente partida!" e, em geral, comporta-se de maneira mais educada do que um jogador real, mesmo que tenha sido vencido no jogo.

Nossa tarefa, agora, é descobrir como representar esses estados e usá-los para descobrir quando devemos exibir a janela de abertura, ou a janela principal do jogo, ou a janela de fim de jogo. Felizmente, isso é muito fácil.

Representando os estados do jogo

Podemos usar números para representar os estados do jogo – por exemplo, 0, para o estado "intro state", 1 para o "setup state", 2 para "playing" e 3 para "game over". Então, poderíamos armazenar o estado corrente em uma variável inteira (de tipo int). Para fazer isso, adicione à classe BattleshipApp, a seguinte declaração da variável de instância de nome "gameState" (estado do jogo):

```
private int gameState = 0;
```

A variável gameState é iniciada com o valor 0, que corresponde ao estado inicial do jogo.
Ok, está tudo certo, exceto que... para alguém que olha o código do programa (ou para nós mesmos, revendo o código depois de algumas semanas), não fica claro a que estado corresponde cada número. Seria melhor poder substituir '0' por algo mais descritivo. A idéia é usar constantes – que, em Java, são variáveis declaradas com os atributos static e final, como mencionamos em um módulo anterior deste tutorial. Além disso, como essas consantes apenas serão usadas pela classe BattleshipApp, vamos declará-las também com o atributo private:

```
private static final int GAME_STATE_INTRO = 0;
private static final int GAME_STATE_SETUP = 1;
private static final int GAME_STATE_PLAYING = 2;
private static final int GAME_STATE_GAMEOVER = 3;
```

Depois de inserir esse trecho de código, a "seção de variáveis" da nossa classe BattleshipApp estará assim:

```
private static final int GAME_STATE_INTRO = 0;
private static final int GAME_STATE_SETUP = 1;
private static final int GAME_STATE_PLAYING = 2;
private static final int GAME_STATE_GAMEOVER = 3;

// variáveis de instância
private Board redBoard;
private Board blueBoard;
private FriendlyPlayer friendlyPlayer;
private EnemyPlayer enemyPlayer;
private int gameState;
```

Agora, ao atribuir um valor inicial à variável gameState, que representa o estado corrente do jogo, podemos usar GAME_STATE_INTRO, em lugar de '0'. Faça isso, e o código do construtor da classe BattleshipApp deverá ser o seguinte:

```
/**
```

```
* Construtor de objetos da classe BattleshipApp
*/
public BattleshipApp()
{
    this.setSize(600, 500);
    this.setVisible(true);
     this.addMouseListener(this);
      this.gameState = GAME_STATE_INTRO;
}
```

Agora que já criamos e inicializamos a variável de estado do jogo, temos que usá-la.

Determinando e modificando o estado do jogo

De volta ao nosso problema. Queremos inserir uma chamada de drawGrid no método paint, de maneira que as grades do tabuleiro sejam exibidas sempre que ocorrere um evento de "requisição de pintura" da janela. Infelizmente, também queremos uma chamada de drawTitleScreen no método paint, pela mesma razão. O que queremos é que paint desenhe a janela de abertura, caso o jogo esteja no estado "intro state", e desenhe as grades do tabuleiro, caso esteja no estado "setup state". Em outras palavras, queremos que paint seja mais esperto: "Se o estado corrente do jogo for intro state, desenhe a janela de abertura, mas se eu estiver no estado setup state, desenhe as grades do tabuleiro".

Como você já deve ter adivinhado, existe um comando Java que nos escrever exatamente isso – é o comando "if...else". Comandos desse tipo são extremamente úteis em programas, e muito fáceis de se entender. Eles nos permitem definir um bloco de código que é executado condicionalmente, conforme uma determinada condição seja verdadeira (true) ou falsa (false). Vamos experimentar essa idéia, primeiramente escrevendo nosso comando if...else, em português.

SE (alguma condição) ENTÃO execute este código:

[INÏCIO DO BLOCO]

...código...

[FIM DO BLOCO]

Podemos tornar o comando mais poderoso, adicionando a ele a idéia de 'senão':

SE (alguma condição) ENTÃO execute este código:

[INÏCIO DO BLOCO]

...código...

[FIM DO BLOCO]

SENÃO SE (alguma outra condição) ENTÃO execute este código:

[INÏCIO DO BLOCO]

...código...

[FIM DO BLOCO]

SENÃO SE (uma terceira condição) ENTÃO execute este código:

[INÏCIO DO BLOCO]

...código...

[FIM DO BLOCO]

...e assim por diante.

Nesse caso, a primeira condição é avaliada e, caso ela não seja verdadeira, pulamos para a verificação da segunda condição. Isso é repetido, até que uma condição verdadeira seja encontrada, ou o comando if... else termine.

Essa sintaxe "ingênua" é, de fato, muito próxima da maneira como o comando é escrito em Java. Sabemos que Java usa chaves ('{' e '}') para marcar o início e o fim de blocos de código. Além disso, a palavra "ENTÃO" poderia perfeitamente ser omitida. No mais, é só traduzir para o inglês: SE para IF e SENÃO para ELSE. Fazendo essas alterações, temos a seguinte sintaxe do comando if...else, em Java:

```
if (alguma condição) {
      ... código...
}
elseif (alguma outra condição) {
      ... código...
}
elseif (uma terceira condição) {
      ... código...
}
```

Existe ainda uma pequena alteração a ser feita: Java possibilita incluir uma cláusula "else", após todas as outras seções, que sera sempre executada, caso nenhuma das condições seja verdadeira. Vamos ver um exemplo simples, usando português, em vez de Java:

```
SE (Eu tenho menos que 1 ano de idade) {
       diga  "Goo goo, daa daa."
}
SENÃO SE (Eu tenho menos que 60 anos de idade) {
       diga "A aposentadoria está chegando!"
}
SENÃO {
       diga "Parabéns, vovô!"
}
```

Não é preciso explicar muito: é claro que a mensagem associada ao "SENÃO" final será impressa quando esse trecho de código for avaliado tendo como objeto alguém com idade maior ou igual a 60 anos.
Aplicando essa idéia ao nosso exemplo, vamos usar um comando if...else no corpo do método paint:

```
/**
```

```
* Desenha a janela do jogo.
*/
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
       this.drawTitleScreen();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       this.drawGrid();
   }
}
```

Note que Java usa o operador (==) para indicar uma operação de comparação: "os dois valores são iguais"?
Bem, estamos quase terminando: declaramos uma variável de instância (gameState) para representar o estado do jogo, declaramos constantes para representar cada estado particular e modificamos o corpo do método paint, de maneira a chamar diferentes métodos, com base no estado corrente. Agora apenas precisamos adicionar o código para fazer passar do estado "intro" para o estado "setup", quando o usuário solta o botão do mouse – ou seja, modificar o código do método mouseReleased, do seguinte modo:

```
public void mouseReleased(MouseEvent event) {
    this.gameState = GAME_STATE_SETUP;
    this.repaint(); // Redesenha a janela do jogo.
}
```

É isso. Apenas uma última observação: o método repaint usado acima, é semelhante ao método paint (e também pré-definido, na classe Component, e herdado por BattleshipApp, via JFrame), sendo usado para redesenhar a janela, em lugar de paint, por questões de maior eficiência.

Se você compilar e executar seu programa novamente, verá que ele agora tem o comportamento correto.

Se você obtiver algum erro de compilação, verifique o seu programa com o código 9, apresentado na seção de códigos incluída no final deste tutorial.

Isso conclui a parte mais difícil. Agora, basta desenhar o restante do tabuleiro.

Janela principal do jogo

Finalmente, estamos prontos para o último passo deste módulo – traçar as duas grades do tabuleiro do jogo. Podemos nos sentir tentados a adaptar o código que já escrevemos, para fazer isso. Entretanto, existem alguns problemas com esse código:

Esse código traça apenas uma grade do tabuleiro, em uma posição fixa.

Esse código traça a grade apenas na cor vermelha.

O código está incluído na classe BattleshipApp, mas deveria estar na classe Board (Tabuleiro), uma vez que a grade é parte do tabuleiro do jogo.

Em vez de tentar adaptar o código para traçar a grade, que já escrevemos na classe BattleshipApp, será instrutivo começarmos tudo de novo.

Começando de novo

Vamos parar um pouco pensar sobre o nosso novo código do método drawGrid, para traçar as grades do tabuleiro do jogo. Primeiro, esse método passará a ser definido na classe Board. Segundo, queremos poder traçar grades de qualquer cor – o que significa que a cor desejada deve ser um parâmetro do método. De maneira análoga, queremos poder traçar a grade em qualquer posição da janela – portanto, novamente, as coordenadas (x, y) da posição da grade devem constituir parâmetros do método.
Colocando em prática essa idéia, o esqueleto do nosso novo método drawGrid, da classe Board seria:

```
/**
 * Draw the board's grid.
 */
public void drawGrid(Color gridColor, int startX, int startY) {
}
```

Observe o tipo do parâmetro gridColor , que representa a cor das linhas da grade: Color. A classe Color é também uma classe pré-definida em Java, na biblioteca java.awt. Objetos dessa classe representam, naturalmente, cores. Para que a classe Board possa usar Color, precisamos incluir, no início do arquivo que contém a classe Board, a seguinte cláusula de importação:

```
import java.awt.*;
```

Modificando BattleshipApp

Vamos ver como a classe BattleshipApp deve ser modificada, de maneira a usar o novo método drawGrid. Primeiro, vamos modificar o nome do método drawGrid, da classe BattleshipApp, para drawGrids, uma vez que vamos traçar duas grades no tabuleiro, daqui em diante. Certifique-se de também modificar o nome do método na chamada ao mesmo incluída no corpo do método paint:

```
/**
* Desenha a janela do jogo.
*/
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
       this.drawTitleScreen();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       this.drawGrids();
   }
}
```

Em seguida, vamos remover o código existente no corpo do (novo) método drawGrids e substituí-lo pelo seguinte:

```
/**
* Desenha as grades do tabuleiro do jogo.
*/
private void drawGrids() {
    // Obtém o objeto que representa a área de conteúdo da janela.
```

```
    Container clientArea = this.getContentPane();

    // Obtém o objeto da classe Graphics associado à area de conteúdo da janela.
    Graphics gfx = clientArea.getGraphics();

    // Obtém as dimensões da area de conteúdo da janela.
    int width = clientArea.getWidth();
    int height = clientArea.getHeight();

    // Preenche o fundo em cor preta.
     gfx.setColor(Color.black);
    gfx.fillRect(0, 0, width, height);

    // Desenha as duas grades do tabuleiro.
     this.redBoard.drawGrid(Color.red, 5, 5);
     this.blueBoard.drawGrid(Color.blue, 255, 5);
}
```

A maior parte desse código já foi vista anteriormente. Entretanto, preste atenção nas duas últimas linhas. Você se lembra que, para representar os dois tabuleiros do jogo, declaramos originalmente as variáveis redborad e blueborard, no início da classe BattleshipApp? Finalemente vamos usar essas variáveis. Infelizmente ainda temos um problema: nós apenas declaramos essas variáveis de instância, mas ainda não atribuímos a elas nenhum valor. Reveja, por exemplo, a declaração de redborad:

```
private Board redBoard;
```

Essa declaração apenas especifica o nome e o tipo da variável redBoard, mas não atribui à variável nenhum valor. Em outras palavras, quando essa declaração é processada, a variável declarada ainda não referencia nenhum objeto – dizemos que ela é uma referência "null" (ou seja, nula). Naturalmente, o que desejamos é atribuir a essa variável uma referência ao objeto que vai representar a grade vermelha do tabuleiro do jogo. Isso pode ser feito incluindo-se o seguinte comando em algum ponto do código da classe BattleshipApp:

```
this.redBoard = new Board();
```

Usualmente, atribuímos valores iniciais a variáveis de instância de uma classe no construtor dessa classe. Tendo isso em mente, vamos modificar o construtor da classe BattleshipApp, para incluir o comando acima, assim como um comando para atribuir valor inicial à variável blueBoard:

```
/**
* Construtor de objetos da classe BattleshipApp
*/
public BattleshipApp()
{
    this.setSize(600, 500);
    this.setVisible(true);
     this.addMouseListener(this);
    this.gameState = GAME_STATE_INTRO;

     // Atribui valores iniciais às variáveis que representam as grades do tabuleiro do
jogo.
    this.redBoard = new Board();
```

```
    this.blueBoard = new Board();
}
```

As duas últimas linhas fazem com que redBoard e blueBoard passem a conter referências válidas a objetos – isto é, diferentes de "null". Os objetos referenciados por essas variáveis são criados, nos comandos acima, como resultado da avaliação expressão new Board(). Como isso é feito no corpo do construtor da classe BattleshipApp, as variáveis redBoard e blueBoard conterão referências válidas a objetos, quando forem executadas as chamadas do método drawGrid, no corpo do método paint, tendo como alvo os objetos referenciados por essas variáveis. Se essas chamadas ocorressem antes que os objetos tivessem sido previamente criados, Java indicaria a ocorrência do erro – ou exceção – "Null Pointer Exception". "Null Pointer Exception" é simplesmente a maneira de Java indicar que foi usada, no programa, uma variável (cujo tipo é uma classe) que não contém uma referência válida a um objeto, mas sim uma referência "null".

Compile e execute o seu projeto. É claro que não sera exibida nenhuma grade do tabuleiro do jogo, porque ainda não escrevemos o código da classe Board. OK, de qualquer maneira, é bom compilar o programa, nesse ponto, apenas para garantir que não introduzimos, no código, nenhum erro que possa ser detectado pelo compilador.

A classe Board

Finalmente, podemos de fato escrever o código para traçar uma grade do tabuleiro do jogo. A primeira coisa a fazer é especificar a cor da linha da grade. Opa! Temos um problema:

```
/**
 * Desenha uma grade do tabuleiro do jogo.
 */
public void drawGrid(Color gridColor, int startX, int startY) {
    // Não podemos especificar a cord as linhas da grade, porque não temos acesso
    // ao objeto gráfico associado à área de conteúdo da janela!
}
```

Hummm... Para especificar a cor da linha da grade, usamos anteriormente o seguinte código:

```
    Container workArea = this.getContentPane();
    Graphics workAreaGfx = workArea.getGraphics();
   // Especifica a cor das linhas da grade.
    workAreaGfx.setColor(Color.red);
```

Note que esses comandos podiam ser usados no método drawGrid, porque ele era definido na classe BattleshipApp. Essa classe herda, de JFrame, o método getContentPane, que é usado para obter o objeto que representa a área de conteúdo da janela. Esse objeto, por sua vez, é usado como alvo da chamada de getGraphics, para obter o objeto gráfico associado à área de conteúdo da janela. Agora que passamos a definição de drawGrid para a classe Board, isso não é possível, uma vez que Board não herda de JFrame. Portanto, não temos como usar o método setColor, para especificar a cor das linhas da grade.
Para contornar esse problema, a idéia é que o método drawGrid tenha, como parâmetro,

um objeto da classe Graphics. O objeto gráfico associado à área de conteúdo da janela seria então passado como argumento, no lugar desse parâmetro, nas chamadas ao método drawGrid incluídas no corpo do método drawGrids, o qual é definido na classe BattleshipApp. Portanto, as duas últimas linhas de drawGrids teriam a seguinte forma:

```
// Desenha as duas grades do tabuleiro.
this.redBoard.drawGrid(gfx, Color.red, 5, 5);
this.blueBoard.drawGrid(gfx, Color.blue, 255, 5);
```

Repare o novo argumento – gfx – adicionado às duas chamadas de drawGrid: gfx referencia o objeto da classe Graphics associado à área de conteúdo da janela do jogo (Compare esse código com o que havíamos escrito anteriormente).
A definição do método drawGrid, na classe Board, também deve modificada, passando a ser assim:

```
/**
 * Desenha uma grade do tabuleiro do jogo.
 */
public void drawGrid(Graphics gfx, Color gridColor, int startX, int startY) {
    // Especifica a cor das linhas da grade.
     gfx.setColor(gridColor);
}
```

Agora que já especificamos a cor, podemos traçar as linhas da grade. Aqui está o código para traçar as linhas horizontais:

```
// Traça as linhas horizontais da grade.
int lineCounter = 1;
int x = startX;
int y = startY;
while (lineCounter <= 11) {

    gfx.drawLine(x, y, x + 200, y);
   y = y + 20;
    lineCounter++;
}
```

Como já vimos esse código antes, no módulo 3, não deveria ser necessário explicá-lo, de novo, aqui. Entretanto, gostaríamos de fazer alguns comentários. Primeiro, ao ler esse código, lembre-se que startX e startY são parâmetros do método! Seus valores podem ser diferentes em cada chamada de drawGrid.

Em segundo lugar, observe a seguinte linha do código: y = y + 20; Isso talvez possa parecer completamente sem sentido: "como um número poderia ser igual a ele próprio mais 20?" Lembre-se que, em Java, a comparação de igualdade entre dois valores é feita por meio do operador '=='. O símbolo '=' tem significado diferente: indica uma atribuição de um determinado valor a uma variável – o comando acima deve ser lido como: 'atribua à variável y o valor resultante de somar 20 ao valor armazenado nessa variável anteriormente'. Por exemplo, se o valor de y, antes da execução desse comando, fosse 10, depois da execução do comando, o valor armazenado em y seria 30.

De modo geral, um comando de atribuição tem a forma: 'variável = expressão'. A avaliação desse comando consiste em avaliar a expressão e atribuir o valor resultante à

variável que aparece do lado esquerdo do operador de atribuição.

O terceiro comentário é que existe uma maneira mais sucinta de escrever o comando y = y + 20; em Java. Ele também pode ser escrito como y += 20;

O ultimo comentário refere-se ao comando lineCounter++;, O operador '++' indica que o valor da variável ao qual ele é aplicado deve ser 'incrementado de 1'. Portanto, se o valor de lineCounter for igual a 3 e o comando 'lineCounter++;' for executado, então lineCounter passará a ter o valor 4.

Depois de ter aprendido tudo isso, tente completar o código do método drawGrid, de maneira a traçar as linhas verticais da grade, como você já deve ter feito anteriormente. Se você tiver alguma dificuldade, confira o seu código com as versões de BattleshipApp e Board que apresentamosno código 10, da seção de códigos incluída no final deste tutorial.

Você pode fazer o download de todos os arquivos do projeto a partir do endereço http://see.photogenesis.com.br/lucilia/projetos/BattleshipSimple.zip.

Você pode também obter uma versão um pouco melhorada do projeto, a partir do endereço http://see.photogenesis.com.br/lucilia/projetos/battleship.zip. Essa versão tem algumas funcionalidades extras, com as quais você poderá brincar. Aproveite para dar uma olhadinha no código, e procurar entender as modificações introduzidas.

Bem, agora vamos rever o que aprendemos neste módulo.

Conclusão

O principal tópico abordado neste módulo foi o conceito de programação dirigida por eventos e, em particular, o tratamento de eventos do mouse e do evento de requisição de pintura de uma janela. Esses não são, entretanto, os únicos tipos de evento que podem ocorrer em componentes de interfaces gráficas. Outros exemplos são: digitar algo no teclado, evento de pressionar um botão, selecionar um item de menu, mover uma janela, arrastar as bordas de uma janela etc.

Cada um desses eventos é representado, em Java, por um objeto, de alguma subclasse da classe Event. Por exemplo, eventos do mouse são objetos da classe MouseEvent, eventos relacionados a janelas são objetos da classe WindowEvent e eventos do teclado são objetos da classe KeyboardEvent. De maneira geral, o tratamento de qualquer evento, envolve as seguintes etapas, que seguimos no nosso código para tratar o evento de clicar o mouse sobre a janela do jogo:

Identificar a classe do evento: essa é a classe dos objetos que representam o evento. Por exemplo, na nossa aplicação, o evento de interesse era clicar o mouse, que é um evento da classe MouseEvent, subclasse de Event. A classe MouseEvent define métodos tais como getX() e getY(), que podem ser usados para obter as coordenadas x e y, respectivamente, da localização do mouse na janela, quando o evento ocorrer.

Identificar a fonte do evento: esse é o componente que gera o evento e gerencia a criação do ouvinte do evento. Na nossa aplicação, esse componente é o objeto da classe BattleshipApp que representa a janela do jogo. Quando o usuário clica o mouse em sobre qualquer ponto dessa janela, esse evento é informado ao ouvinte especificado o evento. Como vimos, o ouvinte do evento de clicar o mouse é especificado no

construtor da classe BattleshipApp, por meio do comando this.addMouseListener(this);

Escolher a classe ouvinte do evento: essa é a classe que vai tratar o evento. Vimos que, no caso de eventos do mouse, essa classe deve implementar aos métodos da interface MouseListener. De maneira análoga, uma classe ouvinte de eventos de janela deve implementar os métodos da interface WindowListener, um ouvinte de eventos do teclado deve implementar métodos da interface KeyboardListener etc.
Essas classes que descrevem objetos relacionados a eventos estão definidas na biblioteca java.awt.event. Portanto, para incluir código relacionado a tratamento de evento, em uma determinada classe, é necessário inserir, no início do arquivo de código dessa classe, uma cláusula de importação da forma:

```
import Java.awt.event.*;
```

É tudo bastante simples, não é? Não se preocupe se você ainda não se sentir muito familiarizado com a idéia de programação dirigida por evento, ou com as classes pra tratamento de eventos, pré-definidas em Java. Teremos muitas oportunidades de lidar com eventos, ao continuar a programar nosso jogo de Batalha Naval.

Vamos continuar? No próximo módulo, vamos aprender como preencher os quadros do tabuleiro do jogo, como exibir mensagens de instruções... em resumo, vamos introduzir novas funcionalidades na interface do nosso jogo.

Módulo 6

Mais sobre arrays e comandos de repetição

Introdução

Neste módulo, vamos continuar a programar nossa aplicação, escrevendo, agora, o código correspondente ao estado de preparação do jogo - "Game Setup". Isso não é muito simples... Existem diversos conceitos novos envolvidos, que você terá que aprender:

- Como usar arrays de 2 dimensões
- Como usar loops "aninhados"
- Como usar um novo tipo de comando de repetição

Você também irá aprender como escrever o código para posicionar os navios de cada jogador no tabuleiro do jogo.

Vamos começar logo!

Limpando o código

Antes de começar a adicionar código novo ao nosso programa, vamos "limpar" um pouco o código já existente. O que queremos dizer com isso? Simples: vamos gastar algum adaptando o código a boas normas de programação, que irão facilitar futuras alterações ou correções que eventualmente tenham que ser feitas. Em particular, vamos substituir valores numéricos diretamente usados no código por constantes simbólicas.

"Constantes simbólicas"? O que significa isso? Espere, um exemplo vai esclarecer tudo. Se você examinar o código da sua classe BattleshipApp, encontrará o seguinte comando, no corpo do construtor da classe:

```
this.setSize(600, 500);
```

Os valores 600 e 500 especificam, respectivamente, a largura e a altura da janela do jogo, em pixels. Vamos substituir esses dois valores por constantes simbólicas, do seguinte modo:

```
public static final int GAME_WIDTH = 600;     // largura da janela do jogo
public static final int GAME_HEIGHT = 500;    // altura da janela do jogo
.
.
.
this.setSize(GAME_WIDTH, GAME_HEIGHT);    // especifica as dimensões da janela do
jogo
```

Porque nos preocuparmos com isso? Existem diversas razões. Primeiro, isso economiza tempo de programação. Vamos sempre definir constantes como GAME_WIDTH (LARGURA DO JOGO) e GAME_HEIGHT (ALTURA DO JOGO) no início do nosso arquivo. Desse modo, se for necessário alterar algum desses valores, bastará procurar no início do

arquivo e editar o valor correspondente. Usando o método anterior, seria necessário procurar ao longo do código, até encontrar setSize, para então alterar para o valor desejado. Em Segundo lugar – e mais importante – essa técnica reduz a probabilidade de erros, Suponha que existam referências à largura da janela do jogo em vários pontos do programa. O programa poderia ter, por exemplo, comandos como os seguintes:

```
public void paint(Graphics gfx) {
    gfx.setColor(Color.black);
   gfx.fillRect(0, 0, 600, 500);
    gfx.setColor(Color.red);
  int i = 0;
   while (i < 10) {
       gfx.drawLine(i * 600 / 10, 0, i * 600 / 10, 500);
  }
}
```

Imagine, agora, que você deseja modificar a largura do jogo, de 600 para 640 pixels. Você terá que encontrar toda ocorrência do número 600 e alterá-lo para 640. Mesmo que você faça isso corretamente, ainda assim poderá introduzir erros – particularmente se o valor 600 for usado para algum outro dado do programa, que não a largura do jogo, e esse valor for alterado acidentalmente.
Considere agora o mesmo código, usando constantes simbólicas:

```
public static final int GAME_WIDTH = 600;      // largura da janela do jogo
public static final int GAME_HEIGHT = 500;     // altura da janela do jogo
public static final int LINE_COUNT = 10;       // número de linhas da grade do tabuleiro
.
.
.
public void paint(Graphics gfx) {
    gfx.setColor(Color.black);
   gfx.fillRect(0, 0, GAME_WIDTH, GAME_HEIGHT);
    gfx.setColor(Color.red);
  int i = 0;
   while (i < LINE_COUNT) {
       gfx.drawLine(i * GAME_WIDTH / LINE_COUNT, 0, i *
                  GAME_WIDTH / LINE_COUNT, GAME_HEIGHT);
  }
}
```

Nesse caso, independentemente de quantas vezes GAME_WIDTH ou GAME_HEIGHT são usados no código, você poderá alterar o valor de todas essas ocorrências simultaneamente, bastando, para isso, modificar o valor definido para essas constantes simbólicas, no início do arquivo. Ainda gostaria de alterar a largura da janela do jogo, de 600 para 640? Não tem problema – faça apenas a seguinte modificação:

```
public static final int GAME_WIDTH = 600;
```

para:

```
public static final int GAME_WIDTH = 640;
```

Finalmente, como você certamente terá notado, usar constantes torna mais claro o código do programa. Por exemplo, o método paint parecia, acima, bastante abstrato – provavelmente você não teria adivinhado o que ele faz. A segunda versão desse método parece muito mais clara – é fácil ver que ele traça linhas, do topo para a base da janela, com comprimento constante, ao longo do eixo dos x, e regularmente espaçadas ao longo da direção vertical.

Esperando ter convencido você da utilidade de usar constantes simbólicas, vamos introduzir algumas em nosso código, na seção a seguir.

Introduzindo constantes

Vamos então adicionar as constantes simbólicas requeridas, no código do nosso projeto. Vamos começar pelo código da classe BattleshipApp. Se você não tem o código, pode obtê-lo a partir do endereço http://see.photogenesis.com.br/lucilia/projetos/BattleshipSimple.zip.

Primeiramente, vamos procurar identificar todos os comandos em que usamos valores numéricos diretamente, tais como 600 e 500 em this.setSize(600,500):
No construtor da classe BattleshipApp:

```
this.setSize(600, 500);
```

No método drawTitleScreen:

```
gfx.drawString("BATALHA NAVAL", 260, 225);
gfx.drawString("(clique o mouse para continuar)", 228, 275);
```

No método drawGrids:

```
this.redBoard.drawGrid(gfx, Color.red, 5, 5);
this.blueBoard.drawGrid(gfx, Color.blue, 255, 5);
```

Se você olhar no início do arquivo, verá as seguintes constantes já declaradas:

```
// variáveis de classe.
// estados do jogo
private static final int GAME_STATE_INTRO = 0;
private static final int GAME_STATE_SETUP = 1;
private static final int GAME_STATE_PLAYING = 2;
private static final int GAME_STATE_GAMEOVER = 3;
```

Portanto, vamos adicionar novas declarações a essa lista:

```
// variáveis de classe.
// estados do jogo
private static final int GAME_STATE_INTRO = 0;
private static final int GAME_STATE_SETUP = 1;
private static final int GAME_STATE_PLAYING = 2;
private static final int GAME_STATE_GAMEOVER = 3;

// dimensões da janela do jogo
private static final int GAME_WIDTH = 600;
private static final int GAME_HEIGHT = 500;
```

```
// posicionamento de mensagens na janela de abertura do jogo
private static final int TITLE_X = 260;
private static final int TITLE_Y = 225;
private static final int MOUSE_MSG_X = 228;
private static final int MOUSE_MSG_Y = 275;

// posicionamento das grades vermelha e azul no tabuleiro do jogo
private static final int RED_GRID_X = 5;
private static final int RED_GRID_Y = 5;
private static final int BLUE_GRID_X = 255;
private static final int BLUE_GRID_Y = 5;
```

Agora, vamos usar essas constantes em lugar dos respectivos valores, no código:

No constructor da classe BattleshipApp:

this.setSize(GAME_WIDTH, GAME_HEIGHT);

No método drawTitleScreen:

gfx.drawString("BATALHA NAVAL", TITLE_X, TITLE_Y);

gfx.drawString("(clique o mouse para continuar)", MOUSE_MSG_X, MOUSE_MSG_Y);

No método drawGrids:

this.redBoard.drawGrid(gfx, Color.red, RED_GRID_X, RED_GRID_Y);

this.blueBoard.drawGrid(gfx, Color.blue, BLUE_GRID_X, BLUE_GRID_Y);

Agora, vamos modiifcar o código da classe Board. Humm... as coisas são um pouco mais complicadas, nesse caso. Alguns dos valores constituem combinações de constantes. Vamos indicar as alterações a serem feitas no código, para depois explicar o significado das mesmas. Comece introduzindo as seguintes constantes:

```
// variáveis de classe.
private static final int ROW_COUNT = 10;              // número de linhas da grade
private static final int COLUMN_COUNT = 10;           // número de colunas da grade
private static final int ROW_PIXEL_HEIGHT = 20;       // espaçamento vertical entre as
linhas
private static final int COLUMN_PIXEL_WIDTH = 20;  // espaçamento horizontal entre as
linhas
```

Em seguida, modifique o método drawGrid do seguinte modo (as áreas alteradas são destacadas em azul):

```
/**
* Desenha a grade na janela do jogo.
*/
public void drawGrid(Graphics gfx,
                     Color gridColor,
                    int startX,
                    int startY) {
   // Especifica a cor das linhas.
    gfx.setColor(gridColor);

   // Traça linhas horizontais.
   int lineCounter = 1;
   int x = startX;
   int y = startY;
```

```
        while (lineCounter <= ROW_COUNT + 1) {
            gfx.drawLine(x, y, x + COLUMN_PIXEL_WIDTH * COLUMN_COUNT, y);
            y = y + ROW_PIXEL_HEIGHT;
            lineCounter++;
        }

        // Traça linhas verticais.
        lineCounter = 1;
        x = startX;
        y = startY;
        while (lineCounter <= COLUMN_COUNT + 1) {
            gfx.drawLine(x, y, x, y + ROW_PIXEL_HEIGHT * ROW_COUNT);
            x = x + COLUMN_PIXEL_WIDTH;
            lineCounter++;
        }
    }
```

Note como o usa das constantes torna o código mais claro. Observe o primeiro trecho de código destacado acima, logo abaixo do comentário que diz "traça as linhas horizontais". Podemos ver que o loop será executado enquanto lineCounter for menor ou igual ao número de linhas da grade mais 1. Portanto, como são 10 linhas, serão 11 iterações do loop.

Em seguida, podemos ver que a linha é traçada desde o ponto de coordenadas (x, y) até o ponto de coordenadas (x + COLUMN_PIXEL_WIDTH * COLUMN_COUNT, y). Em outras palavras, a linha horizontal terá comprimento igual ao número de colunas da grade vezes a largura de uma coluna, ou seja, a linha se estende ao longo de toda a grade.

Finalmente, observe que, em cada iteração do loop, a posição onde é traçada a linha horizontal é arrastada para baixo, uma quantidade igual à altura de uma linha: (y = y + ROW_PIXEL_HEIGHT)

Pense nessas idéias enquanto examina a grade que é traçada pelo código acima:

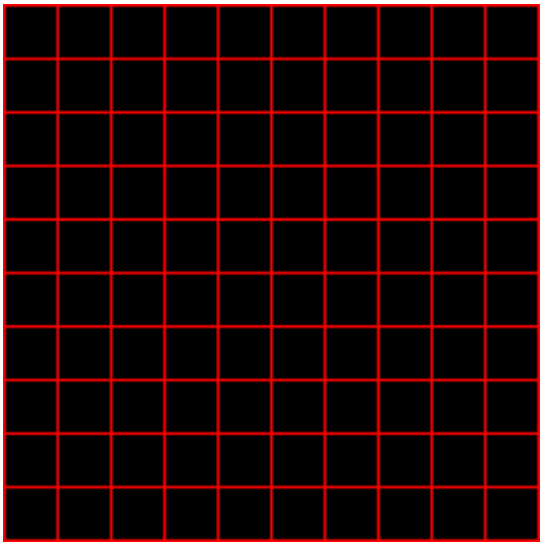


Figura 1 – Grade da Batalha Naval.

Note que existem 10 posições horizontais, mas 11 linhas horizontais, na grade. Note que essas linhas se estendem ao longo de toda a largura da grade. Finalmente, veja o espaçamento vertical entre as linhas horizontais. Se você quer ver o poder das constan-

tes, tente modificar os valores de ROW_COUNT, COLUMN_COUNT, ROW_PIXEL_HEIGHT, e COLUMN_PIXEL_WIDTH, e, em seguida, compilar e executar o programa novamente, para ver como a grade é modificada.

Se você tiver algum problema para compilar ou executar seu código corretamente, poderá verificá-lo, confrontando-o com o código 11, daseção de códigos incluída no final deste tutorial.

Uma vez que você tenha terminado de brincar, podemos começar a escrever o código correspondente ao modo de preparação do jogo.

Preparando o jogo

Estamos prontos para começar a programar a parte do código referente à "preparação do jogo". Você provavelmente já deve ter adivinhado que esse estado possibilita ao jogador posicionar sua esquadra no tabuleiro do jogo. Bem complicado, não é? Como vamos manter a informação sobre a posição de cada navio? Como saber se uma posição do tabuleiro contém um submarino, ou parte de um cruzador? Como um jogador pode posicionar um navio no tabuleiro?

Antes de começar a escrever algum código, temos que decidir como nossas classes vão interagir entre si. Como vamos posicionar navios no tabuleiro, as duas classes que temos que focalizar são, naturalmente, as classes Ship e Board (humm...talvez não seja assim tão óbvio, será?).

Nesta primeira parte, vamos fazer exatamente isso – listar as funcionalidades necessárias para prover suporte ao estado de preparação do jogo e traduzir isso na forma de requerimentos de código para nossas classes.

Preparando a Preparação do Jogo

Quando o jogo entra no estado de "preparação", várias ações devem ser realizadas:

1. Atribuir a cada jogador (usuário e computador) seus 5 navios (1 porta-aviões, 1 navio de guerra, 1 cruzador, 1 submarino, 1 bote).

2. Especificar a orientação de cada navio (para cima, para baixo, para a esquerda ou para a direita).

3. Determinar as posições dos navios no tabuleiro.

4. Desenhar os navios no tabuleiro.

Essas tarefas pressupõem algumas características de navios e do tabuleiro. Em primeiro lugar, cada navio deve ter um tipo (porta-aviões, cruzador etc), posição e orientação. Segundo, o tabuleiro deve ter uma maneira de manter informação sobre quais navios ocupam quais posições. Finalmente, ou um navio ou o tabuleiro deve saber desenhar os quadros do tabuleiro que são ocupados.

E então, como traduzir isso em código no código do programa?

Primeiramente, podemos ver que a classe Ship deve declarar várias variáveis: tipo do navio (porta-aviões, navio de guerra, cruzador, submarino ou bote), posição (especificada pelos números da linha e da coluna que ocupam no tabuleiro), orienta-

ção (para cima, para baixo, para a esquerda ou para a direita) e tamanho (5 para um porta-aviões, 4 para um navio de guerra, 3 para um cruzador, 2 para um submarino e 1 para um bote).
Usando nossa recente experiência com constantes como modelo, vamos adicionar ao código da classe Ship as seguintes declarações de constantes:

```
// variáveis de classe.
// tipo de um navio
public static final int TYPE_AIRCRAFT_CARRIER = 0;   // porta-aviões
public static final int TYPE_BATTLESHIP = 1;          // navio de guerra
public static final int TYPE_CRUISER = 2;            // cruzador
public static final int TYPE_SUBMARINE = 3;           // submarino
public static final int TYPE_PT_BOAT = 4;             // bote
// orientação de um navio
public static final int ORIENTATION_UP = 0;           // para cima
public static final int ORIENTATION_RIGHT = 1;        // para a direita
public static final int ORIENTATION_DOWN = 2;         // para baixo
public static final int ORIENTATION_LEFT = 3;         // para a esquerda
```

Vamos também adicionar variáveis de instância para manter informação sobre o tipo, a orientação, a posição e tamanho do navio:

```
// variáveis de instância.
private int type;
private int orientation;
private int row;
private int column;
private int size;
```

Nesse momento, você já deve ter adivinhado que vamos atribuir valores iniciais a essas variáveis de instância no construtor de objetos da classe Ship:

```
public Ship(int shipType, int dir, int row, int col, int length) {
    this.type = shipType;
    this.orientation = dir;
    this.row = row;
    this.column = col;
    this.size = length;
}
```

Note que os valores iniciais de todas as variáveis de instância de um navio soa determinados pelos valores dos parâmetros correspondentes do construtor. Em outras palavaras, podemos crier um navio, usando o operador new, do seguinte modo:

```
Ship sub = new Ship(Ship.TYPE_SUBMARINE, Ship.ORIENTATION_RIGHT,3,2,3);
```

A execução do comando acima irá resultar na criaçào de um objeto (da classe Ship) que representa um navio do tipo submarino (TYPE_SUBMARINE – definido como 3, acima), orientação para a direita (ORIENTATION_RIGHT (1)), posicionado na linha 3 e coluna 2 (row = 3 e col = 2) , e com tamanho 3 (size = 3), ou seja, o código do construtor será reduzido para:

```
this.type = 3;

this.orientation = 1;
```

```
this.row = 3;

this.col = 2;

this.size = 3;
```

Por enquanto, é só, para a classe Ship. Essa classe ainda não faz muita coisa, mas vamos adicionar outras funcionalidades, mais adiante.

E o tabuleiro?

Dissemos que o tabuleiro precisa manter informação sobre onde os navios estão posicionados. Como isso poderia ser feito? Bem, imagine o tabuleiro do jogo de Batalha Naval. Vemos uma grade, com buracos em cada quadro da grade, onde colocamos pinos de navios – isso faz com que o navio ocupe um conjunto de posições adjacentes na grade. Portanto, podemos imaginar o tabuleiro como uma coleção de células, cada uma das quais capaz de armazenar 1 pino de navio, como mostra a figura a seguir:

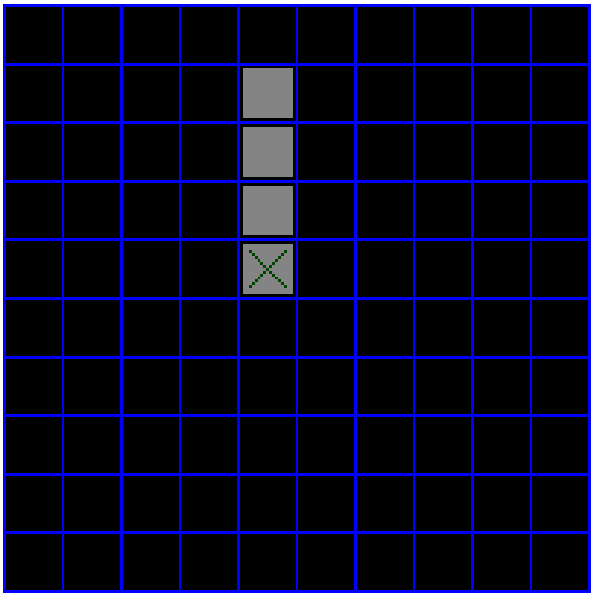


Figura 2—um Navio de Guerra posicionado no tabuleiro.

Vemos, na figura acima, um navio de Guerra (tamanho 4), com sua posição inicial na linha 4 e coluna 4 (lembre-se que Java numera linhas e posições de arrays a partir de 0), e com orientação "para cima". Vemos também que o tabuleiro contém 100 células (10 x 10) e quarto dessas células contêm partes de um navio de guerra. A próxima questão é, portanto, "como representar essas células no código do programa?"

Uma grande gama de opções

Uma maneira de representar isso seria introduzir uma variável inteira para cada célula da grade e armazenar um valor, em cada variável, para representar o tipo de navio que ocupa a célula. Por exemplo, poderíamos escrever, na classe Board, algo como o seguinte:

```
// variáveis de classe adicionais.
public static final int ID_EMPTY = 0;
public static final int ID_BATTLESHIP = 1;
public static final int ID_AIRCRAFT_CARRIER = 2;
public static final int ID_CRUISER = 3;
```

```
public static final int ID_SUBMARINE = 4;
public static final int ID_PT_BOAT = 5;

// variáveis de instância adicionais.
private int cell_00; // Row 0, column 0.
private int cell_01; // Row 0, column 1.
private int cell_02; // etc.
...
private int cell_10;
private int cell_11;
...
private int cell_97; // Row 9, column 7.
private int cell_98; // Row 9, column 8.
private int cell_99; // Row 9, column 9.
```

As células corresponderiam à grade da seguinte maneira:

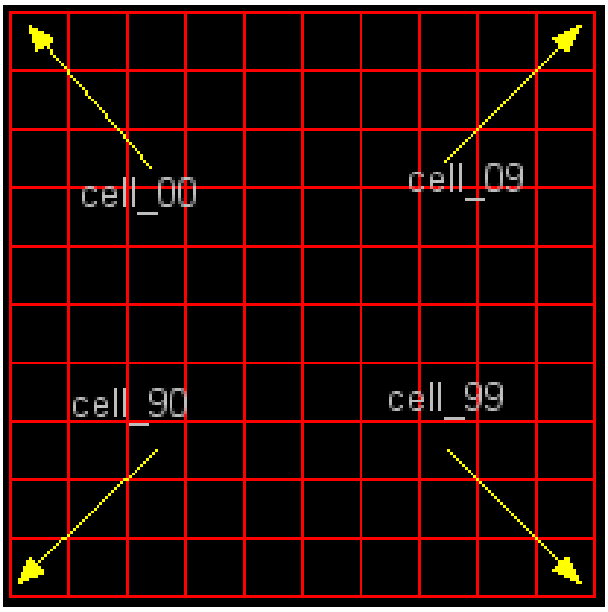


Figura 3— Correspondência entre as variáveis de célula e as posições na grade.

Isso funcionaria corretamente – mas dá um bocado de trabalho! Como você pode ver, teríamos que declarar 100 variáveis de célula! Seria muito melhor se pudéssemos comandar a criação dessas variáveis automaticamente.

Felizmente, existe uma maneira de fazer isso. Se você deverá lembrar-se que, no módulo 2 deste tutorial, falamos sobre arrays Como vimos, um Array em Java armazena uma coleção de valores, de um mesmo tipo, possibilitando referenciar cada um desses valores por meio de sua posição no Array. A seguir, vamos rever as idéias básicas sobre arrays e então usá-los para armazenar informação sobe as posições dos navios no tabuleiro do jogo.

Arrays de várias dimensões

Anteriormente, trabalhamos com Arrays unidimensionais – de uma dimensão. Isso é apenas uma maneira sofisticada de dizer uma seqüência de elementos que podemos referenciar por meio da sua posição na seqüência. Por exemplo, suponha que temos a seguinte lista de nomes:

Kirk, Spock, Scotty, Uhura

E queremos armazená-los em um Array unidimensional. Em Java, isso pode ser feito do seguinte modo:

```
private String[] names = {“Kirk”, “Spock”, “Scotty”, “Uhura”};
```

Lembre-se que, em Java, usamos os colchetes (‘[‘ e ‘]’), após um tipo, para indicar um array de elementos desse tipo – no exemplo acima String[] é o tipo de um array de strings (cadeias de caracteres). Se quisermos imprimir o terceiro elemento da lista acima, podemos usar o seguinte comando (lembre-se que Java numera os elementos de um array a partir de 0):

```
System.out.println(names[2]);
```

No caso da grade do nosso jogo de Batalha Naval, seria muito mais fácil se pudéssemos referenciar cada posição da grade por meio de dois índices: o número da linha e o número da coluna correspondentes. Como isso poder ser feito? A resposta é usar um Array de duas dimensões (ou bidimensonal). A declaração da variável gridCells (células da grade) como um array bidimensional de componentes inteiros seria feita do seguinte modo:

```
private int[][] gridCells;
```

Note como o tipo agora requer dois pares de colchetes – isso indica que gridCells é um array bidimensional. Note que a declaração acima apenas associa um nome de variável – gridCells – ao seu tipo – int[][] – mas não atribui nenhum valor inicial a essa variável – o que significa que é atribuído a ela, automaticamente, o valor null, indicando que ela não contém uma referência válida a nenhum objeto. Para atribuir um valor a gridCells, devemos criar um array com as dimensões desejadas, tal como no comando abaixo:

```
gridCells = new int[10][10];
```

Alternativamente, a atribuição de valor a gridCells poderia também ser feita na própria declaração da variável:

```
private int[][] gridCells = new int[10][10];
```

Agora que já criamos um array bidimensional, com as dimensões desejadas, precisamos saber como obter acesso a cada um dos seus componentes. Isso é fácil! Por exemplo, se quisermos armazenar um valor, digamos, 15, na terceira linha e primeira coluna do array, escreveríamos o seguinte comando:

```
this.gridCells[2][0] = 15;
```

De maneira análoga, para obter o valor armazenado na décima linha e décima coluna, escreveríamos:

```
int value = this.gridCells[9][9];
```

Fácil, não é? Então estamos prontos para ver como isso se aplica ao código do nosso jogo de Batalha Naval.

Preenchendo as células da grade do tabuleiro

Parece natural representar a grade do tabuleiro do jogo, na classe Board, pelo array bidimensional referenciado pela variável gridCells. Vamos inicializar essa variável no

construtor da classe Board, atribuindo a ela um array com as dimensões da grade do tabuleiro. Precisamos, também, definir um novo método, para posicionar um navio no tabuleiro. Por enquanto, não vamos nos preocupar com o código desse método.
Lembre-se que, no módulo anterior, declaramos também, na classe Board, um array unidimensional – fleet – para representar a esquadra de navios. A variável fleet será também inicializada no construtor da classe Board, atribuindo-se a ela um array de cinco componentes (sendo cada posição correspondente a um dos cinco tipos de navios). Traduzindo essas idéias em Java, teríamos o seguinte código para a classe Board:

```
// variáveis de classe.
private static final int ROW_COUNT = 10;                // número de linhas da grade
private static final int COLUMN_COUNT = 10;             // número de colunas da grade
private static final int ROW_PIXEL_HEIGHT = 20;         // espaçamento vertical entre as
linhas
private static final int COLUMN_PIXEL_WIDTH = 20;       // espaçamento horizontal entre
as linhas
private static final int SHIPS_PER_FLEET = 5;           // número de navios da esquadra

// variáveis de instância.
private RedPeg[] hitMarkers;          // pinos para marcar acertos
private WhitePeg[] missMarkers;       // pinos para marcar erros
private Ship[] fleet;                 // esquadra de navios
private int[][] gridCells;            // tabuleiro de um jogador

// Métodos.
/**
 * Construtor de objetos da classe Board
 */
public Board()
{
    this.fleet = new Ship[SHIPS_PER_FLEET];
    this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];
}

/**
* Posiciona um navio na grade do tabuleiro.
*/
public void addShip(Ship newShip) {
    // Inserir aqui o código para posicionar um navio na grade.
}
```

Note que o comentário em azul indica onde deve ser inserido o código para posicionar um navio na grade – vamos substituir esse comentário pelo código correspondente, mais adiante.

Quase pronto...

Já está quase pronto – descrevemos as características fundamentais do comportamento do tabuleiro e de um navio. Entretanto, falta ainda o código para "posicionar um navio na grade do tabuleiro". Para fazer isso, precisaremos aprender um pouco mais...

Antes de mais nada, precisamos definir como vamos representar os navios no tabuleiro. Note que a nossa grade, denotada pela variável gridCells, armazena valores inteiros.

Sabemos também que, a cada navio, corresponde um número inteiro, que indica o seu tipo. Uma solução simples seria casar essas duas idéias: cada posição na grade deve conter um número que indica o tipo do navio colocado nessa posição. Isso é ilustrado na figura a seguir:

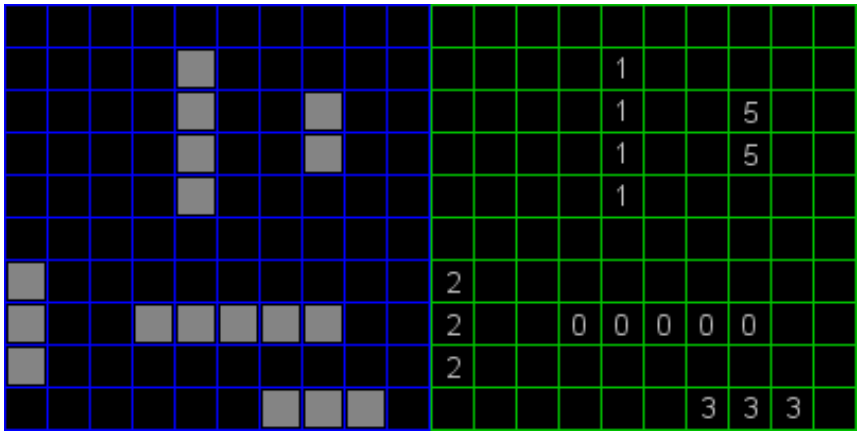


Figura 4— Armazenando navios na grade do tabuleiro.

Nesse diagrama, a grade em azul mostra a representação visual do tabuleiro e a grade em verde apresenta o conteúdo correspondente do array bidimensional gridCells.
Essa solução funciona corretamente, exceto por um inconveniente. Quando declaramos, em Java, uma variável do tipo int, ela é inicializada com o valor 0 (zero), caso nenhum valor seja atribuído a essa variável na sua declaração. O mesmo acontece quando criamos um array de valores inteiros – cada posição do array terá o valor 0, caso nenhum outro valor seja atribuído a essa posição. Ou seja, quando o comando a seguir é executado,

```
this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];
```

é atribuído o valor 0 a cada uma das posições do array gridCells. Note que usamos esse mesmo valor para especificar o tipo porta-aviões. Isso gera um problema, porque então não conseguimos diferenciar uma posição vazia de outra em que foi colocado um porta-aviões.

Existem duas possíveis soluções para contornar essa dificuldade. A primeira seria simplesmente modificar o valor utilizado para especificar o tipo de um porta-aviões, adotando algum valor diferente de zero. A segunda seria atribuir a cada posição da grade, explicitamente, um valor que indique posição vazia – diferente dos que utilizamos para os tipos de navios. Vamos preferir essa segunda solução – você verá que ela torna o código mais claro, e menos sujeito a erros, no caso de eventuais modificações.

Escrevendo código...

Estamos agora prontos para escrever um pouco de código. Lá vai: vamos apresentar o código pronto e explicá-lo passo a passo, pois uma boa maneira de começar a aprender a programar é procurando entender código já escrito. Você verá que já aprendeu bastante: não será necessário explicar a maior parte do código que vamos apresentar, para que você entenda o que está acontecendo.
Primeiro, vamos introduzir, na classe Ship, um novo tipo de navio, que representa, de fato, uma posição vazia na grade do tabuleiro (isto é, uma posição que não está ocupada por nenhum tipo de navio):

```
public static final int TYPE_NONE = -1;                  // nenhum navio
public static final int TYPE_AIRCRAFT_CARRIER = 0;       // porta-aviões
public static final int TYPE_BATTLESHIP = 1;             // navio de guerra
public static final int TYPE_CRUISER = 2;                // cruzador
public static final int TYPE_SUBMARINE = 3;              // submarino
public static final int TYPE_PT_BOAT = 4;                // bote
```

Na classe Ship, além do construtor vamos precisar de alguns outros métodos, que retornam características de um navio: o tipo do navio – getType; a sua orientação no tabuleiro – getOrientation; o seu tamanho – getSize; e as coordenadas da sua posição no tabuleiro – getRow (linha) e getColumn (coluna):

```
public int getType() {
   return this.type;
}

public int getOrientation() {
   return this.orientation;
}

public int getSize() {
   return this.size;
}

public int getRow() {
   return this.row;
}

public int getColumn() {
   return this.column;
}
```

Métodos como esses, que simplesmente retornam o valor de uma determinada variável de instância de um objeto, são bastante comuns em programas Java. A idéia é "encapsular" o uso dessas variáveis, declarando-as como privadas e possibilitando acesso a seus valores apenas por meio de chamadas de métodos "get" correspondentes. A vantagem de se fazer isso é evitar que alterações na representação de uma determinada característica de um objeto impliquem em alterações na parte do código do programa que usa esse objeto.

Note que o valor retornado, em cada método, é indicado por meio da palavra reservada return. De modo geral, para que um método retorne um valor, deve ser incluído, no corpo desse método, um comando na forma return <expressão>; - o valor retornado por uma chamada do método é igual ao valor obtido pela avaliação da <expressão>.

Continuando com o nosso programa, vamos agora modificar o construtor da classe Board, inserindo aí o código para preencher todas as posições da grade do tabuleiro, com o valor que representa posição vazia:

```
/**
* Construtor de objetos da classe Board
*/
public Board()
{
    this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];
```

```
    // Preenche as células da grade com vazio.
    int i = 0;
    while (i < ROW_COUNT) {
        int j = 0;
        while (j < COLUMN_COUNT) {
            this.gridCells[i][j] = Ship.TYPE_NONE;
            j++;
        }
        i++;
    }
}
```

O código acima envolve o que chamamos de "loops aninhados". Opa! Não aprendemos sobre isso ainda! No código acima, a idéia é simples: o loop mais externo itera sobre linhas e o loop mais interno, sobre colunas. Assim, para cada linha j, do loop externo, percorrem-se todas as colunas dessa linha, no loop mais interno. Portanto, quando a execução do loop mais externo tiver sido concluída, terão sido percorridas todas as células da grade.

Se você quiser saber mais sobre loops aninhados, consulte o apêndice 1 deste módulo. Você poderá também preferir seguir em frente, continuando com o nosso código, e aprender um pouco mais sobre loops, mais tarde.
Finalmente, vamos adicionar código para posicionar um navio na grade do tabuleiro:

```
/**
* Posiciona um navio na grade do tabuleiro
*/
public void addShip(Ship newShip) {
    // Obtém a posição onde deve ser colocado o navio
    int row = newShip.getRow();            // linha
    int col = newShip.getColumn();          // coluna
    // Obtém a orientação do navio
    int orientation = newShip.getOrientation();
    int i = 0;

    // Adiciona o navio à esquadra.
    this.fleet[newShip.getType()] = newShip;

    // Armazena o navio na grade gridCells
    if (orientation == Ship.ORIENTATION_UP) {          // Orientação para cima
        while (i < newShip.getSize()) {
            this.gridCells[row - i][col] = newShip.getType();
            i++;
        }
    }
    else if (orientation == Ship.ORIENTATION_RIGHT) {   // Orientação para a direita
        while (i < newShip.getSize()) {
            this.gridCells[row][col + i] = newShip.getType();
            i++;
        }
    }
    else if (orientation == Ship.ORIENTATION_DOWN) {   // Orientação para baixo
        while (i < newShip.getSize()) {
```

```
            this.gridCells[row + i][col] = newShip.getType();
          i++;
        }
    }
    else {
        // A orientação, nesse caso, é para a ESQUERDA.
        while (i < newShip.getSize()) {
            this.gridCells[row][col - i] = newShip.getType();
          i++;
        }
    }
}
```

Em lugar de explicar como o código acima funciona, vamos discutir o seu comportamento no caso do navio representado na figura 2. Rather than explicitly explain how this code works, we'll discuss its behavior for the case of the ship shown in figure 2. Observando essa figura, vemos que o jogador posicionou um navio de Guerra na linha 4, coluna 4, com orientação para cima. Se o método adddShip for chamado tendo como argumento um navio (objeto da classe Ship) de tipo TYPE_BATTLESHIP, com row = 4, column = 4, e orientation = ORIENTATION_UP, o trecho inicial do corpo do método, ou seja:

```
int row = newShip.getRow();
int col = newShip.getColumn();
int orientation = newShip.getOrientation();
int i = 0;
```

fica reduzido ao seguinte código:

```
int row = 4;
int col = 4;
int orientation = 0;
int i = 0;
```

Portanto, será executado o primeiro ramo do comando if...else (correspondente a orientação para cima). Esse ramo do comando if...else executa o seguinte loop while:

```
while (i < newShip.getLength()) {
    this.gridCells[row - i][col] = newShip.getType();
   i++;
}
```

onde i começa com o valor 0, e newShip.getLength() retorna 4 (porque um navio e guerra tem tamanho igual a 4). Como a linha e a coluna da posição inicial do navio são ambas 4, o código acima se reduz a :

```
while (i < 4) {
    this.gridCells[4 - i][4] = 1;
   i++;
}
```

Quando i=0, o comando que ocorre no corpo do comando while se reduz a:

```
this.gridCells[4 - 0][4] = 1;
```

De maneira análoga, quando i=1, 2 e 3, esse comando fica reduzido a:

```
this.gridCells[4 - 1][4] = 1;
this.gridCells[4 - 2][4] = 1;
this.gridCells[4 - 3][4] = 1;
```

Em outras palavras, a execução do loop armazena o valor '1' nas seguintes células da grade: (4, 4), (3, 4), (2, 4) e (1, 4) – que é exatamente o que desejamos, de acordo com a figura 2.
Outro ponto a ser notado, nesse código, é:

```
// Adiciona o navio à esquadra.
this.fleet[newShip.getType()] = newShip;
```

Isso armazena um novo navio – um objeto da classe Ship – no array que representa a esquadra de navios – fleet. Isso não tem nenhum efeito sobre gridCells, mas possibilita recuperar informação sobre o navio que está colocado em cada posição do tabuleiro. Essa informação será necessária mais tarde, quando formos escrever o código correspondente a jogadas de cada jogador.

Por enquanto é só, em relação à questão de armazenar informação sobre o posicionamento de navios no tabuleiro do jogo. Sugerimos que você reveja o código acima, verificando os demais casos, correspondentes à orientação do navio para baixo, para a direita e para a esquerda, certificando-se de compreender o que acontece.

Desenhando o navio no tabuleiro

Por fim, precisamos escrever o código para desenhar navios no tabuleiro. Já definimos um método para desenhar uma grade do tabuleiro – o método drawGrid – na classe Board. Parece natural adicionar, também nessa classe, o método para desenhar os navios na grade.

Antes de escrever o código, vamos pensar como ele deve ser. Observando novamente a figura 2, vemos que um navio deverá ser desenhado, na grade do tabuleiro, como uma seqüência de quadrados de cor cinza, cada qual preenchendo uma célula da grade. Poderíamos escolher um desenho mais bonito, mas, por enquanto, é melhor manter essa opção mais simples.

Quais seriam os passos do método para desenhar os navios na grade? Esses passos são bastante simples:

1. Comece pela primeira linha da grade.
2. Na linha corrente, comece pela primeira coluna.
3. Se a célula na linha e coluna correntes não está vazia, desenhe nela um quadrado cinza.
4. Mova para a coluna seguinte.
5. Se não tiverem sido percorridas todas as colunas da linha, volte ao passo 3.
6. Se todas as colunas já tiverem sido percorridas, mova para a próxima linha.
7. Se não tiverem sido percorridas todas as linhas, volte para o passo 2.
8. Se todas as linhas já tiverem sido percorridas, termine.

Vamos agora escrever o código correspondente, em Java. Para isso, vamos introduzir um novo tipo de comando de repetição – chamado do loop. Você pode aprender mais sobre esse tipo de comando, consultando o apêndice 2 deste módulo.
Aqui está o código:

```
/**
* Desenha os navios na grade do tabuleiro
*/
public void drawShips(Graphics gfx, Color shipColor,
                int startX, int startY) {
   // Especifica a cor dos navios.
    gfx.setColor(shipColor);
   int row = 0;    // Inicia na primeira linha.
   do {

      int col = 0; // Inicia na primeira coluna.
      do {

         // A célula está vazia?
          if (this.gridCells[row][col] != Ship.TYPE_NONE) {
             // Não—a célula contém parte de um navio.

            // Calcula a posição inicial da célula.
             int x = startX + col * COLUMN_PIXEL_WIDTH;
             int y = startY + row * ROW_PIXEL_HEIGHT;

             // Desenha um quadrado pouco menor que a célula.
            gfx.fillRect(x + 2, y + 2,
                     COLUMN_PIXEL_WIDTH - 4,
                     ROW_PIXEL_HEIGHT - 4);
        }

          col++;  // Move para a próxima coluna.
      } while (col < COLUMN_COUNT);

      row++;      // Move para a próxima linha.
   } while (row < ROW_COUNT);
}
```

Note que novamente usamos loops aninhados. Se você ainda não leu o apêndice que trata dete assunto, é um bom momento para faze-lo: clique aqui.
Agora, vamos explicar código acima, detalhadamente. Note como os comentários (em cor verde) correspondem aos passos que definimos anteriormente para o método de desenhar os navios – você deverá ser capaz de entender o código por meio desses comentários. Devemos apenas observar o seguinte: o método fillRect, pré definido na classe Graphics, desenha e preenche um retângulo, e tem a seguinte assinatura:

```
void fillRect(int upperLeftX, int upperLeftY, int width, int length);
```

No código acima, x e y representam as coordenadas (em pixels) do canto superior esquerdo da célula corrente. Portanto, x+2 e y+2 definem um ponto que está localizado 2 pixels à direita e 2 pixels abaixo do canto superior esquerdo da célula. Queremos deixar uma margem de 2 pixels entre a borda da célula e o quadrado cinza que vamos desenhar no interior da mesma. Isso significa que a largura desse quadrado seria 4

pixels menor que a largura da célula (COLUMN_PIXEL_WIDTH) e sua altura seria 4 pixels menor que a altura da célula (ROW_PIXEL_HEIGHT). Usando esses valores, obtemos os argumentos da chamada ao método fillRect:

```
fillRect(x + 2, y + 2, COLUMN_PIXEL_WIDTH - 4, ROW_PIXEL_HEIGHT - 4);
```

Agora, só falta chamar esse método para desenhar os navios na grade do tabuleiro. Isso é simples – basta adicionar os seguintes commandos no corpo do método drawGrids, da classe BattleshipApp:

```
this.redBoard.drawShips(gfx, Color.gray, RED_GRID_X, RED_GRID_Y);
this.blueBoard.drawShips(gfx, Color.gray, BLUE_GRID_X, BLUE_GRID_Y);
```

Bem, terminamos, por enquanto, as novas classes Board e Ship. nossas. Fizemos um bocado de modificações. Vamos então tentar compilar o nosso programa, para ver se tudo funciona corretamente. Se algo der errado, confira seu código com o código 12 da seção de códigos incluída no final deste tutorial.

Testando, Testando, 1, 2, 3...

O código que adicionamos neste módulo simplesmente adiciona algumas funcionalidades básicas ao jogo, mas de fato não faz nada por si próprio. Em tese, agora podemos adicionar navios ao tabuleiro do jogo e vê-los aparecer na tela. Precisamos dessa funcionalidade para possibilitar ao jogador e ao computador posicionarem seus navios no tabuleiro. Entretanto, ainda não sabemos se o código que produzimos funciona corretamente. Poderíamos ignorar esse problema potencial e seguir em frente – mas isso não é uma boa idéia. Considere o que aconteceria se você detectasse um erro, ao testar o código que irá escrever para possibilitar ao jogador posicionar seus navios. Você não saberia se o problema se origina do código de posicionamento ou do código subjacente, que acabamos de escrever, para armazenar os navios na grade e desenhá-los na janela do jogo.

A moral da estória é: teste seu código sempre que puder, mesmo que isso signifique "inventar" dados que você terá que remover posteriormente!

No nosso caso, teremos que criar alguns "navios de teste" e posicioná-los em determinadas células da grade. Isso nos permitirá ver se eles estão sendo armazenados corretamente no array que representa a grade, e se estão sendo desejados na posição correta, no tabuleiro apresentado na janela do jogo. Depois disso, iremos apagar esse código de teste e começar a escrever o código que permite aos jogadores posicionarem os seu navios – mas isso é assunto para o próximo módulo deste tutorial.

Escrever código apenas para testar outro código pode parecer perda de tempo. Mas, acredite: isso de fato economiza um bocado de tempo, em tarefas subseqüentes.

Para o nosso teste, vamos usar o diagrama apresentado na figura 4:

- Porta-aviões: tamanho 5, linha 7, coluna 3, orientação para a direita.
- Navio de guerra: tamanho 4, linha 1, coluna 4, orientação para baixo.
- Cruzador: tamanho 3, linha 8, coluna 0, orientação para cima.
- Submarino: tamanho 3, linha 9, coluna 8, orientação para a esquerda.
- Bote: tamanho 2, linha 2, coluna 7, orientação para baixo.

Vamos então adicionar, ao construtor da classe Board, o código para criar nossa esquadra de teste:

```
public  Board()
{
    this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];

   // Preenche as células da grade com vazio.
   int i = 0;
   while (i < ROW_COUNT) {
      int j = 0;
      while (j < COLUMN_COUNT) {
          this.gridCells[i][j] = Ship.TYPE_NONE;
        j++;
      }
      i++;
   }

   // Cria uma esquadra de teste.
   // REMOVA ESTE TRECHO DE CÓDIGO DEPOIS QUE O TESTE FOR CONCLUÍDO!
   Ship testShip;
    testShip = new Ship(Ship.TYPE_AIRCRAFT_CARRIER,
                  Ship.ORIENTATION_RIGHT, 7, 3, 5);
    this.addShip(testShip);
    testShip = new Ship(Ship.TYPE_BATTLESHIP,
                  Ship.ORIENTATION_DOWN, 1, 4, 4);
    this.addShip(testShip);
    testShip = new Ship(Ship.TYPE_CRUISER,
                  Ship.ORIENTATION_UP, 8, 0, 3);
    this.addShip(testShip);
    testShip = new Ship(Ship.TYPE_SUBMARINE,
                  Ship.ORIENTATION_LEFT, 9, 8, 3);
    this.addShip(testShip);
    testShip = new Ship(Ship.TYPE_PT_BOAT,
                  Ship.ORIENTATION_DOWN, 2, 7, 2);
    this.addShip(testShip);
}
```

Temos agora uma esquadra de teste!
Note como testShip é usada no código acima – é o que chamamos de uma variável temporária. Ela está sendo usada apenas para armazenar cada novo navio criado, para que ele seja passado como argumento na chamada do método addShip. Ela não trem nenhum outro uso. De fato, poderíamos eliminar essa variável completamente, pela combinação dos comandos acima, do seguinte modo:

```
this.addShip(new  Ship(Ship.TYPE_PT_BOAT,
                Ship.ORIENTATION_DOWN, 2, 7, 2));
```

Entretanto, usar a variável testShip torna o código mais claro e legível.

Finalmente, poderemos executar nosso programa e ver como ele funciona. Compile todo o programa e execute-o. Se tudo correr bem, deverão ser exibidas as duas grades – azul e vermelha – do tabuleiro, ambas tendo navios posicionados como no diagrama da figura 4.

Quando você executar o programa, provavelmente notará um pequeno erro – os quadrados cinzas usados para representar partes de um navio são 1 pixel menor do que o desejado, tanto na direção x como na direção y. Isso é fácil de corrigir – basta alterar a chamada do método fillRect, no corpo do método drawShips da classe Board, para o seguinte:

```
gfx.fillRect(x + 2, y + 2,
        COLUMN_PIXEL_WIDTH - 3,
        ROW_PIXEL_HEIGHT - 3);
```

Compile e execute o seu programa novamente, e veja se tudo funciona corretamente.

Uma vez que tudo funcione direitinho, você pode brincar um pouco com o código. Que tal modificar as posições dos navios da sua esquadra de teste? Ou alterar a cor dos navios, de Color.gray para alguma outra (Color.pink?). Divirta-se.

Quando você tiver terminado, remova o código para criar a esquadra de teste. Vamos agora rever o que aprendemos neste módulo

Conclusão

Neste módulo, adicionamos ao nosso programa algumas funcionalidades básicas, para nos permitir programar, nos próximos módulos, o código que possibilita aos jogadores posicionarem sua frota de navios no jogo. Essas funcionalidades foram:

Armazenar no tabuleiro informação sobre os navios nele colocados

Desenhar os navios nas células da grade do tabuleiro exibido na janela do jogo

Para fazer isso, usamos arrays de uma e duas dimensões e lidamos com loops aninhados. Aprendemos também um novo tipo de comando de repetição – do loop.

Prepare-se para adicionar "um pouco de inteligência" ao seu programa: no próximo módulo, o computador irá posicionar seus navios no tabuleiro do jogo!

Apêndice 1 – loops aninhados

Ao desenvolver programas, freqüentemente nos deparamos com situações que requerem realizar uma tarefa repetidas vezes, sobre elementos de uma determinada seqüência.Se você parar para pensar um pouco, verá que isso envolve pelo menos dois loops: um para a repetição da tarefa e outro para percorrer os elementos da seqüência. Na maioria das vezes, a solução toma a forma de "loops aninhados".

O que é um loop aninhado? Simples, é um loop dentro de outro loop. Se isso parece confuso, um exemplo simples poderá tornar tudo mais claro. Suponha que você deseja calcular a pontuação de jogadores, em um campeonato de um determinado jogo. Considere uma lista de 5 jogadores: "Tarzã", "Batman", "Mandraque", "Hulck" e "Fantomas", cada qual com pontos em 3 rodadas. Queremos calcular o número médio de pontos por rodada, de cada jogador.
Suponha que definimos a seguinte classe:

```
/**
* Tournament – Representa um jogador do campeonato
```

```
*/
public class TournamentEntry() {
   // variáveis de instância.
   private String name;        // nome do jogador
   private int[] scores;        // pontuações do jogador
   private int average;        // média de pontos

   // Construtor de objetos da classe.
   // Inicialize todas as variáveis aqui.
   public TournamentEntry(String name, int score1, int score2, int score3) {
      this.name = name;
      this.scores = new int[3];
      this.scores[0] = score1;
      this.scores[1] = score2;
      this.scores[2] = score3;
   }

   // Retorna o nome do jogador.
   public String getName() {
      return this.name;
   }

   // Retorna a i-ésima pontuação do jogador.
   public int getScore(int i) {
      return this.scores[i];
   }

   // Atribui a media de pontos.
   public void setAverage(int average) {
      this.average = average;
   }
}
```

Vamos também supor que são criados os cinco jogadores, em algum lugar do programa, usando o seguinte código:

```
private TournamentEntry[] entries;

public void initEntries() {
   // Cria um array de jogadores.
   this.entries = new TournamentEntry[5];
   // Preenche esse array com os dados dos jogadores.
   entries[0] = new TournamentEntry("Tarzã", 98, 99, 100);
   entries[1] = new TournamentEntry("Batman", 100, 100, 75);
   entries[2] = new TournamentEntry("Mandrake", 88, 92, 87);
   entries[3] = new TournamentEntry("Hulck", 100, 80, 90);
   entries[4] = new TournamentEntry("Fantomas", 92, 97, 85);
}
```

Agora suponha que você deseja calcular a pontuação media de cada jogador. Bem é fácil escrever um trecho de código para calcular a media de um único jogador, por exemplo, Tarzã:

```
// Calcula a media (average) do jogador 0 – Tarzã
float average = 0;
```

```
int i = 0;
while (i < 3) {
    average = average + entries[0].getScore(i);
   i++;
}
average = average / 3;
```

Poderíamos repetir esse código 5 vezes – uma vez para cada jogador – mas isso seria uma enorme perda detempo. Sabemos que, sempre que temos que descrever a repetição de uma tarefa, usar um loop facilita muito. Nesse caso, queremos um loop que percorra os cinco jogadores do jogo e calcule a média de pontos, em cada caso. O código deverá ser parecido com o seguinte:

```
int j = 0;
while (j < 5) {
    // Calcule a media do j-ésimo jogador.
   j++;
}
```

Agora, tudo o que temos a fazer é inserir o código para calcular a média, no local indicado:

```
int j = 0;
while (j < 5) {
    float average = 0;
   int i = 0;
   while (i < 3) {
       average = average + entries[j].getScore(i);
      i++;
   }
    average = average / 3;
   j++;
}
```

Nesse caso, o loop mais externo é mostrado em azul, e o mais interno, em laranja. Note que o contador do loop mais interno (i) controla as pontuações do jogador cuja media este sendo calculada. O contador do loop mais externo (j) controla o jogador corrente. Considere, cuidadosamente, como o computador executa esse código. Primeiro, ele atribui o valor 0 a j, em seguida inicia a execução do loop mais externo. A média de pontos é inicializada com o valor 0, e também o contador do loop mais interno – i. Os valor armazenado em i passa a ser 1, depois 2, e finalmente 3 – o que faz com que a condição do loop se torne falsa, o que termina o loop mais interno. Em resumo, a execução do loop mais interno, durante a primeira iteração do loop mais externo, se resume à execução da seguinte seqüência de comandos:

```
average = average + entries[0].getScore(0);
average = average + entries[0].getScore(1);
average = average + entries[0].getScore(2);
```

Depois que o loop mais interno é completado, o contador do loop mais externo – j – é incrementado, e a execução recomeça, outra vez, testando a condição do loop mais externo. O loop mais interno é novamente executado, dessa vez resumindo-se à execu-

ção da seguinte seqüência de comandos:

```
average = average + entries[1].getScore(0);
average = average + entries[1].getScore(1);
average = average + entries[1].getScore(2);
```

Depois disso, o valor de j é incrementado para 2 e o processo se repete. O mesmo acontece para j igual a 3 e j igual a 4. Finalmente, quando j é igual a 5, a condição do loop mais externo se torna falsa, e esse loop termina.

Loops aninhados são uma ferramenta poderosa e fácil de usar – uma vez que você pratique um pouco.

Apêndice 2 – do loops

Existem diversas formas de comandos de repetição, em Java. Nosso velho amigo é o while loop. Você provavelmente já entende melhor como ele funciona. Por exemplo, se você quisesse imprimir uma mensagem, no console Java, 10 vezes, escreveria o seguinte comando while:

```
int i = 0;
while (i < 10) {
    System.out.println("Minha mensagem aqui.");
   i++;
}
```

O comando do loop é apenas uma outra maneira de se descrever a repetição de uma tarefa. Conceitualmente, ele é ligeiramente diferente do while loop. Lembre-se de como interpretamos o while loop em termos da nossa liguagem natural: "enquanto uma dada condição é verdadeira, realize uma determinada tarefa". O comando do loop, por outro lado, pode ser parafraseado do seguinte modo: "realize uma determinada tarefa, enquanto uma condição é verdadeira."
Esse rearranjo afeta o código, tornando-o ligeiramente diferente, quando reescrito com o comando do while:

```
int i = 0;
do {
    System.out.println("Minha mensagem aqui.");
   i++;
} while (i < 10);
```

Posso ouvi você dizendo: "Se esses dois commandos são apenas duas maneiras diferentes de dizer a mesma coisa, porque ter os dois? Porque não usar apenas um deles, e jogar o outro fora? ". Existem duas razões. A primeira – mais prática – é que eles se comportam de maneira ligeiramente diferente, em alguns casos. Por exemplo, suponha que, no código acima, em lugar de atribuir o valor 0 a i inicialmente, começamos com o valor 10. O while loop ficaria assim:

```
int i = 10;
while (i < 10) {
    System.out.println("Minha mensagem aqui.");
   i++;
```

```
}
```

Nesse caso, a mensagem nunca será impressa, porque a condição (i < 10) é testada no início da execução do comando e, sendo falsa, o corpo do while nunca é executado. Considere agora como o do loop se comporta, sob as mesmas condições iniciais:

```
int i = 10;
do {
    System.out.println("Minha mensagem aqui.");
   i++;
} while (i < 10);
```

Na execução desse comando, primeiro é executado o corpo do loop, imprindo então a mensagem e incrementando o contador do loop, para depois ser testada a condição de terminação – nesse caso, ela será falsa, uma vez que o valor de i será igual a 11. Portanto, o loop termina depois de imprimir a mensagem 1 vez.

A segunda razão para a existência dos dois tipos de comando é que algumas vezes é mais fácil pensar em termos de um do que do outro. Por exemplo, para o caso de "imprimir a mensagem 10 vezes", você poderia pensar na solução da seguinte maneira:

1. Inicialize o contador com o valor 0.
2. Se o contador for menor que 10, imprima a mensagem.
3. Incremente o contador de 1.
4. Volte ao passo 2.

(o que corresponde ao while loop)

ou da seguinte maneira:

1. Inicialize o contador com o valor 0.
2. Imprima a mensagem.
3. Incremente o contador de 1.
4. Se o contador for menor que 10, volte ao passo 2.

(o que corresponde ao do loop).

Nenhuma das soluções é melhor que a outra, mas, dependendo de como o seu cérebro funciona, você poderá achar uma mais confortável do que a outra.

Módulo 7

Mais sobre Interfaces Gráficas e Tratamento de Eventos

Introdução

Neste módulo, vamos desenvolver a parte do código que possibilita aos dois jogadores – o computador e o usuário – posicionarem suas esquadras no tabuleiro do jogo. Os dois casos são bastante diferentes.

No caso do computador, as posições dos navios serão escolhidas aleatoriamente. Para escrever o código que implementa isso, você terá que aprender a lidar com números aleatórios em Java.

No caso do usuário, o programa terá que permitir que ele indique a posição escolhida para cada navio, clicando sobre uma célula da grade do tabuleiro. Além disso, o programa terá que oferecer uma interface para que ele especifique o tipo do navio a ser colocado na posição indicada. Para isso, vamos usar, novamente, a idéia de programação dirigida por eventos.

Além disso, neste módulo você vai também introduzir, no jogo, mensagens de instrução para o jogador.

Pronto para começar? Então, lá vamos nós...

Esquadra do computador

Nesta seção, vamos escrever código para permitir ao computador posicionar seus navios no tabuleiro. Como isso envolve a "tomada de decisões pelo computador", esse código é, tecnicamente, inteligência artificial. Você não tinha idéia de que também aprenderia um pouco sobre um tópico tão avançado, não é?

Para ser honesto, o código não será assim tão inteligente – afinal é a primeira vez que estamos programando um jogo e, portanto, não devemos esperar nada assim tão genial. Basicamente, a posição de cada navio no tabuleiro será escolhida aleatoriamente, assim como a sua orientação. Se o navio couber na posição escolhida, ótimo – usamos esse espaço. Caso contrário, é escolhida, aleatoriamente, uma nova posição.

Vamos começar delineando os passos do procedimento para fazer isso. Depois, vamos converter essas idéias em código Java.

Escolhendo uma posição e verificando se é adequada

O algoritmo para posicionamento dos navios no tabuleiro, pelo computador, pode ser resumido nos seguintes passos:

1. Comece pelo primeiro navio (porta-aviões).
2. Escolha, aleatoriamente, uma linha, uma coluna e uma orientação para o navio.

3. Teste se o navio pode ser posicionado no tabuleiro, na linha e coluna escolhidas e com a orientação escolhida.

4. Se isso ocorrer, posicione o navio aí e passe para o próximo navio; caso contrário, retorne para o passo 2.

5. Se os cinco navios já tiverem sido posicionados, termine. Caso contrário, volte para o passo 2.

Os dois primeiros passos do algoritmo bastante simples (não se importe com essa questão de escolha aleatória – já vamos ver como se faz isso). Passemos então ao passo 3: "Teste se o navio pode ser posicionado no tabuleiro, na linha e coluna escolhidas e com a orientação escolhida ". Como isso funciona exatamente?

Dada uma posição na grade, e uma orientação, existem duas situações nas quais o navio não poderia ser posicionado da maneira indicada. A primeira é quando o navio excede os limites da grade. Por exemplo, suponha que você tenta posicionar um navio de guerra, na célula (0, 0), com orientação para cima (veja a figura 1).

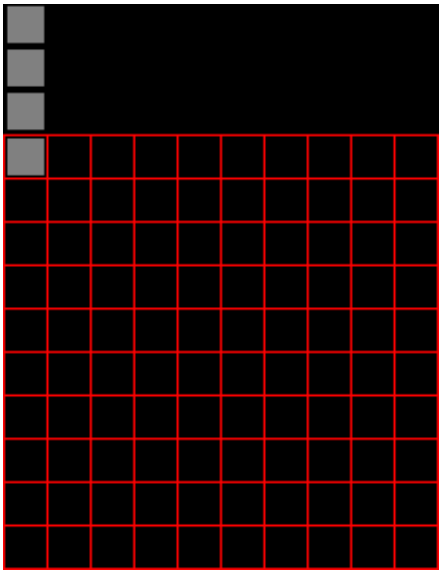


Figura 1— Colocando um navio de guerra em uma posição inadequada.

A segunda situação é quando o navio "colide" com outro navio já posicionado no tabuleiro (veja a figura 2).

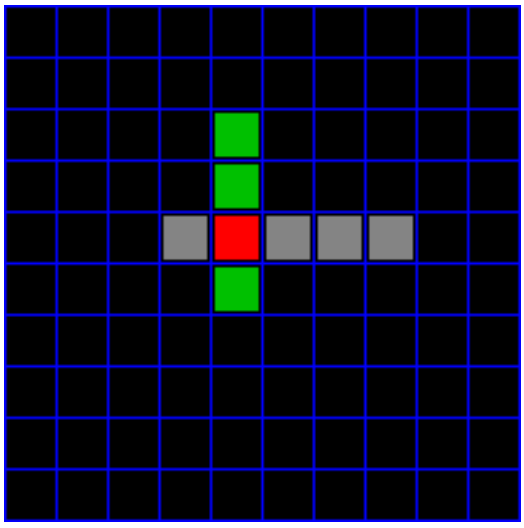


Figura 2— Colisão entre um porta-aviões (cinza) e um navio de guerra (verde).

Supondo que podemos detectar esses casos, será fácil escrever o código para posicionar os navios da esquadra do computador.

O COMPUTADOR POSICIONA SEUS NAVIOS

Comecemos então a escrever o código. Vamos usar o esquema acima, para criar, na classe Board – o esqueleto de um método para posicionar os navios do computador – chamaremos esse método de placeComputerShips (os comentários em azul indicam locais onde deveremos inserir algum código, mais tarde):

```
/**
* Posiciona os navios do computador
*/
public void placeComputerShips() {
   int i = 0;

   do {
      int row;
      int col;
      int orientation;

      // Gera, aleatoriamente, uma linha,
      // uma coluna e uma orientação.

      boolean bFitsOnBoard = false;
      // Verifica se o navio cabe na posição
      // escolhida do tabuleiro.

      boolean bHitsOtherShips = false;
      // Verifica se o navio colide com outros navios
      // já colocados no tabuleiro.

      if ((bFitsOnBoard == true) &&
         (bHitsOtherShips == false)) {
         // Coloca o navio no tabuleiro.

         // Passa para o próximo navio.
        i++;
      }

   } while (i < SHIPS_PER_FLEET);
}
```

Antes de continuar a escrever o código, vamos procurar entender melhor o que foi feito até agora.Observe cuidadosamente a estrutura do comando do loop – note que incrementamos o coontador do loop (i) somente se o navio cabe no tabuleiro e se ele não colide com outros navios. Esse é um conceito muito importantet. Se o navio não cabe no tabuleiro, ou se colide com outro navio, executamos nova iteração do loop, sem modificar o valor de i. Em outras palavras, escolhemos uma nova linha, coluna e orientação, para posicionar o navio corrente. Esse processo se repete até que o navio possa ser posicionado adequadamente, situação na qual incrementamos i, passando assim para o próximo navio.

Só mais um detalhe, antes de continuar: você já deverá saber o que significa o tipo

boolean, usado no código acima – uma variável desse tipo, pode conter apenas um dos seguintes valores: true (isto é, verdadeiro) ou false (isto é, falso). Se você não se lembra disso, reveja a descrição dos tipos primitivos da linguagem Java, no apêndice do módulo 2. Lembre-se também que o símbolo && representa, em Java, o operador booleano "e". Novamente, se você não está lembrado, reveja os operadores Java, usados mais comumente em programas, no apêndice do módulo 3,.

Podemos agora seguir em frente. Vamos substituir os comentários acima, por código que implemente as ações indicadas.

Gerando valores aleatórios

Temos que gerar valores para a linha, coluna e orientação de um navio, aleatoriamente. Como isso pode ser feito? Felizmente, Java prove uma classe que fornece métodos para isso – essa classe chama-se Random e é definida na biblioteca java.util. Isso significa que temos que adicionar, no início do arquivo da classe Board, ao seguinte cláusula de importação:

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

Antes de começar a usar a classe Random, precisamos abordar alguns conceitos novos. Primeiro, não existe, de fato, em programas, o que chamamos de "número aleatório". Java, assim como nas demais linguagens de programação, utiliza um "gerador de números pseudo-aleatórios" para criar uma série de números que parecem aleatórios. Você pode pensar sobre isso do seguinte modo: em algum lugar da classe Random, existe um código para produzir uma série de números, tal como:

1, 12, -77, 32, 109, 0, 0, -2, -2, 45, ...

Quando você pede um número aleatório, em seu programa – por meio de uma chamada do método apropriado – esse método simplesmente retorna o próximo número dessa série. Parâmetros especiais podem ser passados para esse método, de maneira a afetar a escolha do que seja o próximo número, mas a série de números permanece sempre a mesma. Isso significa que, se executarmos o programa 15 vezes, usando, a cada vez, exatamente os mesmos parâmetros, iremos obter a mesma série de números em todas as execuções. Em geral, esse não é o comportamento que esperamos de um gerador de números "aleatórios".

Para contornar esse problema, a classe Random usa a idéia de "semente". A semente é um número que pode ser passado para um objeto da classe Random (isto é, um gerador de números pseudo-aleatórios), que controla a posição na série onde começamos a obter os números. Se diferentes valores para a semente forem usados em diferentes execuções do programa, os números serão obtidos a partir de diferentes posições da série e, portanto, números diferentes serão obtidos.

Já posso ouvir você dizendo: "Mas espere! Se eu usar de novo a mesma semente, vou obter uma série de números idêntica. Portanto, de alguma maneira, vou precisar gerar um número aleatório para o valor da semente. Mas, se eu tiver um mecanismo para gerar um número aleatório, de fato, então não precisarei do gerador de números pseudo-aleatórios, definido pela classe Random!".

Não é bem assim. Felizmente, temos uma maneira simples de realmente gerar uma semente aleatoriamente – podemos usar, para isso, a data e hora correntes. Desse modo,

se o usuário tiver ligado o seu computador 15 minutos antes de começar uma partida no jogo de Batalha Naval, obterá um resultado diferente de quando tiver ligado o computador há 14.9 minutos, ou 2 horas, ou 3 segundos.
Bem, já vimos teoria suficiente sobre números aleatórios – agora vamos ver como usar a classe Random para obter números aleatórios em um programa.

```
/**
* Posiciona os navios do computador
*/
public void placeComputerShips() {
    // Inicializa o gerador de números aleatórios.
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

   int i = 0;

   do {
      int row;
      int col;
      int orientation;

      // Gera, aleatoriamente, uma linha,
      // uma coluna e uma orientação.
       row = randomizer.nextInt(ROW_COUNT);
       col = randomizer.nextInt(COLUMN_COUNT);
       orientation = randomizer.nextInt(4);

      boolean bFitsOnBoard = false;
      // Verifica se o navio cabe na posição
      // escolhida do tabuleiro.
      boolean bHitsOtherShips = false;

      // Verifica o navio colide com outros navios
      // já colocados no tabuleiro.

      if ((bFitsOnBoard == true) &&
         (bHitsOtherShips == false)) {
         // Coloca o navio no tabuleiro.
         // Passa para o próximo navio.

        i++;
      }

   } while (i < SHIPS_PER_FLEET);
}
```

O código destacado em laranja ilustra o uso da classe Random. O comando:

```
long seed = System.currentTimeMillis();
```

simplesmente obtém a hora corrente, em milisegundos (1/1000 de segundo), e armazena esse valor na variável seed (isto é, semente). O tipo dessa variável – long – é semelhante ao tipo int, exceto que variáveis desse tipo podem armazenar valore inteiros muito grandes (reveja a descrição dos tipos primitivos da linguagem Java no apêndice do módulo 2.

O comando seguinte:

```
Random randomizer = new Random(seed);
```

cria uma nova instância da classe Random e armazena a referência a esse objeto na variável randomizer. Ao passar a semente – seed – para o constructor da classe Random, garante-se que, cada vez que o jogo for executado, os números "pesudo-aleatórios" serão obtidos números a partir de pontos diferentes da série.
A geração de números aleatórios é feita pelos seguintes comandos:

```
row = randomizer.nextInt(ROW_COUNT);
col = randomizer.nextInt(COLUMN_COUNT);
orientation = randomizer.nextInt(4);
```

Usamos o método nextInt – definido na classe Random – para obter o próximo valor inteiro da série de números pseudo-aleatórios. Note que é passado um valor inteiro como argumento em uma chamada do método nextInt - isso indica que desejamos obter um número compreendido entre 0 e o valor passado como argumento menos 1. Por exemplo, a chamada randomizer.nextInt(4) gera um número aleatório na faixa de valores de 0 a 3.
Porque desejamos um valor para a orientação compreendido nessa faixa de valores? Recorde-se dos valores das constantes que representam as possíveis orientações, definidas na classe Ship:

```
public static final int ORIENTATION_UP = 0;          // orientação para cima
public static final int ORIENTATION_RIGHT = 1;       // orientação para a direita
public static final int ORIENTATION_DOWN = 2;        // orientação para baixo
public static final int ORIENTATION_LEFT = 3;        // orientação para a esquerda
```

Portanto, note que, de fato, estamos gerando um valor igual a ORIENTATION_UP, ORIENTATION_RIGHT, ORIENTATION_DOWN, ou ORIENTATION_LEFT.

Vá em frente – compile o seu programa, para certificar-se de que você não cometeu nenhum erro bobo. Não execute o programa agora – existe um loop infinito no seu código (você é capaz de dizer aonde?).

E agora? Vamos verificar se o navio cabe na posição escolhida do tabuleiro?

Verificando se o navio pode ser posicionado no tabuleiro

Felizmente, o código para verifica se um pode ser posicionado no tabuleiro, na posição indicada, não é muito difícil. A nova versão do método placeComputerShips, incluindo e

```
/**
* Posiciona os navios do computador
*/
public void placeComputerShips() {
    // Inicializa o gerador de números aleatórios.
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

  // tipos de navio
```

```
 int[] shipType = {Ship.TYPE_AIRCRAFT_CARRIER,
                   Ship.TYPE_BATTLESHIP,
                   Ship.TYPE_CRUISER,
                   Ship.TYPE_SUBMARINE,
                   Ship.TYPE_PT_BOAT};
  // tamanhos de cada tipo de navio
 int[] shipLength = {5, 4, 3, 3, 2};

int i = 0;
do {
   int row;
   int col;
    int orientation;

    // Gera, aleatoriamente, uma linha,
    // uma coluna e uma orientação.
     row = randomizer.nextInt(ROW_COUNT);
     col = randomizer.nextInt(COLUMN_COUNT);
     orientation = randomizer.nextInt(4);

    // Verifica se o navio cabe na posição
    // escolhida do tabuleiro.
    boolean bFitsOnBoard = false;
    int testLength = shipLength[i] - 1;

    if (orientation == Ship.ORIENTATION_UP) {              // orientação para cima
       if (row >= testLength) {
          bFitsOnBoard = true;
       }
    }
     else if (orientation == Ship.ORIENTATION_RIGHT) {    // orientação para a direita
        if (COLUMN_COUNT - col > testLength) {
          bFitsOnBoard = true;
       }
    }
     else if (orientation == Ship.ORIENTATION_DOWN) {     // orientação para baixo
        if (ROW_COUNT - row > testLength) {
          bFitsOnBoard = true;
       }
    }
     else if (orientation == Ship.ORIENTATION_LEFT) {      // orientação para a esquer-
da
        if (col >= testLength) {
          bFitsOnBoard = true;
       }
    }

     // Verifica o navio colide com outros navios
     // já colocados no tabuleiro.
     boolean bHitsOtherShips = false;

     if (bFitsOnBoard && !bHitsOtherShips) {
        // Coloca o navio no tabuleiro.
        Ship newShip = new Ship(shipType[i],
```

```
                        orientation,
                        row,
                        col,
                         shipLength[i]);
          this.addShip(newShip);

          // Passa para o próximo navio.
          i++;
       }
    } while (i < SHIPS_PER_FLEET);
}
```

Antes de discutir o código acima, vamos pensar no algoritmo para verificar se o navio pode ser colocado na posição escolhida, descrevendo os passos desse algoritmo, em português. Imagine que você está colocando um navio no tabuleiro – por exemplo, um cruzador (o terceiro navio) – e você quer verificar se ele cabe na posição escolhida. O cruzador ocupa três células – a célula inicial e duas outras. Você poderá saber se o cruzador cabe na posição escolhida usando as respostas às seguintes questões:

1. Se a orientação do cruzador é para cima, existem duas ou mais células vazias acima da linha corrente?

2. Se a orientação do cruzador é para a direita, existem duas ou mais células vazias à direita da coluna corrente?

3. Se a orientação do cruzador é para baixo, existem duas ou mais células vazias abaixo da linha corrente?

4. Se a orientação do cruzador é para a esquerda, existem duas ou mais células vazias à esquerda da coluna corrente?

Para traduzir esse código temos primeiro que resolver o seguinte problema. Precisamos decidir a ordem na qual o computador irá colocar os navios. A ordem não importa; portanto, suponhamos que seja do maior para o menor:

Quando i é igual a 0, coloque o porta-aviões.

Quando i é igual a 1, coloque o navio de guerra.

Quando i é igual a 2, coloque o cruzador.

Quando i é igual a 3, coloque o submarino.

Quando i é igual a 4, coloque o bote.
Para garantir essa ordem, introduzimos o seguinte código no corpo do método:

```
/**
* Posiciona os navios do computador
*/
public void placeComputerShips() {
    // Inicializa o gerador de números aleatórios.
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

   // tipos de navio
    int[] shipType = {Ship.TYPE_AIRCRAFT_CARRIER,
                Ship.TYPE_BATTLESHIP,
```

```
                Ship.TYPE_CRUISER,
                Ship.TYPE_SUBMARINE,
                Ship.TYPE_PT_BOAT};
    // tamanhos de cada tipo de navio
    int[] shipLength = {5, 4, 3, 3, 2};

    int i = 0;
    do {
        // ... Mais código entra aqui ...

        if (bFitsOnBoard && !bHitsOtherShips) {
            // Coloca o navio no tabuleiro.
            Ship newShip = new Ship(shipType[i],
                                    orientation,
                                    row,
                                    col,
                                    shipLength[i]);
            this.addShip(newShip);

            // Passe para o próximo navio.
            i++;
        }
    } while (i < SHIPS_PER_FLEET);
}
```

Note, em particular, as seguintes linhas:

```
            Ship newShip = new Ship(shipType[i],
                                    orientation,
                                    row,
                                    col,
                                    shipLength[i]);
            this.addShip(newShip);
```

Observe que, quando i é igual a 0, usamos o primeiro elemento de shipType e shipLength, que são TYPE_AIRCRAFT_CARRIER e 5, respectivamente, que correspondem ao porta-aviões. De maneira análoga, quando i é igual a 2 (o terceiro navio), usamos TYPE_CRUISER e 3, que correspondem ao cruzador. Desse modo, usamos arrays para definir a ordem em que os navios são colocados no tabuleiro.
Com isso em mente, podemos agora decifrar o seguinte código:

```
        // Verifica se o navio cabe na posição
        // escolhida do tabuleiro.
        boolean bFitsOnBoard = false;
        int testLength = shipLength[i] - 1;
        if (orientation == Ship.ORIENTATION_UP) {          // orientação para cima

            if (row >= testLength) {
                bFitsOnBoard = true;
            }
        }
        else if (orientation == Ship.ORIENTATION_RIGHT) {    // para a direita
            if (COLUMN_COUNT - col > testLength) {
                bFitsOnBoard = true;
```

```
        }
      }
       else if (orientation == Ship.ORIENTATION_DOWN) {     // para baixo
          if (ROW_COUNT - row > testLength) {
             bFitsOnBoard = true;
          }
      }
       else if (orientation == Ship.ORIENTATION_LEFT) {      // para a esquerda
          if (col >= testLength) {
             bFitsOnBoard = true;
          }
      }
```

Primeiro, note que, inicialmente, atribuímos o valor false a bFitsOnBoard – isso significa que supomos, inicialmente, que o navio não cabe na posição escolhida. O código que vem em seguida altera o valor de bFitsOnBoard para true , se, ao contrário, for possível provar que o navio cabe nessa posição.
Note que utilizamos uma variável auxiliar, testLenght, cujo valor é determinado pelo seguinte comando:

```
int testLength = shipLength[i] - 1;
```

Lembre-se que, no caso do cruzador, vimos que, se o navio tem tamanho igual a 3, temos que verificar se existem duas células vazias acima, abaixo, à direita, ou à esquerda da posição do navio, dependendo da sua orientação, para verificar se ele cabe no tabuleiro. Note que 2 é igual a 3 – 1. O comando acima generalize essa idéia: o número de células que precisamos verificar se estão disponíveis, é igual ao tamanho do navio menos 1.

Por exemplo, quando i é igual a 4, testLength = shipLength[4] - 1, ou seja 2 – 1, que é igual a 1. Portanto, nesse caso, estamos posicionando no tabuleiro um bote: temos apenas que verificar se existe 1 célula acima, abaixo, à esquerda ou à direita da posição escolhida.

De maneira análoga, quando i é igual a 1, testLength = shipLength[1] - 1, ou seja 24–1, que é igual a 3 – isso corresponde ao caso do navio de guerra.

Finalmente, considere o comando if...else. Vamos focalizar apenas os dois primeiros casos. Você deverá então entender os demais casos, com facilidade.
Primeiro, temos (lembre-se que o símbolo >=:representa o operador relacional “menor ou igual”):

```
if (orientation == Ship.ORIENTATION_UP) {
    if (row >= testLength) {
       bFitsOnBoard = true;
    }
}
```

Portanto, se o navio está orientado para cima, a partir da linha e coluna indicadas, será executado o seguinte comando:

```
if (row >= testLength) {
    bFitsOnBoard = true;
```

```
}
```

Se o número da linha for maior do que testLength, o valor true é atribuído a bFitsOnBoard – indicando que o navio cabe na grade do tabuleiro. Considere agora a situação de posicionar o navio de guerra na linha 2 (ou seja, na terceira linha, já que as linhas são numeradas a partir de zero). NO caso do navio de guerra, temos que testLength é igual a 4 - 1 = 3. Nesse caso, o teste

```
if (row >= testLength)
```

se reduz a if (2 >= 3), que não é verdadeiro e, portanto, o comando bFitsOnBoard = true; não é executado (ou seja, bFitsOnBoard permanece com o valor false).

Considere agora o caso e posicionar um bote na linha 8 (a nona linha). Nesse caso, testLength é igual a 2 - 1 = 1, e, portanto, if (row >= testLength) se reduz a

if (8 >= 1), que é verdadeiro, e, portanto, o comando bFitsOnBoard = true; é executado, indicando que o bote pode ser posicionado no tabuleiro.
Suponha agora que a orientação é para a direita – ou seja, igual a Ship.ORIENTATION_RIGHT. Nesse caso, é executado o teste:

```
else if (orientation == Ship.ORIENTATION_RIGHT) {
    if (COLUMN_COUNT - col > testLength) {
        bFitsOnBoard = true;
    }
}
```

Suponha que estamos tentando posicionar um submarino (i = 3) na coluna 6 (a sétima coluna). Nesse caso, testLength é igual a 2, e o comando

```
if (COLUMN_COUNT - col > testLength) {
    bFitsOnBoard = true;
}
```

se reduz a

```
if (10 - 6 > 2) {
    bFitsOnBoard = true;
}
```

Como 4 é maior que 2, o código do corpo do comando if é executado, sendo atribuído o valor true a bFitsOnBoard.
Se tentarmos, agora, colocar o submarino na coluna 8 (isto é, na nona coluna), o comando se reduziria a

```
if (10 - 8 > 2) {
    bFitsOnBoard = true;
}
```

e, portanto, bFitsOnBoard permaneceria com o valor false, que lhe foi atribuído anteriormente.

Os outros casos se comportam de maneira análoga.

Tente compilar o seu programa, mas não o execute ainda. Embora a lógica do nosso

jogo ainda não esteja completa, é bom compilá-lo agora, para verificar se nenhum erro de sintaxe foi cometido.

Colisão entre navios

Finalmente, podemos escrever a última parte do código – para testar se o navio que está sendo posicionado não colide com outros navios colocados no tabuleiro anteriormente. Felizmente, isso é fácil. Lembre-se que, quando é feita uma chamada do método addShip, o tipo do navio é armazenado nas células da grade – isto é, no array gridCells. Portanto, o que temos que fazer, para detectar uma colisão, é apenas ler o valor existente nas células da grade onde queremos posicionar o navio. Se o valor armazenado aí for diferente de Ship.TYPE_NONE, que representa uma célula vazia, então, houve colisão. Adicionando o código para fazer isso, o método placeComputerShips ficaria assim:

```
/**
* Posiciona os navios do computador
*/
public void placeComputerShips() {
    // Inicializa o gerador de números aleatórios.
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

   // tipos de navio
    int[] shipType = {Ship.TYPE_AIRCRAFT_CARRIER,
                Ship.TYPE_BATTLESHIP,
                Ship.TYPE_CRUISER,
                Ship.TYPE_SUBMARINE,
                Ship.TYPE_PT_BOAT};
  // tamanhos de cada tipo de navio
   int[] shipLength = {5, 4, 3, 3, 2};

  int i = 0;
  do {
     int row;
     int col;
     int orientation;

     // Gera, aleatoriamente, uma linha,
     // uma coluna e uma orientação.
      row = randomizer.nextInt(ROW_COUNT);
      col = randomizer.nextInt(COLUMN_COUNT);
      orientation = randomizer.nextInt(4);

     // Verifica se o navio cabe na posição
     // escolhida do tabuleiro.
     boolean bFitsOnBoard = false;
     int testLength = shipLength[i] - 1;

     if (orientation == Ship.ORIENTATION_UP) {       // orientação para cima
        if (row >= testLength) {
           bFitsOnBoard = true;
        }
     }
      else if (orientation == Ship.ORIENTATION_RIGHT) {    // para a direita
```

```
    if (COLUMN_COUNT - col > testLength) {
      bFitsOnBoard = true;
    }
  }
  else if (orientation == Ship.ORIENTATION_DOWN) {     // para baixo
    if (ROW_COUNT - row > testLength) {
      bFitsOnBoard = true;
    }
  }
  else if (orientation == Ship.ORIENTATION_LEFT) {      // para a esquerda
    if (col >= testLength) {
      bFitsOnBoard = true;
    }
  }

  // Verifica se o navio colide com outros navios
  // já colocados no tabuleiro.
  boolean bHitsOtherShips = false;
  if (bFitsOnBoard == true) {
    int j;
    if (orientation == Ship.ORIENTATION_UP) {
      j = 0;
      while (j < shipLength[i]) {
        if (this.gridCells[row - j][col] !=
          Ship.TYPE_NONE) {
          bHitsOtherShips = true;
          break;
        }
        j++;
      }
    }
    else if (orientation == Ship.ORIENTATION_RIGHT) {
      j = 0;
      while (j < shipLength[i]) {
        if (this.gridCells[row][col + j] !=
          Ship.TYPE_NONE) {
          bHitsOtherShips = true;
          break;
        }
        j++;
      }
    }
    else if (orientation == Ship.ORIENTATION_DOWN) {
      j = 0;
      while (j < shipLength[i]) {
        if (this.gridCells[row + j][col] !=
          Ship.TYPE_NONE) {
          bHitsOtherShips = true;
          break;
        }
        j++;
      }
    }
```

```
            else if (orientation == Ship.ORIENTATION_LEFT) {
              j = 0;
               while (j < shipLength[i]) {
                   if (this.gridCells[row][col - j] !=
                      Ship.TYPE_NONE) {
                      bHitsOtherShips = true;
                     break;
                  }
                  j++;
               }
            }
         }

          if (bFitsOnBoard && !bHitsOtherShips) {
             // Coloca o navio no tabuleiro.
             Ship newShip = new Ship(shipType[i],
                             orientation,
                             row,
                             col,
                              shipLength[i]);
             this.addShip(newShip);

             // Passa para o próximo navio.
            i++;
         }

      } while (i < SHIPS_PER_FLEET);
}
```

Como a maior parte do código adicionado (destacado em laranja) deve parecer familiar, não vamos analisá-lo em detalhe, mas apenas comentar alguns pontos. Primeiro, note que todo esse trecho de código está contido no corpo da cláusula if do comando:

```
if (bFitsOnBoard == true) {
  // ...more code in here...
}
```

Isso faz com que o código para testar a colisão entre navios apenas seja executado caso o navio a ser posicionado caiba no tabuleiro. Caso contrário, naturalmente não é necessário testar se ele colide com algum navio já colocado no tabuleiro.
Em seguida, verifica-se a possibilidade e colisão no caso de cada uma das possíveis orientações do navio (para cima, para baixo, para a esquerda e para a direita). Vamos examinar o caso em que a orientação é para cima – os demais casos são análogos.

```
if (orientation == Ship.ORIENTATION_UP) {
   j = 0;
    while (j < shipLength[i]) {
       if (this.gridCells[row - j][col] !=
          Ship.TYPE_NONE) {
          bHitsOtherShips = true;
         break;
      }
      j++;
```

```
    }
}
```

Note que o valor do contador do loop – j – varia de 0 até o tamanho do navio. Para cada um desses valores, verificamos o valor da célula correspondente na grade, por meio dos seguintes comandos (lembre-se que '!=' significa diferente):

```
if (this.gridCells[row - j][col] != Ship.TYPE_NONE) {
    bHitsOtherShips = true;
    break;
}
```

Observe os valores de [row-j], à medida que j varia, de 0 até o tamanho do navio. Acompanhe a execução desse código, para o caso de posicionar um cruzador, na linha 4 e coluna 5. Temos:

j = 0, [row - j] = [4 - 0] = 4, e é verificada a célula (4, 5)

j = 1, [row - j] = [4 - 1] = 3, e é verificada a célula (3, 5)

j = 2, [row - j] = [4 - 2] = 2, e é verificada a célula (2, 5)

j = 3, [row - j] = [4 - 3] = 1, e é verificada a célula (1, 5)

Você entendeu como funciona? Então viu que, à medida que varia o valor de j, verificam-se diferentes células da grade. Em cada caso, é testado se a célula corrente contém parte de algum navio (ou, literalmente, se a célula corrente não está vazia). Se isso ocorrer, então é atribuído o valor true à variável bHitsOtherShips e...

...break? O que isso significa, exatamente?

A palavra chave break indica que o loop corrente deve ser abandonado, não importando quais sejam as condições do loop. Nesse caso, o break faz com que o loop termine, independentemente do valor corrente de j. Isso é feito apenas por razão de eficiência: se já detectamos a colisão em uma determinada célula, não é necessário examinar as células restantes – sabemos imediatamente que o navio não pode ser colocado na posição escolhida.

Getting Testy

Bem, escrevemos um bocado de código! – está na hora de testá-lo. Isso é fácil – apenas temos que chamar o método placeComputerShips em algum lugar do programa. Vamos colocar essa chamada na classe BattleshipApp, como mostrado a seguir:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
         this.blueBoard.placeComputerShips();
       this.repaint();
   }
}
```

Note que o uso do comando if...else acima garante que o computador (que joga no tabuleiro azul) posiciona os seus navios apenas na fase de transição entre o estado de introdução – GAME_STATE_INTRO – e o estado de preparação – GAME_STATE_SETUP – do jogo.

Mais uma observação final – o código dessa versão do método placeComputerShip é longo demais, o que o torna difícil de ser lido. Como boa prática de programação, é aconselhável sempre manter menor o código de um método, dividindo-o em partes menores, quando necessário.

O código que escrevemos até agora para as classes Board e BattleshipApp está disponível como o código 13a, apresentado na seção de códigos incluída no final deste tutorial. Além disso, o código 13b apresenta uma versão melhorada da classe Board, na qual o método placeComputerShips é dividido em partes menores. Compare os dois códigos e veja se você consegue entender bem o que foi feito.

Você provavelmente vai querer fazer uma pequena pausa agora. Na seção a seguir, vamos combinar os códigos da grade, do mouse e alguns outros truques, para criar o código para que o usuário também possa posicionar seus navios.

A esquadra do usuário

Na seção anterior, escrevemos a parte do código do programa que trata o posicionamento dos navios no tabuleiro, pelo computador. Agora vamos escrever o código que possibilita ao usuário também posicionar sua esquadra. Felizmente, a base do código é a mesma, em ambos os casos; portanto, parte do nosso trabalho já está feita.

Como sempre, antes de começar a escrever o código, vamos pensar sobre as tarefas envolvidas nesse caso. Temos muitas opções, mas todas elas devem possibilitar ao usuário especificar os seguintes parâmetros: a linha, a coluna e a orientação do navio que se deseja posicionar.

A seguir, vamos discutir um método para prover essa funcionalidade. Em seguida, vamos resumir, em português, as tarefas envolvidas. Finalmente, vamos escrever o código Java para implementar o que desejamos. Então, vamos lá.

Como posicionar um navio

A maneira mais simples para permitir ao usuário posicionar os seus navios é deixando que ele clique sobre a posição desejada, na grade do tabuleiro. Para implementar isso, vamos nos basear na interface pré-definida MouseListener, que já conhecemos.

Infelizmente, não basta que o usuário apenas clique sobre uma célula da grade, para posicionar um navio. Ele também terá que especificar, de alguma forma, a orientação do navio. Vamos supor, então, que o usuário deverá clicar sobre a posição inicial e, em seguida, sobre a posição da outra extremidade, para cada navio.

Para quem já jogou Batalha Naval, isso fica fácil, pois saberá que um porta-aviões deve ocupar 5 posições, um navio de guerra, 4 posições, um cruzador, 3 posições etc. Entretanto, para um jogador não familiarizado com o jogo, será necessário prover alguma instrução adicional: depois que ele clicar sobre a posição inicial, o programa deverá indicar quais são as possíveis células que ele poderá marcar, como a outra extremidade do navio.

Finalmente, devemos planejar uma maneira de o usuário alterar a posição de algum navio, caso ele deseje. A forma mais simples de se fazer isso é permitir que o usuário clique sobre uma nova célula inicial. Se ele fizer isso antes de selecionar a outra extremidade do navio, devemos abortar a escolha corrente e reiniciar o processo de

posicionamento do navio, a partir dessa nova posição.

Bem, vamos então procurar relacionar os passos para este procedimento, em linguagem natural.

Os passos do procedimento

O algoritmo de posicionamento de um navio pelo usuário pode ser descrito do seguinte modo:

1. Comece com o primeiro navio.

2. Quando o usuário clicar sobre uma célula da grade, armazena a linha e a coluna da posição inicial escolhida.

3. Tomando como base essa posição inicial, calcule as possíveis posições da outra extremidade do navio.

4. Destaque, no tabuleiro, as posições válidas para a outra extremidade do navio.

5. Quando o usuário clicar sobre o tabuleiro pela segunda vez, verifique se a célula escolhida é uma das posições válidas para a outra extremidade do navio e, em caso afirmativo, coloque o navio no tabuleiro; caso contrário, mova o ponto inicial para essa nova posição e volte ao passo 3.

6. Se o usuário tiver colocado o navio no tabuleiro corretamente, passe para o próximo navio.

7. Se não houver mais navios para serem posicionados, termine o procedimento; caso contrário, vá para o passo 2.

Se usarmos um 'x' para marcar a posição inicial e pequenos quadrados para marcar as possíveis posições da outra extremidade do navio. A aparência do tabuleiro, ao longo desse processo, será semelhante à seguinte:

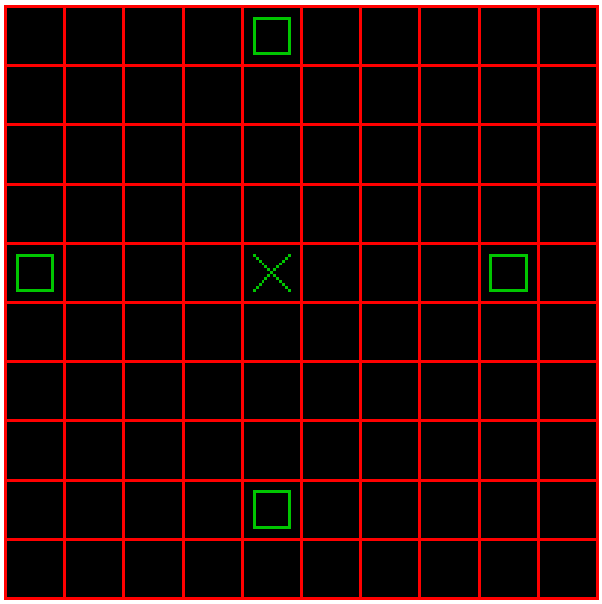


Figura 3—O usuário posiciona seu porta-aviões.

Parece que nosso algoritmo está OK – mas existem alguns pequenos problemas. Primeiro, suponha que o usuário seleciona uma posição inicial muito próxima da margem do tabuleiro, de maneira que não seja possível posicionar o navio, a partir dessa posição, em cada uma

das 4 direções. O que se deve fazer então? Em Segundo lugar, suponha que o usuário seleciona, como posição inicial, uma célula que já contém parte de um navio, posicionado anteriormente. Assim como no caso do nosso algoritmo para posicionamento da esquadra pelo computador, o programa deve detectar essas situações, permitindo que o usuário apenas possa escolher um posicionamento legal para a sua esquadra. A figura 4 ilustra isso.

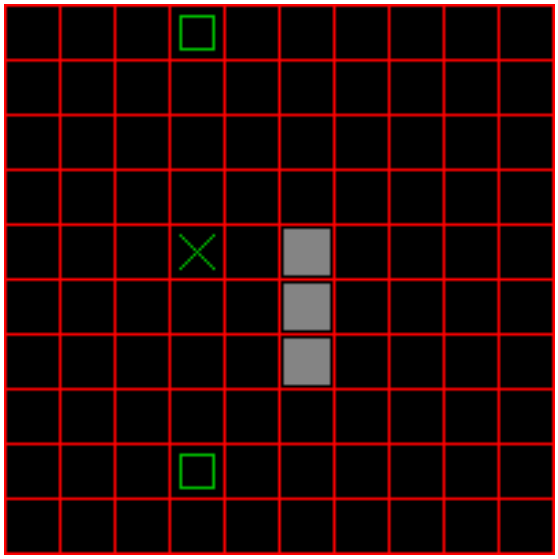


Figura 4 — Posição inválida do porta-aviões, nas direções "esquerda" e "direita".

Observe, na figura acima, observe que não existe nenhum pequeno quadrado marcado à esquerda ou à direita do 'x'. Nesse caso, a tentativa de posicionar o porta-aviões com orientação para a excederia os limites do tabuleiro, e o posicionamento com orientação para a direita resultaria em colisão.

Note que o passo 3 do esquema acima supõe que esses casos devem ser verificados.

Vamos então procurar escrever o código Java para implementar o procedimento descrito acima.

As quatro possíveis posições da outra extremidade do navio

Como sempre, vamos construir o código do corpo do nosso novo método, a partir de uma série de passos menores. Vamos começar pelo procedimento de detectar o evento de clicar do mouse sobre uma célula da grade do tabuleiro, identificando a linha e a coluna correspondentes e desenhando um 'x' nessa célula. Isso requer que adicionemos algum código à classe BattleshipApp, que implementa a interface MouseListener, e também à classe Board, para armazenar a posição selecionada no objeto que representa o tabuleiro (denotado pela variável gridCells), assim como para desenhar o 'x' na célula correspondente.
Vamos começar pela classe BattleshipApp:

```
// variáveis de instância
private Board redBoard;                 // grade vermelha do tabuleiro
private Board blueBoard;                // grade azul do tabuleiro
private FriendlyPlayer friendlyPlayer;  // jogador aliado
private EnemyPlayer enemyPlayer;        // jogador inimigo
private int gameState;                  // estado do jogo
private boolean bFirstSetupClick;       // indica se o clique seleciona a posição inicial
.                                       // ou a extremidade final do navio corrente
```

```
.
.
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
         this.blueBoard.placeComputerShips();
       this.repaint();
       this.initializeSetupState();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       if (this.bFirstSetupClick) {
            // Armazena a linha e coluna correspondentes
            // à posição do primeiro clique do mouse.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
            this.redBoard.setFirstRowColumn(row, col);
           this.bFirstSetupClick = false;
           this.repaint();
      }
      else {
          this.bFirstSetupClick = true;
      }
   }
}

private void initializeSetupState() {
    this.bFirstSetupClick = true;
}

private int convertYtoRow(int pixelY) {
    Container clientArea = this.getContentPane();
    int borderSize = (this.getWidth() -
                 clientArea.getWidth()) / 2;
    int titleSize = (this.getHeight() -
                clientArea.getHeight()) -
       borderSize;
    int row = (pixelY - titleSize - RED_GRID_Y) /
     Board.ROW_PIXEL_HEIGHT;
    return row;
}

private int convertXtoCol(int pixelX) {
    Container clientArea = this.getContentPane();
    int borderSize = (this.getWidth() -
                 clientArea.getWidth()) / 2;
    int column = (pixelX - borderSize - RED_GRID_X) /
     Board.COLUMN_PIXEL_WIDTH;
    return column;
}
```

OK, vamos parar por um momento e procurar entender o código que foi introduzido acima. Primeiramente, note que foi introduzida uma nova variável – bFirstSetupClick, de tipo boolean. Essa variável terá valor true, caso o usuário esteja prestes a clicar sobre a posição inicial de um navio, ou terá valor false, caso contrário.

Em seguida, adicionamos código ao método mouseReleased. Em particular, introduzimos o seguinte comando:

```
this.initializeSetupState();
```

Esse comando é uma chamada do método ao método initializeSetupState, definido logo em seguida, que coloca o jogo pronto para receber informação sobre o posicionamento dos navios pelo usuário, quando entramos no estado GAME_SETUP (note que isso acontece somente depois que o usuário clica sobre a janela de abertura do jogo, quando o computador então posiciona a sua esquadra, automaticamente). O método initializeSetupState não é assim tão interessante – ele apenas inicializa a variável bFirstSetupClick, atribuindo-lhe o valor true, para indicar que, a partir de então o usuário poderá clicar sobre o tabuleiro para escolher a posição de um navio.
O seguinte trecho de código vem logo em seguida:

```
else if (this.gameState == GAME_STATE_SETUP) {
    if (this.bFirstSetupClick) {
        // Armazena a linha e coluna correspondentes
        // à posição do primeiro clique do mouse.
        int row = this.convertYtoRow(event.getY());
        int col = this.convertXtoCol(event.getX());
        this.redBoard.setFirstRowColumn(row, col);
       this.bFirstSetupClick = false;
       this.repaint();
   }
   else {
       this.bFirstSetupClick = true;
   }
}
```

Primeiro, note que o corpo da cláusula else if apenas será executado quando o jogo estiver no estado de preparação. O comando if mais interno, ou seja:

```
if (this.bFirstSetupClick) {
    // algum código aqui...
    this.bFirstSetupClick = false;
    // mais código aqui.
}
else {
    this.bFirstSetupClick = true;
}
```

terá sua condição avaliada como true, pois esse valor acaba de ser atribuído a bFirstSetupClick, na entrada no estado de preparação do jogo. Portanto, na primeira vez em que o usuário clica sobre o tabuleiro, a execução do programa entra na cláusula if desse comando. Note que, no corpo desta cláusula, é atribuído o valor false à variável bFirstSetupClick. Ou seja, na próxima vez em que o usuário clicar o mouse, o programa irá executar o corpo da cláusula else, o qual atribui o valor true a bFirstSetupClick novamente. Isso faz com que, na próxima vez que o mouse for clicado, seja executado o corpo da cláusula if e, na vez seguinte, novamente o corpo da cláusula else, e assim por diante. Você verá outros trechos de código semelhantes a esse com freqüência; portanto, é bom que o examinemos mais detalhadamente.

A porção do código que será executada quando o usuário clicar o mouse pela primeira vez é a seguinte:

```
// Armazena a linha e coluna correspondentes
// à posição do primeiro clique do mouse.
int row = this.convertYtoRow(event.getY());
int col = this.convertXtoCol(event.getX());
this.redBoard.setFirstRowColumn(row, col);
this.bFirstSetupClick = false;
this.repaint();
```

Note as duas chamadas de métodos para converter as coordenadas (x,y) da posição do mouse, em pixels, para a posição no tabuleiro do jogo, em termos de linha e coluna. Você poderá ler sobre como essa conversões são feitas no apêndice 1 deste módulo. Entretanto, talvez seja melhor voltar a esse assunto no final desta seção, para não interromper agora a nossa linha de raciocínio.
Vamos então, agora, adicionar o código necessário na classe Board:

```
// variáveis de classe.
private static final int ROW_COUNT        = 10;
private static final int COLUMN_COUNT     = 10;
public static final int ROW_PIXEL_HEIGHT  = 20;
public static final int COLUMN_PIXEL_WIDTH  = 20;
private static final int SHIPS_PER_FLEET  = 5;

public Board()
{
    // Inicializa os arrays usados para armazenar informação sobre os navios.
    this.fleet = new Ship[SHIPS_PER_FLEET];
    this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];

    // Preenche as células da grade com vazio.
   int i = 0;
   while (i < ROW_COUNT) {
      int j = 0;
       while (j < COLUMN_COUNT) {
          this.gridCells[i][j] = Ship.TYPE_NONE;
         j++;
      }

      i++;
   }

   // Inicializa as variáveis que indicam a linha e coluna da posição inicial.
   this.firstClickRow = -1;
   this.firstClickCol = -1;
}

public void setFirstRowColumn(int row, int col) {
   this.firstClickRow = row;
   this.firstClickCol = col;
}

public void drawGrid(Graphics gfx, Color gridColor,
```

```
                int startX, int startY) {
    // Especifica a cord a linha da grade.
     gfx.setColor(gridColor);

    // Desenha as linhas horizontais.
    int lineCounter = 1;
    int x = startX;
    int y = startY;
    while (lineCounter <= 11) {
        gfx.drawLine(x, y, x + COLUMN_PIXEL_WIDTH * COLUMN_COUNT, y);
       y = y + ROW_PIXEL_HEIGHT;
       lineCounter++;
    }

    // Desenha as linhas verticais.
    lineCounter = 1;
    x = startX;
    y = startY;
    while (lineCounter <= 11) {
        gfx.drawLine(x, y, x, y + ROW_PIXEL_HEIGHT *
          ROW_COUNT);
       x = x + COLUMN_PIXEL_WIDTH;
       lineCounter++;
    }

    // Desenha o marcador da posição onde ocorreu o clique.
    if ((this.firstClickRow != -1) &&
       (this.firstClickCol != -1)) {
        x = startX + this.firstClickCol * COLUMN_PIXEL_WIDTH;
        y = startY + this.firstClickRow * ROW_PIXEL_HEIGHT;
       x = x + 2;
       y = y + 2;
        int dx = COLUMN_PIXEL_WIDTH - 4;
        int dy = ROW_PIXEL_HEIGHT - 4;
         gfx.setColor(Color.green);
        gfx.drawLine(x, y, x + dx, y + dy);
        gfx.drawLine(x + dx, y, x, y + dy);
    }
}
```

Grande parte do código da classe Board, apresentado acima, já foi visto anteriormente e, por isso, não vamos discuti-lo em detalhe. Existem, entretanto, alguns pontos importantes. Primeiro, note que modificamos o atributo das variáveis ROW_PIXEL_HEIGHT e COLUMN_PIXEL_WIDTH, de private para public. Porque? Bem, os métodos convertYtoRow e convertXtoCol, definidos na classe BattleshipApp, convertem as coordenadas da posição onde ocorreu o clique do mouse, determinadas em pixels, para linha e coluna do tabuleiro. Esses métodos precisam usar os valores de largura da coluna e altura da linha da grade do tabuleiro. Isso não poderia ser feito, se as variáveis que representam esses valores fossem declaradas, na classe Board, como variáveis privadas.

Em segundo lugar, note que as variáveis firstClickRow e firstClickCol são ambas inicializadas com o valor -1, no construtor da classe Board. No método drawGrid, verificamos se essas variáveis têm valor diferente de -1, antes de desenhar o 'x' que marca a

posição onde o mouse foi clicado. Esse pequeno truque nos permite esconder o 'x' quando não desejamos que seja exibido: se firstClickRow ou firstClickCol for igual a -1, o 'x' não é exibido na grade do tabuleiro.

Faça as alterações indicadas acima no seu código e, depois disso, compile e execute o seu programa. Se você obtiver algum erro de compilação, poderá conferir o seu programa com o código 14 da seção de códigos incluída no final deste tutorial.

Tente clicar na grade vermelha e ver o que acontece. Note que apenas o primeiro clique altera a posição do 'x' exibido na grade. O segundo clique, por enquanto, não faz nada. Em seguida, você estará novamente escolhendo uma posição inicial (primeiro clique), o que altera a posição do 'x' novamente. Recorde-se do que explicamos sobre o uso da variável bFirstSetupClick no método mouseReleased, da classe BattleshipApp. Agora você pode ver esse código em ação.

Tente clicar fora da grade vermelha. Opa! Parece que você pode colocar o 'x' em qualquer posição da tela! Isso não deveria acontecer! Parece que precisamos adicionar algum código ao nosso programa, para corrigir esse erro...

Marcando o X apenas dentro do tabuleiro

Felizmente, é muito fácil evitar o erro acima. Precisamos apenas garantir que os valores de linha e coluna passados como argumento para o método setFirstRowColumn estejam dentro da região do tabuleiro. A seguinte correção resolve o nosso problema:

```
public void setFirstRowColumn(int row, int col) {
   if ((row >= 0) &&
       (row < ROW_COUNT) &&
       (col >= 0) &&
       (col < COLUMN_COUNT)) {
       this.firstClickRow = row;
       this.firstClickCol = col;
   }
   else {
       this.firstClickRow = -1;
       this.firstClickCol = -1;
   }
}
```

Você não deverá ter muita dificuldade para entender o código acima e, por isso, não vamos comentá-lo. Procure fazer essa correção e, em seguida, compilar e executar o programa novamente. Verifique se a correção que você fez de fato garante que o x seja desenhado apenas quando o mouse for clicado dentro dos limites da grade vermelha do tabuleiro.

A outra extremidade do navio

Agora podemos escrever o código para desenhar no tabuleiro os pequenos quadrados que marcam as possíveis posições da outra extremidade do navio corrente, depois que a posição de uma das extremidades é selecionada pelo usuário, clicando sobre uma célula da grade do tabuleiro.

Humm... navio corrente? Parece então que devemos manter informação, de alguma

maneira, sobre o tipo do navio que o usuário está posicionando no tabuleiro, em cada momento. Para isso, vamos utilizar uma variação do método que escrevemos anteriormente, para de posicionamento dos navios pelo computador. Vamos introduzir uma variável de instância de tipo int, para armazenar o tipo do navio corrente – 0, para porta-aviões, 1, para navio de guerra etc.
Vamos ver como isso funciona:

```
// variáveis de instância.
private RedPeg[] hitMarkers;          // pinos marcadores de acerto
private WhitePeg[] missMarkers;       // pinos marcadores de erros
private Ship[] fleet;                 // esquadra de navios
private int[][] gridCells;            // grade do tabuleiro
private int firstClickRow;            // linha da posição inicial do navio corrente
private int firstClickCol;            // coluna da posição inicial do navio corrente

private int currentUserShip;          // tipo do navio corrente
// tipos de navio
private int[] userShipTypes = {Ship.TYPE_AIRCRAFT_CARRIER,
                    Ship.TYPE_BATTLESHIP,
                    Ship.TYPE_CRUISER,
                    Ship.TYPE_SUBMARINE,
                    Ship.TYPE_PT_BOAT};
// tamanhos de navio
private int[] userShipLengths = {5, 4, 3, 3, 2};

public Board()
{
    // Inicializa os arrays usados para armazenar informação sobre os navios.
    this.fleet = new Ship[SHIPS_PER_FLEET];
    this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];

   // Preenche as células da grade com vazio.
   int i = 0;
   while (i < ROW_COUNT) {
      int j = 0;
       while (j < COLUMN_COUNT) {
          this.gridCells[i][j] = Ship.TYPE_NONE;
         j++;
      }
      i++;
   }

   // Inicializa as variáveis que indicam a linha e coluna da posição inicial.
   this.firstClickRow = -1;
   this.firstClickCol = -1;

   // Inicializa o tipo do navio corrente do usuário.
   this.currentUserShip = 0;
}
```

Você deve ter notado que os arrays userShipTypes e userShipLengths correspondem exatamente às variáveis shipType e shipLength que usamos no método placeComputerShips. Como regra geral, é óbvio que não queremos ter código repetido em um programa – isso constitui uma enorme fonte de erros. Portanto, vamos eliminar as antigas variáveis shipType e shipLength, do seguinte modo:

```
// variáveis de instância
private RedPeg[] hitMarkers;        // pinos marcadores de acerto
private WhitePeg[] missMarkers;     // pinos marcadores de erros
private Ship[] fleet;               // esquadra de navios
private int[][] gridCells;          // grade do tabuleiro
private int firstClickRow;          // linha da posição inicial do navio corrente
private int firstClickCol;          // coluna da posição inicial do navio corrente
private int currentUserShip;        // tipo do navio corrente
// tipos de navio
private int[] shipType = {Ship.TYPE_AIRCRAFT_CARRIER,
                          Ship.TYPE_BATTLESHIP,
                          Ship.TYPE_CRUISER,
                          Ship.TYPE_SUBMARINE,
                          Ship.TYPE_PT_BOAT};
// tamanhos de navio
private int[] shipLength = {5, 4, 3, 3, 2};

public void placeComputerShips() {
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

    // As variáveis shipType e shipLength, antes
    // definidas localmente aqui, foram promovidas
    // a variáveis de instância da classe, sendo,
    // portanto, agora visíveis em todo o corpo da classe
    // e não apenas no corpo deste método.

   int i = 0;
   do {
      // ...um bocado de código entra aqui...
   }
}
```

O trecho marcado em azul indica a posição onde as variáveis shipType e shipLength eram anteriormente declaradas: anteriormente, elas eram variáveis locais ao método placeComputerShips, o que significa que não podiam ser usadas fora desse método. Agora que elas são declaradas como variáveis de instância, podem ser usadas em qualquer método da classe Board, na qual são declaradas. Com essa modificação, tanto o método placeComputerShips, como o nosso novo código para posicionamento dos navios pelo usuário, podem usar a mesma informação sobre tipo e tamanho de navio, sendo assim eliminada a duplicação de código.

Agora que já temos uma maneira de manter informação sobre o navio corrente, podemos escrever o código relacionado com a escolha da posição da "extremidade final" do navio.

Indicando as possíveis posições da extremidade final do navio

Antes de tudo, vamos precisar armazenar informação sobre a linha e coluna de cada uma das células onde o usuário pode marcar a "extremidade final" do nario. Vamos então criar arrays para armazenar essa informação:

```
// variáveis de instância.
```

```
private RedPeg[] hitMarkers;          // pinos marcadores de acerto
private WhitePeg[] missMarkers;       // pinos marcadores de erros
private Ship[] fleet;                 // esquadra de navios
private int[][] gridCells;            // grade do tabuleiro
private int firstClickRow;            // linha da posição inicial do navio corrente
private int firstClickCol;            // coluna da posição inicial do navio corrente
private int currentUserShip;          // tipo do navio corrente
// tipos de navio
private int[] shipType = {Ship.TYPE_AIRCRAFT_CARRIER,
                          Ship.TYPE_BATTLESHIP,
                          Ship.TYPE_CRUISER,
                          Ship.TYPE_SUBMARINE,
                          Ship.TYPE_PT_BOAT};
// tamanhos de navio
private int[] shipLength = {5, 4, 3, 3, 2};

// linha e coluna das posições válidas para a extremidade final do navio
private int[] endBoxRow = {-1, -1, -1, -1};    // linha
private int[] endBoxCol = {-1, -1, -1, -1};    // coluna
```

Nos arrays endBoxRow e endBoxCol, o elemento de índice 0 corresponde ao quadrado acima da posição inicial, o de índice 1, à posição à direita, o de índice 2, à posição abaixo e o de índice 3, à posição à esquerda. Note que todas essas posições são inicializadas com -1. Desse modo, podemos usar nosso velho truque de não desejar o quadrado de marcação, quando uma dessas posições cai fora dos limites do tabuleiro ou causa colisão com algum navio colocado anteriormente.
Vamos então adicionar, ao método drawGrid, o trecho de código para desenhar esses quadrados.

```
public void drawGrid(Graphics gfx, Color gridColor,
                int startX, int startY) {
    // Especifica a cord as linhas da grade.
     gfx.setColor(gridColor);

    // Traça as linhas horizontais.
    int lineCounter = 1;
    int x = startX;
    int y = startY;
     while (lineCounter <= 11) {
         gfx.drawLine(x, y, x + COLUMN_PIXEL_WIDTH * COLUMN_COUNT, y);
        y = y + ROW_PIXEL_HEIGHT;
        lineCounter++;
    }

    // Traça as linhas verticais.
    lineCounter = 1;
    x = startX;
    y = startY;
     while (lineCounter <= 11) {
         gfx.drawLine(x, y, x, y + ROW_PIXEL_HEIGHT * ROW_COUNT);
        x = x + COLUMN_PIXEL_WIDTH;
        lineCounter++;
    }
```

```
    // Marca o 'x' na posição inicial.
    if ((this.firstClickRow != -1) &&
        (this.firstClickCol != -1)) {
        x = startX + this.firstClickCol * COLUMN_PIXEL_WIDTH;
        y = startY + this.firstClickRow * ROW_PIXEL_HEIGHT;
       x = x + 2;
       y = y + 2;
        int dx = COLUMN_PIXEL_WIDTH - 4;
        int dy = ROW_PIXEL_HEIGHT - 4;
         gfx.setColor(Color.green);
        gfx.drawLine(x, y, x + dx, y + dy);
        gfx.drawLine(x + dx, y, x, y + dy);
    }

     // Desenha os quadrados que marcam as posições
     // válidas para a extremidade final do navio.
    int i = 0;
     while (i < this.endBoxRow.length) {
         if ((this.endBoxRow[i] != -1) &&
           (this.endBoxCol[i] != -1)) {

             x = startX + this.endBoxCol[i] * COLUMN_PIXEL_WIDTH;
            y = startY + this.endBoxRow[i] * ROW_PIXEL_HEIGHT;
            gfx.setColor(Color.green);
           x = x + 2;
           y = y + 2;
            int dx = COLUMN_PIXEL_WIDTH - 3;
            int dy = ROW_PIXEL_HEIGHT - 3;
             gfx.setColor(Color.green);
            gfx.drawRect(x, y, dx, dy);
       }
       i++;
   }
}
```

A maior parte desse código é nosso velho conhecido, com exceção do método drawRect e da forma como é escrita a condição do comando while.
O método drawRect não tem nenhum mistério – ele desenha um retângulo na tela. Os dois primeiros argumentos desse método são a localização, em pixels, do vértice superior esquerdo do retângulo; o terceiro e quarto argumentos especificam a largura e altura do retângulo, respectivamente. Por exemplo:

```
gfx.drawRect(0, 10, 100, 50);
```

desenha um retângulo com vértice superior esquerdo localizado na posição (0, 10) – no topo da janela, 10 pixels para a direita da margem esquerda da janela – tendo largura de 100 pixels e altura de 50 pixels.

A condição do comando while traz uma pequena novidade: ela usa a expressão this.endBoxRow.length, que ainda não conhecemos. Mas isso significa exatamente o que você está imaginando Em Java, arrays são objetos, e, portanto, podem ter variáveis de instância. Esse é o caso da variável, length, definida na classe Array, cujo valor é

igual ao número de elementos do array.

Testando o código

Sem dúvida seria uma boa idéia compilar e testar o programa, uma vez que modificações substanciais foram introduzidas no código. Infelizmente, não temos nenhum retângulo válido no momento, portanto, nada interessante iria acontecer. Para nos ajudar a ver como se comporta o código que escrevemos até agora, vamos modificar a maneira como inicializamos os arrays endBoxRow e endBoxCol, de maneira a produzir uma situação adequada para o teste:

```
private int[] endBoxRow = {0, 9, 9, 0};
private int[] endBoxCol = {0, 0, 9, 9};
```

Tente agora compilar e executar o programa e veja o que acontece. Se você tiver algum problema, dê uma olhadinha no código 15 da seção de códigos incluída no final deste tutorial.
Quando você estiver satisfeito, restabeleça os valores de inicialização dos arrays endBoxRow e endBoxCol:

```
private int[] endBoxRow = {-1, -1, -1, -1};
private int[] endBoxCol = {-1, -1, -1, -1};
```

Validando a posição do navio

Agora que já sabemos desenhar as posições válidas para "extremidade final" de um navio, precisamos saber como calcular quais são essas posições, com base na posição inicial escolhida pelo usuário. O algoritmo para fazer isso envolve os seguintes passos:

1. Obtenha o tamanho do navio corrente.

2. Se a extremidade pode ser posicionada acima da posição inicial escolhida pelo o usuário, coloque uma marca nessa posição; caso contrário, marque com o valor -1 a entrada de índice 0 nos arrays endBoxRow e endBoxCol.

3. Se a extremidade pode ser posicionada à direita da posição inicial escolhida pelo o usuário, coloque uma marca nessa posição; caso contrário, marque com o valor -1 a entrada de índice 1 nos arrays endBoxRow e endBoxCol.

4. Se a extremidade pode ser posicionada abaixo da posição inicial escolhida pelo o usuário, coloque uma marca nessa posição; caso contrário, marque com o valor -1 a entrada de índice 2 nos arrays endBoxRow e endBoxCol.

5. Se a extremidade pode ser posicionada à esquerda da posição inicial escolhida pelo o usuário, coloque uma marca nessa posição; caso contrário, marque com o valor -1 a entrada de índice 3 nos arrays endBoxRow e endBoxCol.
Traduzindo isso em Java, obtemos o seguinte código:

```
public void setFirstRowColumn(int row, int col) {
   if ((row >= 0) &&
      (row < ROW_COUNT) &&
      (col >= 0) &&
      (col < COLUMN_COUNT)) {
```

```
            this.firstClickRow = row;
            this.firstClickCol = col;

            // Obtém o tamanho do navio corrente.
            int testLength =
                  this.shipLength[this.currentUserShip] - 1;

            // Verifica as posições abaixo, acima, à esquerda
            // e à direita da linha e coluna correntes.
            this.validateUpperEndPoint(0, row, col, testLength);
            this.validateRightEndPoint(1, row, col, testLength);
            this.validateLowerEndPoint(2, row, col, testLength);
            this.validateLeftEndPoint(3, row, col, testLength);
        }
        else {
            // Evita marcar 'x' na grade do tabuleiro.
            this.firstClickRow = -1;
            this.firstClickCol = -1;
        }
    }
```

É claro que precisamos ainda escrever o código dos métodos validate***EndPoint , usados acima para verificar a validade de cada uma das possíveis posições calculadas para a extremidade final do navio. Isso se parece bastante com o código que escrevemos anteriormente para posicionamento dos navios pelo computador, no corpo do método placeComputerShips. Vamos dissecar um desses métodos rapidamente, descrever outro deles, e deixar para você a tarefa de implementar os outros dois.
O método a seguir verifica a validade de se posicionar a extremidade final do navio abaixo da posição inicial escolhida pelo usuário, atualizando a entrada correspondente (de índice 2) dos arrays endBoxRow e endBoxCol.

```
private void validateLowerEndPoint(int boxIndex, int row,
                                   int col, int size) {
    // Testa se a extremidade final está dentro dos limites do tabuleiro.
    int newRow = row + size;
    if (newRow < ROW_COUNT) {
        // A extremidade final está dentro dos limites do tabuleiro.
        // Então, verifica se ocorre colisão.
        boolean bCollision = false;
        int i = row;
        while (i <= newRow) {
            if (this.gridCells[i][col] != Ship.TYPE_NONE) {
                bCollision = true;
                break;
            }
            i++;
        }

        if (bCollision == false) {
            this.endBoxRow[boxIndex] = newRow;
            this.endBoxCol[boxIndex] = col;
        }
        else {
```

```
            // Evita desenhar esse quadrado.
            this.endBoxRow[boxIndex] = -1;
            this.endBoxCol[boxIndex] = -1;
        }
    }
    else {
        // Evita desenhar esse quadrado.
        this.endBoxRow[boxIndex] = -1;
        this.endBoxCol[boxIndex] = -1;
    }
}
```

Primeiro, verificamos se a posição calculada para a extremidade final está dentro da grade do tabuleiro. Posicionar a extremidade final "abaixo" da posição inicial escolhida pelo usuário significa que a linha correspondente à extremidade final – newRow – deve ser igual ao valor da linha da posição inicial – row – mais o tamanho do navio – size. Portanto, obtemos:

```
int newRow = row + size;
if (newRow < ROW_COUNT) {
    // ...código do while loop...
}
else {
    // Evita desenhar esse quadrado.
    this.endBoxRow[boxIndex] = -1;
    this.endBoxCol[boxIndex] = -1;
}
```

Se a nova linha calculada – newRow – cai dentro dos limites da grade do tabuleiro (isto é, if (newRow < ROW_COUNT)), então a cláusula if é executada; caso contrário, atribuímos o valor -1 à posição correspondente nos arrays endBoxRow e endBoxCol, evitando assim que seja desenhado um quadrado na posição calculada, uma vez que ele não é válida.
Supondo que a posição calculada para a extremidade final do navio está dentro dos limites da grade, a cláusula if do comando acima é executada, a qual contém o seguinte código:

```
// A extremidade final está dentro dos limites do tabuleiro.
// Então, verifica se ocorre colisão.
boolean bCollision = false;
int i = row;
while (i <= newRow) {
    if (this.gridCells[i][col] != Ship.TYPE_NONE) {
        bCollision = true;
        break;
    }
    i++;
}

if (bCollision == false) {
    this.endBoxRow[boxIndex] = newRow;
    this.endBoxCol[boxIndex] = col;
}
```

```
else {
    // Evita desenhar esse quadrado.
    this.endBoxRow[boxIndex] = -1;
    this.endBoxCol[boxIndex] = -1;
}
```

Esse código é muito semelhante ao que escrevemos anteriormente no corpo do método placeComputerShips. Começamos pela linha correspondenteà posição da extremidade inicial, escolhida pelo usuário e supomos que não ocorre nenhuma colisão, isto é:

```
boolean bCollision = false;
int i = row;
```

Então, percorremos cada uma das posições nas linhas abaixo, até a nova linha calculada:

```
while (i <= newRow) {
   // ...algum código aqui...
   i++;
}
```

Em cada iteração do loop, verificamos se ocorre colisão com outro navio, na posição determinada pela linha e coluna correntes:

```
if (this.gridCells[i][col] != Ship.TYPE_NONE) {
   bCollision = true;
   break;
}
```

Se for detectada colisão em uma dessas posições, a execução do loop é interrompida, pelo comando break. Depois de terminado o loop, verifica-se se foi detectada alguma colisão, Se isso não ocorreu, a linha e coluna da posição calculada é armazenada na entrada correspondente nos arrays endBoxRow e endBoxCol. Se tiver ocorrido uma colisão, essa posição é invalidada, atribuindo-se o valor -1 à entrada correspondente nos arrays endBoxRow e endBoxCol:

```
if (bCollision == false) {
    this.endBoxRow[boxIndex] = newRow;
    this.endBoxCol[boxIndex] = col;
}
else {
    // Evita desenhar esse quadrado.
    this.endBoxRow[boxIndex] = -1;
    this.endBoxCol[boxIndex] = -1;
}
```

Compare agora esse código com o do método para validar a extremidade final do navio à esquerda da posição inicial escolhida pelo usuário:

```
private void validateLeftEndPoint(int boxIndex, int row,
                        int col, int size) {
   // Testa se a extremidade final está dentro dos limites do tabuleiro.
   int newCol = col - size;
   if (newCol >= 0) {
       // A extremidade final está dentro dos limites do tabuleiro.
```

```
        // Então, verifica se ocorre colisão.
        boolean bCollision = false;
       int i = col;
        while (i >= newCol) {
            if (this.gridCells[row][i] != Ship.TYPE_NONE) {
              bCollision = true;
             break;
           }
           i—;
       }

        if (bCollision == false) {
            this.endBoxRow[boxIndex] = row;
            this.endBoxCol[boxIndex] = newCol;
       }
       else {
           // Evita desenhar esse quadrado.
            this.endBoxRow[boxIndex] = -1;
            this.endBoxCol[boxIndex] = -1;
       }
   }
   else {
       // Evita desenhar esse quadrado.
        this.endBoxRow[boxIndex] = -1;
        this.endBoxCol[boxIndex] = -1;
   }
}
```

A única novidade no corpo desse método é o operador "—". Ele é o oposto do operador "++": enquanto "++" incrementa o valor do operando de 1, o operador "—" decrementa de 1 o valor do operando.

Procure entender detalhadamente o código do método validateLeftEndPoint – isso não deverá ser complicado. Uma vez que você tenha entendido, você estará pronto para implementar, sozinho, os métodos validateRightEndPoint e validateUpperEndPoint. Depois de fazer isso, tente compilar e executar o seu programa. Se você tiver alguma dificuldade, dê uma olhadinha no código 16 da seção de códigos incluída no final deste tutorial.

Depois que você tiver terminado, execute e teste o seu programa. Você deverá ver aparecerem os marcadores de posição inicial e posições válidas para a extremidade final de cada navio, como resposta aos cliques do mouse sobre o tabuleiro do jogo.

Um toque final

Há ainda uma tarefa a ser realizada! Por enquanto, independentemente do número de vezes que o usuário clicar sobre o tabuleiro, o programa nunca posiciona nem mesmo o primeiro navio. O problema é que precisamos ainda adicionar ao programa o código para armazenar a posição do navio corrente e passar para o próximo navio, quando o usuário seleciona uma das possíveis posições válidas para a extremidade final do navio.

O algoritmo para o procedimento relacionado ao "segundo clique" do usuário pode ser descrito do seguinte modo:

1. Armazene os valores da linha e coluna da extremidade final selecionada.

2. Comece pela possível extremidade final calculada acima da posição inicial do navio.

3. Se a linha e a coluna são iguais nessas duas posições, então posicione aí o navio.

4. Caso contrário passe para a próxima extremidade final calculada.

5. Se ainda existir alguma extremidade final calculada a ser examinada, vá para o passo 3.

6. Se a posição selecionada pelo usuário não corresponde a nenhuma das possíveis extremidades finais calculadas, então reposicione o marcador de posição inicial - 'x' – e prepare para a próxima escolha da posição da extremidade final pelo usuário.

Muito bem... Para que o programa possa fazer isso, será necessário adicionar o seguinte código à classe BattleshipApp:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
         this.blueBoard.placeComputerShips();
       this.repaint();
       this.initializeSetupState();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       if (this.bFirstSetupClick) {
            // Armazena a linha e coluna correspondentes
            // à posição do primeiro clique do mouse.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
             this.redBoard.setFirstRowColumn(row, col);
           this.bFirstSetupClick = false;
           this.repaint();
      }
      else {
            // Armazena a linha e coluna correspondentes
            // à posição do segundo clique do mouse.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
             this.redBoard.setSecondRowColumn(row, col);
           this.repaint();
           this.bFirstSetupClick = true;
      }
   }
}
```

Isso requer que o seguinte código seja também adicionado à classe Board:

```
public void setSecondRowColumn(int row, int col) {
   if ((row >= 0) &&
       (row < ROW_COUNT) &&
       (col >= 0) &&
       (col < COLUMN_COUNT)) {
        // Se a posição selecionada pelo usuário for uma
```

```
        // das quatro posições permitidas para a extremidade
        // do navio, então posicione o navio; caso contrário,
        // volte para a escolha da posição inicial 'x'.
    }
}
```

Como sempre, ainda teremos que escrever o código que implementa a tarefa especificada pelo comentário destacado acima, em azul. Basicamente, o que precisamos fazer é traduzir para Java os passos 2 a 6 do algoritmo que acabamos de descrever. Será que isso seria difícil?

Pela descrição do algoritmo, fica claro que vamos precisar de um loop para iterar sobre as possíveis posições calculadas para a extremidade final do navio. Para cada uma dessas posições, devemos comparar sua linha e coluna, respectivamente, com a linha e coluna da posição selecionada pelo usuário. Se em algum dos casos esses valores casarem, então interrompemos o loop e podemos adicionar o navio ao tabuleiro. Se, ao terminar o loop, não tiver havido casamento desses valores, em nenhum dos casos, devemos reposicionar o marcador de posição inicial do navio - 'x' – nessa nova posição selecionada pelo usuário.
O código para fazer isso é apresentado a seguir:

```
public void setSecondRowColumn(int row, int col) {
    if ((row >= 0) &&
        (row < ROW_COUNT) &&
        (col >= 0) &&
        (col < COLUMN_COUNT)) {

        // Começa pela primeira extremidade final calculada.
        boolean bPlacedShip = false;
       int i=0;
        while (i < this.endBoxRow.length) {
            // Verifica se a posição selecionada pelo usuário
            // é essa extremidade final calculada.
            if ((row == this.endBoxRow[i]) &&
                (col == this.endBoxCol[i])) {
                // Posiciona o navio e
                // passa para o próximo navio.
                 int shipType = shipType[this.currentUserShip];
                 int size = shipLength[this.currentUserShip];
                Ship newShip = new Ship(shipType, i,
                                        this.firstClickRow,
                                        this.firstClickCol,
                                        size);
                this.addShip(newShip);
                this.currentUserShip++;
                bPlacedShip = true;
               break;
            }
            i++;
        }

        if (bPlacedShip == false) {
            // Se o usuário não clicou sobre nenhuma das
```

```
            // extremidades finais válidas, então move o
            // marcador de posição inicial 'x' para a posição selecionada.
            this.setFirstRowColumn(row, col);
        }
    }
}
```

Você já viu código bastante parecido anteriormente, por isso, não será difícil entender este. Vamos apenas esclarecer o seguinte: note que referenciamos os arrays shipType e shipLength sem usar a palavra chave this. Podemos fazer isso porque this sempre é opcional em Java, embora algumas vezes seja necessário para distinguir duas variáveis que tenham o mesmo nome. Se você quiser saber mais sobre isso, veja o apêndice 2 deste módulo. Entretanto, talvez seja melhor não interromper nossa linha de raciocínio agora, e dar uma olhada nesse apêndice mais tarde..

Bem, vá em frente: compile e execute o seu programa, depois de fazer as modificações acima. Clique sobre o tabuleiro vermelho e veja o que acontece. Você provavelmente vai perceber que tudo funciona quase como deveria ser... mas não completamente. Além disso, se você posicionar o último navio, e continuar clicando, vai descobrir um erro. O que podemos fazer para sanar esses problemas?

Solucionando alguns dos problemas

Quase completamos nossa tarefa de escrever o código que possibilita ao usuário posicionar sua esquadra – apenas temo que corrigir alguns pequenos problemas. O primeiro problema é que os marcadores das possíveis posições para a extremidade de um navio não desaparecem, depois que selecionamos uma dessas extremidades. Esse problema é fácil de ser corrigido. Basta adicionar o seguinte método, à classe Board:

```
private void clearEndBoxes() {
    int i=0;
    while (i < this.endBoxRow.length) {
        this.endBoxRow[i] = -1;
        this.endBoxCol[i] = -1;
        i++;
    }
}
```

e incluir uma chamada desse método no corpo do método setSecondRowColumn.

O segundo problema é mais difícil de ser detectado. Se clicamos sobre o tabuleiro, marcando um 'x' e, em seguida, clicamos fora de uma das possíveis posições para a extremidade final do navio, então a marca 'x' se move para a nova posição do clique, como esperado. Mas, depois disso, algumas vezes em que clicamos sobre uma posição de extremidade final, o navio é posicionado, e, em outras vezes, a marca 'x' se move novamente. Hummm...

Esse problema está relacionado com a classe BattleshipApp. Lembra-se que o código que escrevemos anteriormente para o método mouseReleased alternava sempre entre os casos de "primeiro clique" e "segundo clique"? É isso que está causando o problema agora.

Para entender o que está acontecendo, precisamos descrever cuidadosamente o comportamento do programa que desenvolvemos até agora. Um objeto da classe BattleshipApp se comporta do seguinte modo, no estado de preparação do jogo:

1. Aguarde por um "primeiro clique" do mouse, que seleciona uma posição inicial de navio.

2. Quando detectar "primeiro clique", exiba a marca 'x' e os quadrados marcadores das possíveis posições da extremidade final do navio, e aguarde por um "segundo clique" do mouse, que seleciona a posição da outra extremidade do navio.

3. Quando detectar um "segundo clique", tente posicionar o navio e aguarde novamente por um "primeiro clique".

O terceiro passo acima é a causa do problema. No "primeiro clique" do mouse, são exibidos o marcador 'x' e os quadrados marcadores das possíveis posições da extremidade final do navio. No terceiro passo, se o usuário não posicionou o navio, a marca 'x' é reposicionada e os quadrados marcadores de possíveis posições para a extremidade final são novamente exibidos – isso significa que estamos o programa deveria estar pronto para esperar que o usuário selecione a extremidade final, isto é, deveria aguardar por um "segundo clique" novamente.

Essa idéia de "primeiro clique" e "segundo clique" está nos atrapalhando. Vamos reescrever nosso algoritmo do seguinte modo:

1. Aguarde por um clique que posiciona a marca 'x' – vamos chamá-lo de "positionClick".

2. Quando detectar um "positionClick", exiba a marca 'x' e e os quadrados marcadores das possíveis posições da extremidade final do navio, e aguarde por um clique que seleciona a extremidade final do navio – vamos chamá-lo de "endpointClick".

3. Se o próximo clique não ocorre em uma posição marcada por um dos quadrados, reposicione a marca 'x' e os marcadores quadrados e aguarde por um "endpointClick". Caso contrário, posicione o navio e aguarde por um "positionClick".
Agora está muito mais claro! Tendo esse algoritmo em mente, vamos modificar um pouco a nossa classe BattleshipApp.

```
// variáveis de instância
private Board redBoard;                  // tabuleiro vermelho
private Board blueBoard;                 // tabuleiro azul
private FriendlyPlayer friendlyPlayer;   // jogador aliado
private EnemyPlayer enemyPlayer;         // jogador inimigo
private int gameState;                   // estado do jogo
private boolean bNeedPositionClick;   // tipo do clique que está sendo aguardado
```

e

```
private void initializeSetupState() {
       this.bNeedPositionClick = true;
}
```

e

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
         this.blueBoard.placeComputerShips();
       this.repaint();
       this.initializeSetupState();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
```

```
        if (this.bNeedPositionClick) {
            // Armazena a linha e a coluna do clique
            // que seleciona a posição inicial do navio.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
            this.redBoard.setFirstRowColumn(row, col);
            this.repaint();
            this.bNeedPositionClick = false;
        }
        else {
            // Armazena a linha e a coluna do clique
            // que seleciona a extremidade final do navio.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
            this.redBoard.setSecondRowColumn(row, col);
            this.repaint();
            this.bNeedPositionClick = true;
        }
    }
}
```

Note que ainda não resolvemos o problema – apenas modificamos o nome de uma variável. Essa alteração nos ajuda a entender melhor qual é a correção que deve ser feita no código. E qual é essa correção?
Primeiro, vamos modificar o código do método mouseReleased:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
        this.blueBoard.placeComputerShips();
        this.repaint();
        this.initializeSetupState();
    }
    else if (this.gameState == GAME_STATE_SETUP) {
        if (this.bNeedPositionClick) {
            // Armazena a linha e a coluna do clique
            // que seleciona a posição inicial do navio.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
            this.bNeedPositionClick =
                this.redBoard.setFirstRowColumn(row, col);
            this.repaint();
        }
        else {
            // Armazena a linha e a coluna do clique
            // que seleciona a extremidade final do navio.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
            this.bNeedPositionClick =
                this.redBoard.setSecondRowColumn(row, col);
            this.repaint();
        }
```

```
    }
}
```

Note que esperamos que os métodos setFirstRowColumn e setSecondRowColumn retornem um valor booleano (isto é, true ou false). O valor retornado é atribuído à variável bNeedPositionClick. Em outras palavras, queremos que setFirstRowColumn retorne true sempre que o usuário clicar em uma posição inválida. Queremos que setSecondRowColumn retorne true sempre que o usuário posicionar o navio com sucesso.

Essa descrição deve ser suficiente para que você mesmo faça as modificações necessárias. E então? Vamos tentar? Se você tiver problemas, poderá encontrar a solução no código 17 da seção de códigos incluída no final deste tutorial.

Quando você tiver concluído sua tarefa, compile e execute o seu programa novamente. Você verá que ainda falta um erro a ser corrigido – evitar que se possa tentar posicionar um novo navio, depois de já ter posicionado o bote.

Solução do ultimo problema

Essa seção está mesmo muito extensa – mas já estamos quase chegando ao final! Nosso último problema é fácil de corrigir, mas requer uma volta às origens – está relacionado com o estado do jogo. De maneira simplificada, o que precisamos é verificar, cada vez que o usuário posiciona um navio, se existe ainda algum navio restante – se houver, o programa deve permanecer no estado de preparação do jogo - setup state – caso contrário, deve entrar no estado em que o usuário e o computador vão poder atirar nos navios adversários - play state. Isso é simples.
Primeiro, adicione o seguinte método à classe Board:

```
public int getShipCount() {
    return this.currentUserShip;
}
```

Em seguida, modifique a classe BattleshipApp do seguinte modo:

```
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
       this.drawTitleScreen();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       this.drawGrids();
   }
    else if (this.gameState == GAME_STATE_PLAYING) {
       this.drawGrids();
   }
}
```

e

```
pubic void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
       this.gameState = GAME_STATE_SETUP;
        this.blueBoard.placeComputerShips();
      this.repaint();
```

```
            this.initializeSetupState();
        }
        else if (this.gameState == GAME_STATE_SETUP) {
            if (this.bNeedPositionClick) {
                // Armazena a linha e a coluna do clique
                // que seleciona a posição inicial do navio.
                int row = this.convertYtoRow(event.getY());
                int col = this.convertXtoCol(event.getX());
                this.bNeedPositionClick =
                    this.redBoard.setFirstRowColumn(row, col);
                this.repaint();
            }
            else {
                // Armazena a linha e a coluna do clique
                //que seleciona a extremidade final do navio.
                int row = this.convertYtoRow(event.getY());
                int col = this.convertXtoCol(event.getX());
                this.bNeedPositionClick =
                    this.redBoard.setSecondRowColumn(row, col);
                this.repaint();

                if (this.redBoard.getShipCount() == 5) {
                    this.gameState =
                        BattleshipApp.GAME_STATE_PLAYING;
                }
            }
        }
    }
```

Finalmente!

Finalmente terminamos de escrever o código que possibilita ao usuário posicionar a sua esquadra. Compile, execute e brinque! Se você tiver algum problema, confira o seu programa com o código 18 da seção de códigos incluída no final deste tutorial.

Ufa! Tratamos de problemas bem complicados neste módulo! Você certamente vai querer descansar um pouco. Entretanto, ainda não podemos parar por aqui. Até agora, se um usuário sentar-se em frente ao nosso jogo, ele não terá a menor idéia de como funciona o processo de posicionamento de navios no tabuleiro. Nosso jogador irá precisar de instruções para orientá-lo. Portanto, vamos escrever um código bastante simples para implementar uma "janela de mensagens" para fornecer instruções sobre o jogo para o usuário.

Depois que você tiver descansado um pouco, anime-se para prosseguir com próxima seção.

Exibindo mensagens

Freqüentemente, em um programa, é desejável exibir informações para o usuário. No nosso caso, queremos exibir instruções sobre como ele deve proceder para posicionar sua esquadra no tabuleiro.

Existem diversas maneiras para fazer isso. Vamos descrever aqui, uma maneira bastante simples: vamos criar uma janela de mensagens na parte inferior da janela do jogo. A maior parte do código para fazer isso já foi visto anteriormente. Por isso, não vamos explicar tudo novamente, com muito detalhe.

Uma janela para o mundo

Primeiramente, vamos definir os parâmetros da nossa janela. Adicione as seguintes variáveis de classe à classe BattleshipApp:

```
private static final int MSG_WINDOW_DX = GAME_WIDTH;
private static final int MSG_WINDOW_DY = 50;
private static final int MSG_WINDOW_X = 0;
private static final int MSG_WINDOW_Y = GAME_HEIGHT-MSG_WINDOW_DY;
```

Note que definimos os valores dessas constantes em termos de outras constantes. Por exemplo, o valor de MSG_WINDOW_Y depende do valor de GAME_HEIGHT e MSG_WINDOW_DY (lembre-se que DX refere-se a largura e DY a altura).
Em seguida, vamos adicionar uma variável, de tipo String, para armazenara a mensagem que queremos exibir na janela. Adicione a seguinte variável de instância na classe BattleshipApp:

```
private String userMessage = null;
```

Vamos precisar de um método para exibir a mensagem na janela:

```
/**
 *  Exibe uma mensagem na janela de mensagens do jogo.
 */
public void drawMessage(Graphics gfx) {
    int x = MSG_WINDOW_X;
    int y = MSG_WINDOW_Y;
    int dx = MSG_WINDOW_DX;
    int dy = MSG_WINDOW_DY;

    // Apaga a mensagem antiga.
     gfx.setColor(Color.black);
    gfx.drawRect(x, y, dx, dy);

    // Desenha um retângulo do tamanho da janela.
     gfx.setColor(Color.gray);
    gfx.drawRect(x, y, dx, dy);

    // Exibe a mensagem dentro do retângulo.
    if (this.userMessage != null) {
        gfx.drawString(this.userMessage, x + 2, y + 20);
    }
}
```

Vamos também precisar de alguma maneira para especificar a mensagem a ser exibida. O seguinte método cuida disso:

```
public void setMessage(String newMessage, boolean bDraw) {
    this.userMessage = newMessage;
```

```
    if (bDraw) {
        this.repaint();
    }
}
```

Finalmente temos que garantir que o método que exibe a mensagem seja chamado quando o programa se encontra nos diferentes estados:

```
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.drawTitleScreen();
    }
    else if (this.gameState == GAME_STATE_SETUP) {
        this.drawGrids();
        this.drawMessage(gfx);
    }
    else if (this.gameState == GAME_STATE_PLAYING) {
        this.drawGrids();
        this.drawMessage(gfx);
    }
}
```

Finalmente, podemos usar nosso noveo código para exibir mensagens. Vamos instruir o usuário sobre como posicionar um navio, quando se entra no estado de preparação do jogo e vamos exibir outra mensagem, quando é iniciado o estão de “jogar”.

```
/**
 *  Incialização para os tabuleiros.
 */
private void initializeSetupState() {
    this.bNeedPositionClick = true;
          // Exibe instruções.
    String instructions = “Clique duas vezes sobre a grade vermelha para posicionar os
navios.” +
                          “Quadrados verdes marcam as extremidades finais válidas para o
navio.”;
    this.setMessage(instructions, true);
}
```

e

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
        this.blueBoard.placeComputerShips();
        this.repaint();
        this.initializeSetupState();
    }
    else if (this.gameState == GAME_STATE_SETUP) {
        if (this.bNeedPositionClick) {
            // Armazena a linha e coluna correspondentes
            // à posição do primeiro clique do mouse.
            int row = this.convertYtoRow(event.getY());
            int col = this.convertXtoCol(event.getX());
```

```
                this.bNeedPositionClick =
                    this.redBoard.setFirstRowColumn(row, col);
                this.repaint();
            }
            else {
                // Armazena a linha e coluna correspondentes
                // à posição do segundo clique do mouse.
                int row = this.convertYtoRow(event.getY());
                int col = this.convertXtoCol(event.getX());
                this.bNeedPositionClick =
                    this.redBoard.setSecondRowColumn(row, col);
                this.repaint();

                if (this.redBoard.getShipCount() == 5) {
                    this.gameState = BattleshipApp.GAME_STATE_PLAYING;
                    this.setMessage("COMECE A JOGAR!", true);
                }
            }
        }
    }
}
```

Pronto, você pode agora compilar e executar o seu programa. Se o seu programa não funcionar corretamente, você poderá verificá-lo, comparando-o código 19 da seção de códigos incluída no final deste tutorial.

Você deve ter percebido que a janela de mensagem não funciona exatamente como esperado. Não vamos corrigir esse problema por enquanto, mas veja se você consegue descobrir a razão do problema. Aqui está uma dica – esse problema está relacionado com o conteúdo do apêndice 1 deste módulo, sobre conversão de coordenadas em pixels, para linha e coluna da grade, que mencionamos anteriormente.

Vamos rever agora o que aprendemos neste módulo

Conclusão

Puxa! Este módulo deu um bocado de trabalho, não? Vimos algoritmos novos, assim como mais algumas regras de sintaxe e semântica da linguagem Java.

Aprendemos a lidar com números aleatórios, que ocorrem com freqüência na implementação de jogos, e os utilizamos na implementação do programa de posicionamento dos navios do computador. Também escrevemos o código para permitir ao usuário posicionar sua esquadra no tabuleiro, tratando eventos do mouse.

Agora estamos prontos para começar a jogar! Nos módulos finais, vamos escrever o código que implementa isso. Ele envolve diversos aspectos interessantes, como alguns detalhes novos sobre interfaces gráficas, como manter informação sobre o estado de um navio, como fazer o computador atirar em um navio do jogador, como verificar se um navio foi afundado etc.

Mas antes de começar o próximo módulo, brinque um pouco com o código do seu jogo. Afinal, para que serve um jogo, senão para brincar?

Nos vemos então no próximo módulo.

Apêndice 1

Conversão de coordenadas em pixels para (linha,coluna) da grade do tabuleiro

O problema tratado aqui é a conversão das coordenadas (em pixels) de um ponto sobre a janela do jogo na linha e coluna correspondentes do tabuleiro. Como isso pode ser feito? Esse é um problema que envolve vários aspectos. Temos que saber como converter coordenadas em linhas e colunas, como levar em conta o fato de que a grade do jogo não está posicionada na origem do sistema de coordenadas, e como ajustar a posição do clique do mouse, levando em conta as áreas de título e de bordas janela. Vamos tratar desses problemas, um de cada vez.

Convertendo coordenadas em pixels para linhas e colunas

Vamos começar pelo seguinte problema: dadas as coordenadas em pixels de um ponto da grade, como podemos obter a linha e coluna da grade correspondentes? Considere o exemplo mostrado na figura 1, a seguir.

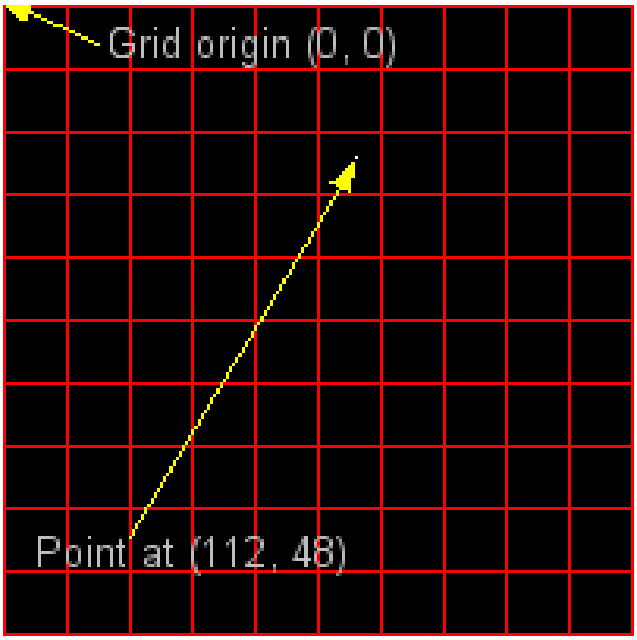


Figura 1— Obtendo a linha e coluna do ponto de coordenadas (112, 48).

Primeiro, vamos nos concentrar apenas no valor da coluna. Sabemos que cada coluna da grade tem largura de 20 (você se lembra da nossa constante COLUMN_PIXEL_WIDTH?). Então, é claro que qualquer ponto com coordenada x na faixa de 0 a 19 estará na primeira coluna (que chamaremos de coluna 0, porque Java numera objetos a partir de 0, e não de 1). De maneira análoga, qualquer ponto com coordenada x entre 20 e 39 estará na segunda coluna (coluna 1). Um x de 40-59 estará na coluna 2 etc. Essa conversão pode ser feita facilmente, pela seguinte fórmula:

```
int column = x / Board.COLUMN_PIXEL_WIDTH;
```

Entretanto, existem alguns problemas escondidos sob o comando acima. Primeiro, suponha x=30 e COLUMN_PIXEL_WIDTH=20. A divisão acima teria resultado 1.5. Como poderíamos ter uma coluna 1.5? Hummm.... Felizmente, não precisamos nos preocupar com esse problema: note que a variável column é declarada com tipo int. Lembre-se que

o tipo int representa valores inteiros. Em Java, quando atribuímos um valor tal como 1.5 a uma variável inteira, esse valor é automaticamente truncado, para o valor inteiro imediatamente inferior, nesse caso, 1 (que corresponde à segunda coluna, ma vez que a primeira coluna é numerada por 0).

Vejamos então se a nossa formula funciona para o ponto mostrado na figura 1, acima. A coordenada x desse ponto é 112 pixels. Temos: 112 / 20 = 5, o que corresponde à sexta coluna. Contando as colunas na figura 1, vemos que o ponto de fato está na sexta coluna. Perfeito!
De maneira análoga, a linha pode ser determinada pela seguinte formula:

```
int row = y / Board.ROW_PIXEL_HEIGHT;
```

Origem do eixo de coordenadas

Agora, temos que compensar o fato de que a origem do sistema de coordenadas da grade não corresponde à origem do sisitema no qual obtemos as coordenadas do clique do mouse. A figura 2, a seguir, ilustra esse problema.

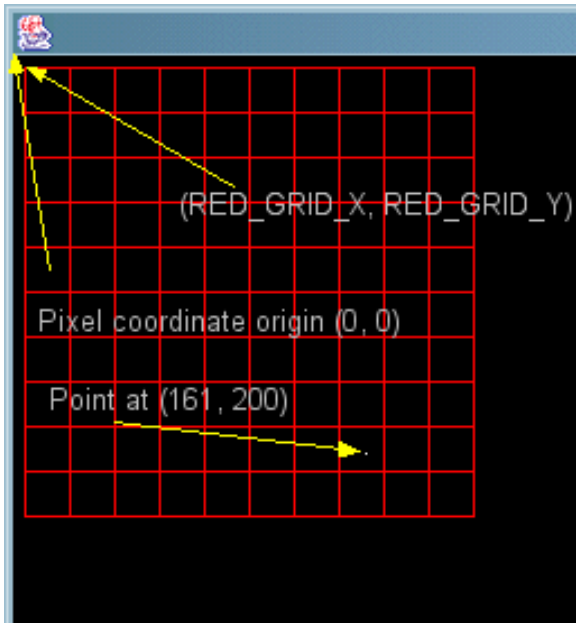


Figura 2—A grade não está na origem do sistema de coordenadas.

Um exemplo concreto vai nos ajudar a enxergar a solução deste problema. Sabemos que o canto superior esquerdo da grade corresponde ao ponto de coordenadas RED_GRID_X e RED_GRID_Y da janela do jogo (essas duas constantes têm valor 5, na nossa implementação). Mais uma vez, considere o caso das colunas. Podemos ver que qualquer ponto de coordenada x na faixa de 5 a 24 estaria na coluna 0. Pontos com coordenada x entre 25-44 estariam na coluna 1 etc. De fato, a única diferença entre este caso e o que tratamos anteriormente é o deslocamento de 5 pixels, resultante da posição da grade em relação à borda da janela do jogo. Portanto, a seguinte linha de código poderia resolver esse problema:

```
int column = (x - RED_GRID_X) / Board.COLUMN_PIXEL_WIDTH;
```

Para ver como esse commando funciona, considere um ponto de coordenadas (161,

200). Primeiro, subtrímos o valor RED_GRID_X de 161. Como RED_GRID_X = 5, obtemos 156. Em outras palavras, se o ponto está 161 pixels à direita da margem esquerda da janela do jogo, e a grade começa a 5 pixels da margem esquerda da janela, então ele está a 156 pixels da linha mais à esquerda da grade. Uma vez obtida a coordenada x do ponto, relativa à margem esquerda da grade, estmoa de volta ao problema inicial. Basta dividir 156 por 20, obtendo 7 – o que significa que o ponto está na oitava coluna. Verifique na figure 2.
O commando para calcular a coluna é análogo:

```
int row = (y - RED_GRID_Y) / Board.ROW_PIXEL_HEIGHT;
```

Janelas dentro de janelas

O ultimo detalhe da nossa conversão está relacionado com uma decisão que tomamos ao implementar a interface do nosso jogo. Você se lembra que o nosso tabuleiro é desenhado em um objeto da classe ContentPane? Um objeto dessa classe representa a parte da janela que denominamos área de conteúdo (ou área de cliente). Essa area não inclui a barra de título ou as margens da janela. Infelizmente, as coordenadas da posição do mouse, que obtemos no método mouseReleased são relatives ao canto superior esquerdo da janela, e não da sua área de conteúdo. Isso é ilustrado na figura 3, a seguir.

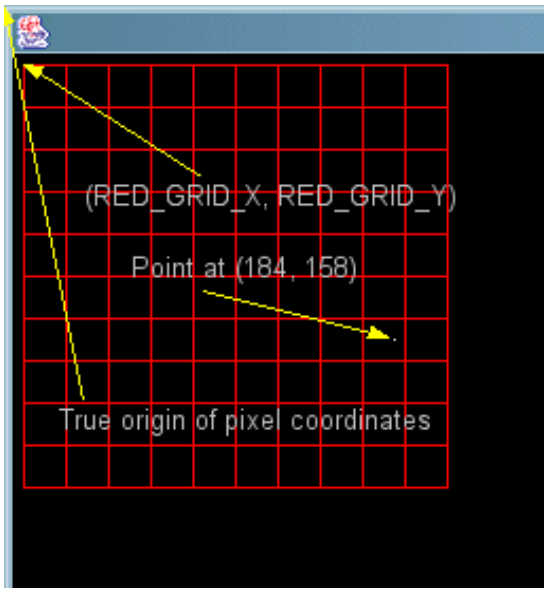


Figura 3— Levando em conta a barra de título e as margens da janela.

O único problema aqui é calcular as dimensões da barra de tituo e das margens da janela. Felizmente, isso é muito fácil. A idéia chave é comparar as dimensões da janela e da sua área de conteúdo. Veja a figura 4, a seguir:

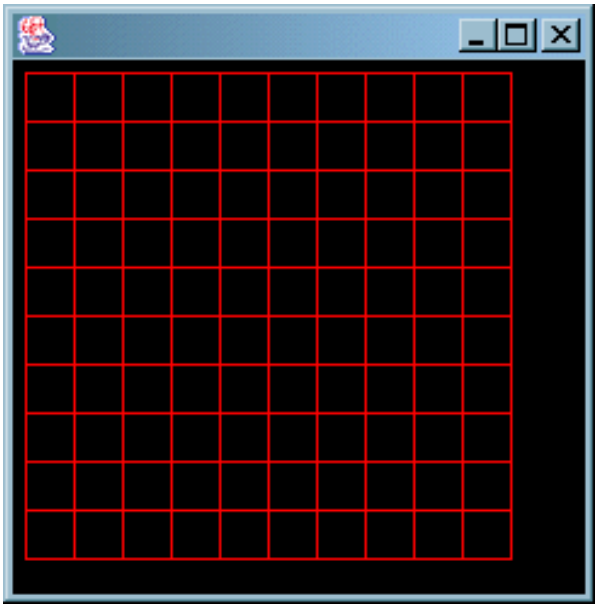


Figura 4—Área da janela versus área de conteúdo.

Note que a janela é mais larga que a área de conteúdo de um valor igual a duas vezes a largura das margens. Note também que a altura da janela excede a altura da área de conteúdo por um valor que é igual à altura da margem mais a altura da barra de título. O seguinte trecho de código (na classe BattleshipApp) nos permite calcular as dimensões das bordas e da barra de título:

```
Container clientArea = this.getContentPane();
int borderSize = (this.getWidth() - clientArea.getWidth()) / 2;
int titleSize = (this.getHeight() - clientArea.getHeight()) - borderSize;
```

A largura da margem – borderSize – é calculada como a metade da diferença entre as larguras da janela e da sua área de conteúdo. A altura da barra de título – titleSize – é calculada como diferença entre a altura da janela e da sua área de conteúdo, menos a largura da margem (inferior).
Podemos agora usar esses valores para compensar a posição da grade. Isso pode ser feito exatamente como fizemos anteriormente:

```
int column = (x - borderSize - RED_GRID_X) / Board.COLUMN_PIXEL_WIDTH;
int row = (y - titleSize - RED_GRID_Y) / Board.ROW_PIXEL_HEIGHT;
```

Vamos ver se nossa formula para cálculo da coluna funciona corretamente no caso do ponto mostrado na figura 3. O valor obtido para a largura da margem é 4 pixels e a altura da barra de título é 23 pixels. Portanto, column = (184 - 4 - 5) / 20 = 175 / 20 = 8, o que significa que o ponto deve estar na nona coluna, da esquerda para a direita – é o que de fato acontece.
Isso então resolve o nosso problema. O código final dos método de conversão é o seguinte:

```
private int convertYtoRow(int pixelY) {
    Container clientArea = this.getContentPane();
    int borderSize = (this.getWidth() - clientArea.getWidth()) / 2;
    int titleSize = (this.getHeight() - clientArea.getHeight()) - borderSize;
   int row = (pixelY - titleSize - RED_GRID_Y) / Board.ROW_PIXEL_HEIGHT;
```

```
  return row;

}

private int convertXtoCol(int pixelX) {
    Container clientArea = this.getContentPane();
    int borderSize = (this.getWidth() - clientArea.getWidth()) / 2;
         int column = (pixelX - borderSize - RED_GRID_X) /
Board.COLUMN_PIXEL_WIDTH;

 return column;
}
```

Detalhes, Detalhes...

Antes de terminar esta seção, temos um ultimo – mas muito importante – detalhe a considerar. Note o que destacamos em azul no código acima.
Você já está familiarizado com o tipo int. O cabeçalho do método:

```
private int convertXtoCol(int pixelX) {
```

indica que ele produz como resultado um valor inteiro. Note que a declaração int column = ... especifica que a variável col armazena valores inteiros. A execução do comando return col; causa o término da execução da chamada do método (retornando a execução do programa para o ponto do programa em que ocorreu a chamada) e returnando, como resultado, o valor armazenado na variável col. O mesmo vale no caso do método convertYtoRow. Portanto, suponha que convertYtoRow seja chamado, tendo como argumento o ponto mostrado na figura 3. Podemos ver, nessa figura, que esse ponto está na linha 6 (isto é, na sétima linha), portanto, int row = this.convertYtoRow(event.getY()); resultaria na seguinte atribuição: int row = 6;
Existe ainda mais um pequeno detalhe... Se você reparar no código dos métodos convertXtoColumn e convertYtoRow, verá que referenciam as constantes Board.COLUMN_PIXEL_WIDTH e Board.ROW_PIXEL_HEIGHT. Até agora, essas constantes eram declaradas, na classe Board, com o atributo private. Para que sejam visíveis na classe BattleshipApp, temos que modificar o seu atributo para public. Ou seja, as declarações de variáveis de classe, na classe Board, seriam:

```
// variáveis de classe.
private static final int ROW_COUNT        = 10;
private static final int COLUMN_COUNT      = 10;
public static final int ROW_PIXEL_HEIGHT    = 20;
public static final int COLUMN_PIXEL_WIDTH  = 20;
private static final int SHIPS_PER_FLEET    = 5;
```

Isso é tudo

Apêndice 2

Escopo de variáveis

Em programação, você frequentemente vai ouvir falar de "escopo" de uma variável. Ë um conceito importante, particularmente em linguagens orientadas a objetos, tais como Java. O que isso significa?
Quando declaramos uma variável em um programa, ela é "visível" (isto é, pode ser usada) apenas em uma determinada seção do código do programa. Em particular, essa variável pode ser usada apenas dentro do trecho de código delimitado pelo par de chaves que a envolve imediatamente. Por exemplo:

```
{
 // Seção 1.
  int myInt = 10;
   {
      // Seção 2.
     float myFloat = -5;
   }
}
```

Nesse exemplo, a variável myInt é visível nas seções 1 e 2, mas myFloat é visível apenas na seção 2. Portanto, o seguinte código é válido:

```
{
   // Seção 1.
   int myInt;
      myInt++;
   {
      // Seção 2.
      float myFloat;
      myFloat += 1;
      myInt = myInt - 3;
   }
   myInt—;
}
```

Mas o código a seguir não é válido, originando um erro de compilação:

```
{
   // Seção 1.
   int myInt = 10;
   {
      // Seção 2.
      float myFloat = -5;
   }
   // ERRO! A vari;avel myFloat não é visível for a da seção 2.
   myFloat = 3;
}
```

Você poderá estar pensando: "Humm...E porque eu teria que me preocupar com isso?" Existem várias razões. Algumas delas você irá perceber, à medida que continuar desen-

volvendo outros programas. Mas já nos deparamos com pelo menos uma. Considere a seguinte classe:

```
public class MyClass {
   int number;
    public MyClass(int number) {
       this.number = number;
   }
}
```

Note que temos uma variável de instância chamada number, e, no construtor, um parâmetro também chamado number. De acordo com as regras de escopo, ambos são visíveis no corpo do construtor. Em Java, quando duas variáveis com o mesmo nome são visíveis dentro do mesmo escopo, todo uso desse nome referencia a variável que foi declarada mais recentemente. No exemplo acima, o parâmetro foi introduzido depois da variável de instância e, portanto, o uso de number no corpo do construtor referencia o parâmetro do método. Se quisermos referenciar a variável de instância, temos que preceder o seu nome pela palavra reservada this – a palavra reservada this é uma referência ao objeto corrente (da classe MyClass) e, portanto, this.number referencia a variável de instância number desse objeto. Portanto, o comando:

```
this.number = number;
```

atribui, à variável de instância number, o valor passado como argumento para o método.

Módulo 8

Mais sobre Tratamento de Eventos

Introdução

No último modulo, implementamos a etapa de preparação do jogo, aquela que exibe os tabuleiros e possibilita aos dois jogadores – o usuário e o computador – posicionem seus navios na grade do tabuleiro.

Bem, agora está na hora de começar a jogar! Para isso, precisamos programar nosso jogo para implementer as seguintes funções:

- Exibir uma janela com informações sobre o estado dos navios de cada jogador
- Possibilitar ao usuário atirar sobre navios do adversário (o computador)
- Possibilitar ao computador atirar sobre navios do adversário (o usuário)

Você deve estar ansioso... então, vamos lá!

Exibindo o estado do jogo

Informação sobre o estado do jogo

Para planejar suas jogadas, o jogador deve estar informado sobre o estado do jogo, isto é sobre os danos sofridos por sua esquadra e pela esquadra adversária. Para fornecer essa informação, vamos implementar uma janela de estado do jogo, seguindo os seguintes passos:

- Criar uma classe que descreve a janela de estado – StatusDisplay.
- Adicionar à classe Ship variáveis necessárias para manter informação sobre os danos sofridos por navios.
- Escrever código para usar os dados armazenados na classe Ship para gerar a informação a ser exibida na janela de estado.

A janela de estado

Primeiramente, vamos criar uma nova classe – StatusDisplay – que irá conter o código para gerenciamento da janela do estado do jogo. Você já sabe como fazer isso – crie uma nova classe, usando o BlueJ, apague o código gerado automaticamente e preencha os comentários de maneira adequada. Se você tiver se esquecido do procedimento para fazer isso, a figura 1, a seguir, poderá ajudá-lo a refrescar sua memória.

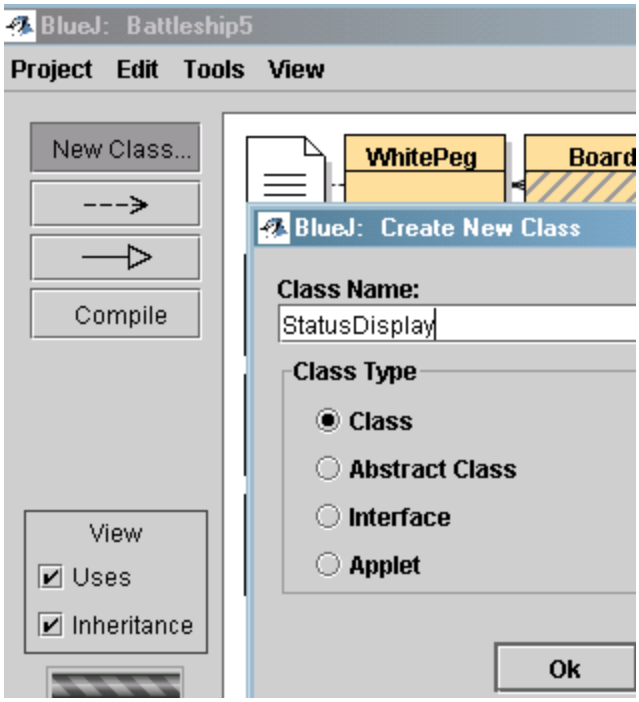


Figura 1 – Creiando a classe StatusDisplay.

Vamos procurar listar os dados que serão necessários. Nossa nova classe terá que saber sobre os navios. Também deverá saber sobre a posição e tamanho da janela de estado, para que possa desenhá-la na tela. Finalmente, deverá usar uma cor diferente para exibir o estado de cada um dos jogadores. De fato, teremos, no programa, dois objetos da classe StatusDisplay – uma para o jogador humano e outro para o computador.
Vamos agora pensar nos métodos dessa classe. É claro que vamos ter um método paint, para que a janela possa ser desenhada em um contexto gráfico (um objeto da classe Graphics). Também devemos ter um constructor, que conterá o código para inicialização dos dados listados acima. Com base nessas idéias, temos o seguinte esqueleto para a nossa nova classe:

```
/**
* Exibe o estado dos navios de um jogador.
*
* @author (seu nome aqui)
* @version v0.1
*/
import java.awt.*;

public class StatusDisplay {
    private int x;
    private int y;
     private Ship[] shipList;
     private Color playerColor;

    /* Cria uma nova janela de estado, com dados
     * correspondentes à situação de início do jogo.
     */
    public StatusDisplay(int x, int y,
                         Ship[] ships, Color col){
      this.x = x;
```

```
        this.y = y;
        this.shipList = ships;
        this.playerColor = col;
    }

    /** Desenha a janela de estado do jogo. */
    public void paint(Graphics gfx) {
    }
}
```

O estado dos navios

Esse é um bom começo... Como esperamos que um objeto dessa classe deva se comportar? Quando criarmos um StatusDisplay, devemos passar como argumento, entre outras coisas, a lista dos navios da esquadra de um jogador. Em cada jogada, devemos solicitar a um objeto da classe StatusDisplay que pinte sua janela na tela. Para isso, ele vai verificar sua lista de navios, calcular os danos a cada navio e exibir as mensagens apropriadas.
Boa idéia – exceto que a nossa classe Ship não mantém nenhuma informação sobre dados aos navios. Para corrigir isso, vamos adicionar os seguintes dados e métodos à classe Ship:

```
// Novo dado
int hitsTaken;              // número de tiros pelos quais foi atingido

// Novo método
public void takeHit() {     // incrementa número de tiros que atingiram o navio
    this.hitsTaken++;
}
```

Além disso, naturalmente precisamos inicializar a variável hitsTaken com o valor 0, no construtor da classe Ship:

```
public Ship(int shipType, int dir, int row,
            int col, int length){
    this.type = shipType;
    this.orientation = dir;
    this.row = row;
    this.column = col;
    this.size = length;
    this.hitsTaken = 0;
}
```

Já é um conjunto substancial de modificações. Se você quiser, compare a sua classe Ship com a do código 20 da seção de códigos, incluída no final deste tutorial.

A integridade de um navio

Agora que nossos navios já incluem informação sobre danos, podemos escrever o código do método paint da janela de estado do jogo, representada pela classe StatusDisplay. Esse código envolve os seguintes passos:

- Repetir, para cada navio
- Computar o percentual de "danos" ao navio
- Exibir uma mensagem, na posição adequada, informando o nome e o percentual de danos do navio
- Passar par o próximo navio

Aqui vai a primeira tentativa:

```
/** Desenha a janela de estado do jogo. */
public void paint(Graphics gfx) {
   int shipIndex = 0;
    while (shipIndex < this.shipList.length) {
       /* Calcula o percentual de danos ao navio corrente. */

       /* Exibe a mensagem de danos ao navio. */

       // Passa para o próximo navio.
       shipIndex++;
  }
}
```

O código acima destaca o loop principal. A única particularidade é o uso da expressão this.shipList.length, que representa o número de navios existentes no array shipList. Como dissemos anteriormente, todo objeto da classe Array possui uma variável de instância – length – que armazena o número de elementos do array.

Agora basta escrever o código especificado pelos comentários destacados acima, em azul. Vamos começar pelo cálculo do percentual de danos a um navio. Olhando para as variáveis da classe Ship, vemos que cada navio tem um 'tamanho' – isto é, o número de células que ele ocupa no tabuleiro. Se um navio tem tamanho 2, ele deverá levar dois tiros, para ser destruído, e assim por diante. Isso significa que poderíamos expressar o dano a um navio pela seguinte razão

número de tiros que atingiram o navio / tamanho do navio

Por exemplo, para um navio e guerra (cujo tamanho é 4) que levou 2 tiros, essa razão seria 2 / 4. Para um porta-aviões (tamanho 5) que levou um tiro, obteríamos 1 / 5. OK, mas seria melhor converter esses valores em percentuais – dessa maneira, o dano a qualquer tipo de navio seria expresso da mesma forma. Felizmente, isso é fácil: basta multiplicar essa razão por 100: percentual de danos = 100 * número de tiros / tamanho. Vamos aplicar isso ao nosso código:

```
/** Desenha a janela de estado do jogo. */
public void paint(Graphics gfx) {
   int shipIndex = 0;
    while (shipIndex < this.shipList.length) {
       /* Calcula o percentual de danos ao navio corrente. */
       Ship ship = this.shipList[shipIndex];
       int damage = 100 * ship.getHits() / ship.getSize();

       /* Exibe mensagem de danos ao navio. */

       // Passa para o próximo navio.
       shipIndex++;
```

```
    }
}
```

Note que adicionamos um novo método à classe Ship:

```
/** Retorna número de tiros que atingiram o navio. */
public int getHits() {
    return this.hitsTaken;
}
```

Finalmente, devemos exibir uma mensagem informando os danos ao navio. Como estamos lidando com uma interface gráfica, isso é feito por meio do método drawString. Vamos exibir as mensagens no seguinte formato:

BATTLESHIP: 0%

CARRIER: 40%

etc.
Isso poderia ser feito usando comandos if...else:

```
if (ship.getType() == Ship.TYPE_BATTLESHIP) {
    /* Imprime 'NAVIO DE GUERRA:' + damage */
}
else if (ship.getType() == Ship.TYPE_AIRCRAFT_CARRIER) {
    /* Imprime 'PORTA-AVIÕES:' + damage */
}
etc...
```

Mas isso seria deselegante e, no caso de existirem muitos tipos de navio, também muito trabalhoso. Aqui vai uma maneira mais concisa e elegante. Imagine um array de Strings como o seguinte:

```
String[] shipNames = {
    "PORTA-AVIÕES",
    "NAVIO DE GUERRA",
    "CRUZADOR",
    "SUMBARINO",
    "BOTE"
};
```

Note que o elemento de índice 0 é o porta-aviões e que a variável estática TYPE_CARRIER (tipo porta-aviões) é igual a 0. De maneira análoga, o elemento de índice 1 é o navio de guerra e a variável estática TYPE_BATTLESHIP (tipo navio de guerra) tem valor 1. Ou seja, inicializamos o array shipNames cuidadosamente, de maneira que seja sempre verdadeiro que shipNames[TYPE_*] = "*" para qualquer nome de navio *. Por exemplo:

shipNames[TYPE_SUBMARINE] = shipNames[3] = "SUBMARINO".

Esse truque pode ser usado para escrever o código do corpo do nosso loop:

```
/* Usado para resolver referências a nomes de navios no loop da janela de estado. */
private String[] shipNames = {
```

```
    "PORTA-AVIÕES",
    "NAVIO DE GUERRA",
    "CRUIZADOR",
    "SUMBARINO",
    "BOTE"
};

/** Desenha a janela de estado do jogo. */
public void paint(Graphics gfx) {
    int shipIndex = 0;
    while (shipIndex < this.shipList.length) {
        /* Calcula o percentual de danos ao navio corrente. */
        Ship ship = this.shipList[shipIndex];
        if (ship != null) {
            int damage = 100 * ship.getHits() /
                         ship.getSize();

            /* Exibe mensagem de daonos ao navio. */
            String name = shipNames[ship.getType()];
            String dmgLabel = "" + damage;
            String message = name + ": " + dmgLabel + "%";
            int messageX = this.x;
            int messageY = y + 25 * shipIndex;
            gfx.setColor(this.playerColor);
            gfx.drawString(message, messageX, messageY);
        }

        // Passa para o próximo navio.
        shipIndex++;
    }
}
```

Isso pode parecer um pouco complexo, mas, de fato, é muito simples. O único conceito relativamente novo é o de concatenação de strings.Repare nas seguintes linhas de código:

```
String name = shipNames[ship.getType()];
String dmgLabel = "" + damage;
String message = name + ": " + dmgLabel + "%";
```

A primeira simplesmente usa o nosso truque para obter o nome do tipo do navio corrente. A segunda e a terceira linhas usam o operador de concatenação de strings – o que é simplesmente uma maneira técnica de dizer "coloque os strings juntos, um depois do outro". Observe a terceira linha:

```
String message = name + ": " + dmgLabel;
```

Nesse comando name e dmgLabel são strings, assim como os símbolos ": " e "%". O operador + indica que esses strings deve ser concatenados. Portanto, se name contém "SUBMARINO" e dmgLabel contém "33", esse comando se reduziria a message = "SUB-

MARINO" + ": " + "33" + "%", ou seja,

```
message = "SUBMARINO: 33%".
```

Voltemos agora ao comando:

```
String dmgLabel = "" + damage;
```

Aqui existe um pequeno truque. Note que damage não é um string, mas sim uma variável de tipo int. Normalmente, Java não aceitaria atribuir um valor numérico a uma variável de tipo String. Para converter esse valor para um string, usamos o truque de concatenar esse valor ao string vazio "" – nesse caso, o valor numérico é automaticamente convertido para um string (pelo compilador) antes da operação de concatenação. Em outras palavras, o comando dmgLabel = "" + 33 se reduz a dmgLabel = "" + "33", ou, simplesmente, dmgLabel = "33".

O restante do loop é muito fácil de entender. Note que adicionamos 25 * shipIndex à coordenada y da janela de estado, sempre que imprimimos a informação sobre danos a um navio. Isso faz com que se mova para a próxima linha na janela de estado – de maneira semelhante àquela pela qual movemos para a próxima linha horizontal, ao desenharmos a grade do tabuleiro.
Você deve estar intrigado pelo seguinte comando if...else:

```
if (ship != null) {
    /* Exibe informação de danos. */
}
```

Isso é necessário para evitar que se tente exibir informação sobre um navio inexistente. Lembre-se que, antes de o usuário posicionar seus navios, do array que representa a sua esquadra tem todas as entradas automaticamente inicializadas com o valor 'null'. Tentar exibir informação de estado para um navio inexistente causaria um erro durante a execução do programa – mais precisamente, causaria uma exceção.

Quase pronto...

Estamos quase prontos para adicionar nossa janela de estado à interface do jogo. Falta apenas criar dois objetos da classe StatusDisplay – cada um para exibir informações de um dos jogadores – e usar esses objetos no código, de maneira adequada. Como a classe Board é responsável pela interface do jogo, é natural que ela também gerencie os objetos da classe StatusDisplay. Vamos, portanto, adicionar a essa classe uma variável para referenciar a janela de estado do jogo:

```
// variáveis de instância.
private RedPeg[] hitMarkers;              // pinos marcadores de acertos
private WhitePeg[] missMarkers;           // pinos marcadores de erros
private Ship[] fleet;                     // esquadra de navios
private int[][] gridCells;                // grade do tabuleiro
private int firstClickRow;                // linha da posição inicial do navio corrente
private int firstClickCol;                // coluna da posição inicial do navio corrente
private int currentUserShip;              // tipo do navio corrente
private int[] shipType = {Ship.TYPE_AIRCRAFT_CARRIER,     // tipos de navio
                  Ship.TYPE_BATTLESHIP,
```

```
                    Ship.TYPE_CRUISER,
                    Ship.TYPE_SUBMARINE,
                    Ship.TYPE_PT_BOAT};
private int[] shipLength = {5, 4, 3, 3, 2};      // tamanhos de navios
private int[] endBoxRow = {-1, -1, -1, -1};     // linha da extremidade final do navio
private int[] endBoxCol = {-1, -1, -1, -1};     // coluna da extremidade final do navio
private StatusDisplay status;                   // janela de estado do jogo
```

Em seguida, vamos criar essa janela, no construtor da classe Board. Note que devemos adicionar alguns argumentos ao construtor, para que tudo funcione como desjeamos: public Board(int statusX, int statusY, Color statusColor)

```
{
    // Inicializa os arrays usados para armazenar informação sobre os navios.
    this.fleet = new Ship[SHIPS_PER_FLEET];
    this.gridCells = new int[ROW_COUNT][COLUMN_COUNT];

   // Preenche as células da grade com vazio.
   int i = 0;
   while (i < ROW_COUNT) {
      int j = 0;
       while (j < COLUMN_COUNT) {
           this.gridCells[i][j] = Ship.TYPE_NONE;
         j++;
      }
      i++;

   }
   // Inicializa linha e coluna do primeiro clique.
   this.firstClickRow = -1;
   this.firstClickCol = -1;

   // Inicializa tipo do navio do usuário.
   this.currentUserShip = 0;

   // Cria gerenciador da janela de estado.
   this.status = new StatusDisplay(statusX,
                          statusY,
                          fleet,
                           statusColor);
}
```

Precisamos também prover uma maneira pela qual a classe Board possa desenhar a janela de estado na tela:

```
public void drawStatus(Graphics gfx) {
   this.status.paint(gfx);
}
```

Além disso, a classe BattleshipApp deve defines as coordenadas da posição onde a janela de estado deve ser exibida:

```
private static final int MSG_WINDOW_DX = GAME_WIDTH;
private static final int MSG_WINDOW_DY = 50;
private static final int MSG_WINDOW_X = 0;
```

```
private static final int MSG_WINDOW_Y = GAME_HEIGHT -
                            MSG_WINDOW_DY;
private static final int PLAYER_STATUS_X = RED_GRID_X;
private static final int STATUS_Y = 250;
private static final int COMPUTER_STATUS_X = BLUE_GRID_X;
```

Como modificamos a assinatura do construtor da classe Board, devemos modificar, de acordo, as chamadas desse construtor, na classe BattleshipApp:

```
/**
 * Construtor de objetos da classe BattleshipApp
 */
public BattleshipApp()
{
    this.setSize(GAME_WIDTH, GAME_HEIGHT);
    this.setVisible(true);
    this.addMouseListener(this);
    this.gameState = GAME_STATE_INTRO;

    // Inicializa os tabuleiros do jogo.
    this.redBoard = new Board(PLAYER_STATUS_X,
                      STATUS_Y,
                      Color.red);
    this.blueBoard = new Board(COMPUTER_STATUS_X,
                       STATUS_Y,
                       Color.blue);
}
```

E ainda, precisamos dizer à classe Board para desenhar a janela de estado na tela:

```
/**

 * Desenha a janela do jogo.
 */
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
       this.drawTitleScreen();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       this.drawGrids();
       this.drawStatus(gfx);
       this.drawMessage(gfx);
   }
    else if (this.gameState == GAME_STATE_PLAYING) {
       this.drawGrids();
       this.drawStatus(gfx);
       this.drawMessage(gfx);
   }
}

/** Desenha a janela de estado do jogo. */
public void drawStatus(Graphics gfx) {
    this.redBoard.drawStatus(gfx);
    this.blueBoard.drawStatus(gfx);
}
```

Fizemos um grande número de pequenas modificações, em várias classes. É uma boa idéia verificar se tudo está funcionando corretamente. Compile o seu programa. Se você quiser, poderá verificar seu código comparando-o com as nossas versões das classes StatusDisplay (código 21a), Board (código 21b), e BattleshipApp (código 21c), disponíveis na eção de códigos, incluída no final deste tutorial.

Depois que tudo estiver funcionando, podemos começar a escrever o código para alternar entre as jogadas do computador e do usuário.

Alternando as jogadas do usuário e do computador

Estrutura de controle da alternância entre jogadores

Nesta seção, vamos escrever o código para gerenciar a alternância ente as jogadas do usuário e do computador. Não vamos ainda implementar as jogadas de cada um dos jogadores, mas apenas estabelecer a estrutura que controla de quem é a vez de jogar. E decide quem ganhou o jogo. Vamos começar pensando como isso deve funcionar:

1. Vamos supor que o usuário é que faz a primeira jogada.

2. Quando ele clicar sobre uma célula da grade azul, isso é interpretado como uma jogada.

3. Imediatamente depois da jogada do usuário, deve-se verificar se ele venceu.

4. Se o usuário venceu, deve ser exibida uma mensagem de vitória e iniciado o estado de fim de jogo - GAME_OVER.

5. Se o usuário não venceu, então passa a ser a vez da jogada do computador.

6. Imediatamente depois da jogada do computador, deve-se verificar se ele venceu.

7. Se o computador venceu, deve ser exibida uma mensagem de vitória e iniciado o estado de fim de jogo - GAME_OVER.

Vamos então ver como isso pode ser implementado.

One Blind Mouse

Comecemos pelo método mouseReleased, da classe BattleshipApp. Precisamos adicionar a esse método uma nova seção do comando if...else para tratar o caso do estado de GAME_STATE_PLAYING, no qual os jogadores alternam suas jogadas. Nesta seção, vamos implementar os passos 2 a 7 descritos anteriormente:

```
else if (this.gameState == GAME_STATE_PLAYING) {
   /* Obtém a posição onde o mouse foi clicado. */
   /* Verifica se o usuário clicou em uma célula vazia do tabuleiro do inimigo */
   /* Se SIM...*/
       /* Se o usuário acertou um navio, incremente o contador de danos ao navio. */
       /* Verifica se o usuário ganhou o jogo */
      /* Se não... */
          /* É a vez do computador jogar. */
            /* Se o computador acertou um navio, incremente o contador de danos ao
```

```
navio. */
          /* Verifica se o computador venceu o jogo */
         /* Se SIM... */
             /* Imprime a mensagem: "Você perdeu." */
             /* Entra no estado de fim de jogo. */
      /* SENÃO... */
          /* Imprime a mensagem: "Você ganhou!" */
          /* Entra no estado de fim de jogo. */
   /* SENÃO...*/
       /* Imprime a mensagem: "Por favor, clique sobre uma célula do tabuleiro azul."
*/
}
```

Vamos começar a substituir esses comentários por código de verdade.

Primeiro, precisamos calcular a linha e coluna da posição onde o usuário clicou o mouse. Você deve lembrar-se dos métodos 'convertYtoRow' e 'convertXtoCol' que escrevemos anteriormente. Eles convertem as coordenadas da posição do mouse em linha e coluna da grade do tabuleiro vermelho do jogador. Queremos o mesmo para o tabuleiro azul. É fácil copiar esses métodos e convertê-los para funcionar para o tabuleiro azul:

```
/**
 *  Converte a coordenada y da posição do mouse em uma linha do tabuleiro azul.
 */
private int convertYtoEnemyRow(int pixelY) {
    Container clientArea = this.getContentPane();
    int borderSize = (this.getWidth() - clientArea.getWidth()) /
                2;
    int titleSize = (this.getHeight() - clientArea.getHeight()) -
                borderSize;
    int row = (pixelY - titleSize - BLUE_GRID_Y) /
            Board.ROW_PIXEL_HEIGHT;
    return row;

}
/**
 *  Converte a coordenada x da posição do mouse em uma coluna do tabuleiro azul.
 */
private int convertXtoEnemyCol(int pixelX) {
    Container clientArea = this.getContentPane();
    int borderSize = (this.getWidth() - clientArea.getWidth()) /
                2;
    int column = (pixelX - borderSize - BLUE_GRID_X) /
              Board.COLUMN_PIXEL_WIDTH;
    return column;
}
```

Em seguida, vamos introduzor alguns métodos novos, para nos ajudar a reduzir o código necessário no corpo do método mouseReleased:

```
/** Verifica o ponto no qual o usuário clicou o mouse.
  * Se esse ponto está sobre uma célula vazia do tabuleiro, retorna 'true'.
  */
```

```
private boolean isPlayerTargetValid(int mouseX, int mouseY) {
    boolean bIsValid = false;

    int row = convertYtoEnemyRow(mouseY);
    int col = convertXtoEnemyCol(mouseX);

    // O ponto está dentro do tabuleiro?
    if (row >= 0 && row < Board.ROW_COUNT &&
        col >= 0 && col < Board.COLUMN_COUNT) {
        // O ponto está dentro de uma célula vazia?
         bIsValid = this.blueBoard.isUnpegged(row, col);
   }

    return bIsValid;
}
```

Já vimos código semelhante anteriormente, portanto, vamos apenas discutir brevemente como ele funciona. Começamos supondo que o usuário não clicou sobre uma célula vazia do tabuleiro, isto é, bIsValid = false. Em seguida, obtemos a linha e coluna correspondentes ao ponto onde ocorreu o clique do mouse. Então verificamos se esse ponto está dentro do tabuleiro, isto é, se a linha está entre 0 e o número de linhas e se a coluna está entre 0 e o número de colunas. Se a linha e coluna forem válidas, então chamamos um novo método da classe Board – 'isUnpegged' – que verifica se existe um pino na célula identificada pela linha e coluna. Se a célula não contém um pino, então a jogada é válida e armazenamos o resultado. Caso contrário, bIsValid permanence como o valor false.
Note que usamos as variáveis ROW_COUNT ed COLUMN_COUNT da classe Board – o que significa wue elas devem ser declaradas como public:

```
// variáveis de classe.
public static final int ROW_COUNT    = 10;
public static final int COLUMN_COUNT = 10;
```

Além disso, é claro que vamos precisar escrever o código do novo método isUnpegged, na classe Board:

```
/** Verifica o ponto no qual o usuário clicou o mouse.
  * Se esse ponto está sobre uma célula vazia do tabuleiro, retorna 'true'.
  */
public boolean isUnpegged(int row, int col) {
    boolean bIsUnpegged = true;

    int squareContents = this.gridCells[row][col];
    if (squareContents == TYPE_PEG_WHITE ||
        squareContents == TYPE_PEG_RED) {
        bIsUnpegged = false;
   }
    return bIsUnpegged;
}
```

Esse código deve ser fácil de entender – lembre-se que o operador || significa "ou" (assim como '&&' significa "e"). Portanto, o comando:

```
   if (squareContents == TYPE_PEG_WHITE ||
```

```
        squareContents == TYPE_PEG_RED) {
        bIsUnpegged = false;
    }
```

pode ser lido do seguinte modo: "Se a célula contém um pino branco OU contém um pino vermelho, então atribua o valor false a bIsUnpegged." Fácil!
Note que verificamos o conteúdo do array gridCells para ver se a célula contém um pino. Isso implica que teremos que preencher esse array com valores que representam o número de pinos que o usuário colocou no tabuleiro do computador, correspondentes a tiros do usuário sobre os navios do inimigo. Vamos então adicionar esses valores à nossa lista de constantes – note que esses valores deverão ser diferentes dos já usados para indicar a presença de um navio, de qualquer dos tipos. Portanto, vamos adicionar o seguinte, no início da classe Board:

```
// variáveis de classe.
public static final int ROW_COUNT             = 10;
public static final int COLUMN_COUNT          = 10;
public static final int ROW_PIXEL_HEIGHT      = 20;
public static final int COLUMN_PIXEL_WIDTH    = 20;
private static final int SHIPS_PER_FLEET      = 5;
private static final int TYPE_PEG_WHITE       = -2;  // indica pino branco
private static final int TYPE_PEG_RED         = -3;  // indica pino vermelho
```

Agora, vamos ver como esses novos métodos impactam o código de mouseReleased:

```
else if (this.gameState == GAME_STATE_PLAYING) {
    // Obtém a linha e coluna correspondents à posição onde foi clicado o mouse.
    int row = convertYtoEnemyRow(event.getY());
    int col = convertXtoEnemyCol(event.getX());

    // Verifica se o usuário clicou sobre uma célula vazia do tabuleiro do inimigo
    if (this.isPlayerTargetValid(event.getX(), event.getY())) {
       /* Se o usuário acertou um navio, incremente o contador de danos ao navio. */
       /* Verifica se o usuário ganhou o jogo */
      /* Se não... */
          /* É a vez do computador jogar. */
            /* Se o computador acertou um navio, incremente o contador de danos ao
navio. */
          /* Verifica se o computador venceu o jogo */
         /* Se SIM... */
             /* Imprime a mensagem: "Você perdeu." */
             /* Entra no estado de fim de jogo. */
      /* SENÃO... */
          /* Imprime a mensagem: "Você ganhou!" */
          /* Entra no estado de fim de jogo. */
   /* SENÃO...*/
       /* Imprime a mensagem: "Por favor, clique sobre uma célula do tabuleiro azul."
*/
   }
   else {
       this.setMessage("Por favor, clique sobre uma célula do tabuleiro azul.", true);
   }
}
```

Nesse ponto, é bom compilar e executar o programa, para verificar se tudo funciona corretamente. Teste se você recebe uma mensagem de instrução, quando clicar for a do tabuleiro azul. Se você tiver problemas, verifique o seu código, comparando-o com as versões das classes BattleshipApp (código 22a) e Board (código 22b) que apresentamos na seção de códigos incluída no final deste tutorial.

Agora vamos passar para o código que implementa a jogada do usuário.

Jogada do usuário

Plano da jogada

Para completar o código que implementa uma jogada do usuário, precisamos adicionar os seguintes passos:

- Testar se o tiro do usuário atingiu um navio inimigo.
- Em caso afirmativo, incrementar o contador de danos do navio atingido e preencher a célula com um pino vermelho.
- Caso contrário, preencher a célula com um pino branco.
- Testar se o jogador ganhou o jogo.

Vamos tratar de implementar cada um desses passos, na ordem acima.

B-5...Acertou!

Podemos facilmente adicionar o código que testa se o tiro do jogador acertou um navio. Sabemos que o array gridCells, da classe Board, mantém informação sobre o conteúdo de cada célula do tabuleiro. Células vazias têm o valor -1. Células com pinos vermelhos ou broncos têm os valores -3 e -2, respectivamente. Finalmente, células contendo navios têm valores de 0 a 4, dependendo do tipo de navio posicionado na célula. Portanto, para testar se o tiro atingiu um navio, basta verificar se o valor da célula correspondente à posição do tiro, no array gridCells, é maior ou igual a 0.
Portanto, vamos adicionar o seguinte método, na classe Board:

```
/** Verifica se a célula contém um navio. */
public boolean didHitShip(int row, int col) {
    return this.gridCells[row][col] >= 0;
}
```

Note que o símbolo >= representa o operador de comparação “menor ou igual”.
Agora, precisamos aplicar ao navio correto, o dano provocado pelo tiro. Isso pode parecer complicado, mas, de fato, é muito fácil. Lembre-se que organizamos o nosso array que representa a esquadra de navios de maneira que

fleet[TYPE_BATTLESHIP] é o navio de guerra,
fleet[TYPE_CARRIER] é o porta-aviões, etc.

Além disso, o array gridCells contém o valor TYPE_BATTLESHIP em toda célula que contém um navio de Guerra, e contém o valor TYPE_CARRIER em toda célula que contém um

porta-aviões, etc. Portanto, se obtivermos o valor armazenado em uma célula do array gridCells, podemos usar esse valor como índice, para obter o navio correto, do seguinte modo:

```
int shipType = gridCells[row][col];
Ship hitShip = fleet[shipType];
```

Tendo isso em mente, vamos escrever o método que aplica o dano ao navio atingido por um tiro do usuário.

```
/** Aplica o dano ao navio atingido por um tiro dado na célula (row, col). */
public int applyDamage(int row, int col) {
    int shipType = this.gridCells[row][col];
    Ship hitShip = this.fleet[shipType];
    hitShip.takeHit();

    return shipType;
}
```

Note que o método retorna o tipo do navio atingido. Podemos usar esse valor para fazer o computador emitir uma mensagem para o usuário, tal como "Você acertou meu bote!", ou para realizar outras tarefas relacionadas. Vamos deixar para que você escreva o código para exibir essa mensagem.
Agora, precisamos preencher agrade com um pino vermelho ou branco, conforme o tiro tenha acertado um navio ou não. Isso é fácil:

```
/** Preenche a célula com um pino. */
public void putPegInSquare(int row, int col, boolean bRed) {
   if (bRed) {
       this.gridCells[row][col] = TYPE_PEG_RED;
   }
   else {
       this.gridCells[row][col] = TYPE_PEG_WHITE;
   }
}
```

Precisamos também adicionar o código para desenhar um pino no tabuleiro. Para isso, vamos adaptar o código para desenhar um navio - drawShips:

```
/**
 *  Desenha um navio na grade do tabuleiro.
 */
public void drawShips(Graphics gfx, Color shipColor,
                      int startX, int startY) {

   /* REMOVA ESSAS LINHAS.
   // Especifica a cor do navio.
    gfx.setColor(shipColor);
   */

   int row = 0;    // Começa pela primeira linha.
   do {

            int col = 0;  // Começa pela primeira coluna.
```

```
        do {

        // A célula está vazia?
         if (this.gridCells[row][col] != Ship.TYPE_NONE) {
            // A célula contém parte de um navio ou um pino.
             if (this.gridCells[row][col] == TYPE_PEG_WHITE) {
                gfx.setColor(Color.white);
           }
             else if (this.gridCells[row][col] == TYPE_PEG_RED) {
                gfx.setColor(Color.red);
           }
             else gfx.setColor(shipColor);

            // Calcula a posição inicial da célula.
            int x = startX + col * COLUMN_PIXEL_WIDTH;
            int y = startY + row * ROW_PIXEL_HEIGHT;

             // Desenha um quadrado pouco menor que a célula.
            gfx.fillRect(x + 2, y + 2,
                      COLUMN_PIXEL_WIDTH - 3,
                      ROW_PIXEL_HEIGHT - 3);
       }

         col++;  // Passa para a próxima coluna.
      } while (col < COLUMN_COUNT);

      row++;      // Passa para a próxima linha.
    } while (row < ROW_COUNT);
}
```

Agora só falta testar se o usuário venceu o jogo. Isso pode ser feito de várias maneiras. Uma opção seria verificar todo o array gridCells para ver se ainda existe alguma célula que contenha parte de um navio. Outra opção seria verificar todo o array de navios – fleet – para verificar se algum navio ainda não foi completamente destruído. Vamos adotar essa segunda opção.

```
/** Verifica se existe algum navio que ainda não foi destruído. */
public boolean isAnyShipLeft() {
    boolean bShipIsLeft = false;

   int i=0;
   while (i < fleet.length) {
      if (fleet[i] != null &&
          fleet[i].getHits() < fleet[i].getSize()) {

          bShipIsLeft = true;
          break;
      }
      i++;
   }

     return bShipIsLeft;
}
```

Lembre-se que o comando break provoca o término do loop que está sendo executado.

Nesse caso, isso significa que quando encontrarmos um navio que ainda não foi destruído, o loop será interrompido. Note que, nesse caso, o valor de bShipIsLeft será true.
Acredite ou não, já concluímos o código que implementa a jogada do usuário. Vamos então incluí-lo no corpo do método mouseReleased, da classe BattleshipApp:

```
else if (this.gameState == GAME_STATE_PLAYING) {
    // Obtém a linha e coluna correspondents à posição do clique do mouse.
    int row = convertYtoEnemyRow(event.getY());
    int col = convertXtoEnemyCol(event.getX());

    // Verifica se o usuário clicou sobre uma célula vazia do tabuleiro do inimigo
    if (this.isPlayerTargetValid(event.getX(), event.getY())) {
       /* Se o usuário acertou um navio, aplique o dano ao navio. */
       if (this.blueBoard.didHitShip(row, col)) {
          this.blueBoard.applyDamage(row, col);
          this.blueBoard.putPegInSquare(row, col, true);
       } this.blueBoard.putPegIsSquare(row, col, false);

       /* Verifica se I jogador venceu o jogo */
       if (!this.blueBoard.isAnyShipLeft()) {
          this.setMessage("Você venceu!", true);
          this.gameState = GAME_STATE_GAMEOVER;
      }
      else {
          /* É a vez do computador jogar. */
            /* Se o computador acertou um navio, incremente o contador de danos ao
navio. */
          /* Verifica se o computador venceu o jogo */
         /* Se SIM... */
             /* Imprime a mensagem: "Você perdeu." */
             /* Entra no estado de fim de jogo. */
      }
   }
   else {
       this.setMessage("Por favor, clique sobre uma célula do tabuleiro azul.", true);
   }

    // Atualiza o tabuleiro, de maneira que os pinos sejam desenhados.
    this.repaint();
}
```

Finalmente, precisamos tratar o fato de que agora podemos entrar no estado final do jogo:

```
/**
 * Desenha a janela do jogo.
 */
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
       this.drawTitleScreen();
   }
    else if (this.gameState == GAME_STATE_SETUP) {
       this.drawGrids();
       this.drawStatus(gfx);
       this.drawMessage(gfx);
```

```
    }
     else if (this.gameState == GAME_STATE_PLAYING) {
        this.drawGrids();
        this.drawStatus(gfx);
        this.drawMessage(gfx);
    }
     else if (this.gameState == GAME_STATE_GAMEOVER) {
        this.drawGrids();
        this.drawStatus(gfx);
        this.drawMessage(gfx);
    }
}
```

Note que o código deste método pode ser simplificado para o seguinte:

```
/**
 * Desenha a janela do jogo.
 */
public void paint(Graphics gfx) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.drawTitleScreen();
    }
    else {
        this.drawGrids();
        this.drawStatus(gfx);
        this.drawMessage(gfx);
    }
}
```

Estamos prontos para compiler e executar o programa. Se você tiver algum problema, verifique seu programa, comparando-o com as nossas versões das classes Board (código 23b) e BbattleshipApp.(código 23ª), disponíveis na seção de códigos incluída no final deste tutorial

Por enquanto, nosso jogo possibilita jogadas de um único jogador. Ok, isso não tem a menor graça! Além disso, o jogador pode ver a posição dos navios no tabuleiro do computador – isso também não tem a menor graça! Bem, mas aproveite para abusar do computador e vencer bastante. Quando você se cansar, vamos prepará-lo para jogar contra você – vamos ao código para a jogada do computador.

Jogada do computador

Cérebro eletrônico

Estamos perto de fazer nosso jogo de Batalha Naval funcionar! Só falta fazer o computador aprender a jogar. Para isso, temos que implementar os seguintes passos:

- Selecionar uma célula sobre a qual computador vai atirar.
- Se o tiro atingiu algum navio do usuário, aplicar o dano ao navio.
- Preencher a célula com um pino branco ou vermelho.

- Verificar se o computador ganhou o jogo.

Hmm...isso parece bastante familiar – de fato, é quase idêntico ao conjunto de passos da jogada do usuário. Isso significa que já escrevemos a maior parte do código requerido. Portanto, esta seção tem um objetivo principal: escrever o código para o computador escolher a posição de tiro.

Se você já brincou de Batalha Naval, sabe que vencer o jogo depende, em grande parte, de sorte. Geralmente, escolhemos aleatoriamente a posição onde atirar, até que atingimos algum navio. Para tornar as coisas mais simples, a primeira lógica que vamos implementar para as jogadas do computador consiste simplesmente em sempre atirar aleatoriamente. Ok, essa é uma estratégia muito pobre, mas irá nos ajudar a entender a estrutura básica de uma jogada do computador. Uma vez que tenhamos implementado essa estratégia, vamos procurar adicionar uma estratégia mais inteligente, no módulo seguinte.

OK – vamos começar a escrever o código para nossa jogada básica do computador!

Já vi isso antes ...

Antes de começar, precisamos criar um objeto para representar o jogador inimigo – o computador. Isso deve ser feito na classe BattleshipApp, no mesmo local onde criamos os tabuleiros:

```
/**
 * Construtor de objetos da classe BattleshipApp
 */
public BattleshipApp()
{
    this.setSize(GAME_WIDTH, GAME_HEIGHT);
    this.setVisible(true);
    this.addMouseListener(this);
    this.gameState = GAME_STATE_INTRO;

    // Inicializa os tabuleiros do jogo.
    this.redBoard = new Board(PLAYER_STATUS_X,
                    STATUS_Y,
                    Color.red);
    this.blueBoard = new Board(COMPUTER_STATUS_X,
                     STATUS_Y,
                     Color.blue);

    // Inicializa o jogador inimigo – o computador.
    this.enemyPlayer = new EnemyPlayer();
}
```

Agora vamos voltar ao método mouseReleased para ver como devemos estruturar uma jogada do computador.

```
else if (this.gameState == GAME_STATE_PLAYING) {
    // Obtém a linha e coluna correspondents à posição do clique do mouse.
    int row = convertYtoEnemyRow(event.getY());
    int col = convertXtoEnemyCol(event.getX());

    // Verifica se o usuário clicou sobre uma célula vazia do tabuleiro do inimigo
```

```
        if (this.isPlayerTargetValid(event.getX(), event.getY())) {
           /* Se o usuário acertou um navio, aplique o dano ao navio. */
           if (this.blueBoard.didHitShip(row, col)) {
               this.blueBoard.applyDamage(row, col);
               this.blueBoard.putPegInSquare(row, col, true);
           } this.blueBoard.putPegIsSquare(row, col, false);

           /* Verifica se I jogador venceu o jogo */
           if (!this.blueBoard.isAnyShipLeft()) {
              this.setMessage("Você venceu!", true);
              this.gameState = GAME_STATE_GAMEOVER;
          }
          else {
               // Determina a posição do tiro do computador.
               this.enemyPlayer.selectTarget();
               row = this.enemyPlayer.getTargetRow();
               col = this.enemyPlayer.getTargetColumn();

               // Verifica se o tiro atingiu um navio do usuário
               if (this.redBoard.didHitShip(row, col)) {
                   this.redBoard.applyDamage(row, col);
                   this.redBoard.putPegInSquare(row, col, true);
              }
               else this.redBoard.putPegInSquare(row, col, false);

               /* Verifica se o computador venceu o jogo */
               if (!this.redBoard.isAnyShipLeft()) {
                  this.setMessage("You lose...", true);
                  this.gameState = GAME_STATE_GAMEOVER;
              }
          }
       }
       else {
            this.setMessage("Por favor, clique sobre uma célula do tabuleiro azul.", true);
       }

       // Atualiza o tabuleiro, de maneira que os pinos sejam desenhados.
       this.repaint();
}
```

Note que as linhas de código destacada em laranja, espelham exatamente as que correspondem à jogada do usuário. Viu, não dissemos que a maior parte do código requerido já tinha sido escrita anteriormente? Portanto, apenas temos que nos preocupar com a parte do código destacada em azul:

```
this.enemyPlayer.selectTarget();
int row = this.enemyPlayer.getTargetRow();
int col = this.enemyPlayer.getTargetColumn();
```

Para que isso funcione, vamos precisar adicionar alguns novos métodos e variáveis de instância à classe do jogador inimigo – EnemyPlayer:

```
// Inclui a blilioteca que contém a classe Random.
import java.util.*;
```

```
public class EnemyPlayer
{
   // variáveis de instância.
   private int targetRow;
   private int targetColumn;

   // Métodos.
   /**
    * Construtor de objetos da classe EnemyPlayer
    */
   public EnemyPlayer()
   {
   }

   /* Seleciona aleatoriamente a linha e coluna alvo do tiro. */
   public void selectTarget() {
       long seed = System.currentTimeMillis();
       Random randomizer = new Random(seed);

       this.targetRow = randomizer.nextInt(Board.ROW_COUNT);
      this.targetColumn =
           randomizer.nextInt(Board.COLUMN_COUNT);
   }

   /* Retorna a linha alvo do tiro */
   public int getTargetRow() {
      return this.targetRow;
   }

   /* Retorna a coluna alvo do tiro */
   public int getTargetColumn() {
      return this.targetColumn;
   }
}
```

Observe o código do método selectTarget. Ele usa a classe Random, nossa conhecida de módulos anteriores. Lembre-se que um objeto dessa classe é um gerador de números pseudo-aleatórios – isso significa que ele produz uma série infinita de números que parecem aleatórios, mas de fato não são. Por exemplo, os dez primeiros números da série poderiam ser:

5, -15, 331660, 12372, -5, -5, 0, -623456, 99, 11255

Embora esses números pareçam aleatórios, se o programa for executado várias vezes, a mesma série de números aparecerá, a cada vez. Para evitar isso, utiliza-se uma semente como geradora da série de números, quando essa é criada. Isso é apenas uma maneira de dizer que se começa a obter números da série a partir de um ponto diferente, a cada vez. Em outras palavras, ao invés de sempre começar pelo início da série (5, -15, etc), começa-se em algum outro ponto, o que efetivamente resulta em uma nova série de números aleatórios. É claro que, se o mesmo valor de semente for usado, a mesma série de números será obtida. Portanto, ao invés de atribuir à semente um valor constante, usamos como valor da semente o resultado da chamada ao método currentTimeMillis, isto é, o tempo decorrido desde o início da execução do sistema, em milisegundos. Por isso, obtemos uma série diferente de números, cada vez que o programa é executado.

Uma vez especificada a semente do gerador de números aleatórios, o método nextInt fornece o próximo número da série. Esse valor é usado para se obter um valor compreendido entre 0 e Board.ROW_COUNT – 1, e outro entre 0 e Board.COLUMN_COUNT - 1. Pronto – isso nos dá uma posição alvo de tiro aleatória.

Fácil demais, não é? Inclua esse código na classe EnemyPlayer e, em seguida, compile e execute o seu programa. Brinque um pouco e veja o que acontece...

De volta ao projeto da jogada do computador

Se você jogou um jogo mais longo, provavelmente notou algo – algumas vezes o computador não responde à jogada do usuário, atirando em seguida. Bem, de fato ele atira, mas algumas vezes escolhe uma célula que já contém um pino. Esse é o problema que existe no nosso código atual – não temos como garantir que o computador sempre atira em uma célula válida, isto é, que não contém um pino.

Existem diversas maneiras para contornar esse problema. Uma opção seria escolher aleatoriamente uma célula, verificar se ela contém um pino, examinando o array gridCells, e, em caso afirmativo, escolher aleatoriamente uma nova posição de tiro. Isso funcionaria muito bem, até que o tabuleiro começasse a ficar cheio de pinos. Imagine o que aconteceria no pior caso, em que existisse apenas uma célula do tabuleiro a ser atingida: o computador poderia ter que tentar atirar várias vezes, antes que essa célula fosse escolhida aleatoriamente. Vamos então procurar uma maneira mais elegante para solucionar nosso problema.

Uma solução mais elegante

Alguns problemas acabam por tornar-se bastante simples, quando pensamos sobre eles da maneira adequada. Nosso problema atual não é uma exceção. Para obter uma boa solução, normalmente, temos que pensar em diversas soluções possíveis, para então escolher a mais adequada. Você é um felizardo – nesse caso, apenas precisará ler e entender a solução que descrevemos a seguir.

Imagine que você divide o tabuleiro vermelho em células e coloca todas as células em um saco. Imagine agora que o computador, em cada jogada, retira uma única célula desse saco, para servir como alvo do tiro. Isso garante que ele nunca irá atirar sobre uma célula onde já atirou anteriormente (isto é, que contenha um pino). Simples, não é?

Vamos então implementar essa idéia. Vamos precisar de duas novas classes – uma para representar o "saco" e outra para representar uma célula alvo. Felizmente, Java provê algumas classes que implementam um "saco" – vamos usar, em nosso programa, a classe Vector, definida na biblioteca java.util.
A classe que representa uma célula alvo é simples de ser definida:

```
/** Target – representa uma célula alvo de tiro */
class Target {
   public int row;
   public int column;

   /** Construtor. */
   public Target(int row, int col) {
      this.row = row;
```

```
        this.column = col;
    }
}
```

Note que as variáveis row (linha) e column (coluna) são públicas. Normalmente, isso não seria aconselhável – usualmente declaramos variáveis de instância como privadas e declaramos métodos get correspondentes, para que outras classes possa usar essas variáveis. Entretanto, não fazemos isso, nesse caso. A razão é que nenhuma outra classe, além de EnemyPlayer irá usar a classe Target, e, portanto, vamos "embutir" a classe Target na classe EnemyPlayer. Isso é feito do seguinte modo:

```
// Inclua a cláusula de importação da biblioteca que contém a classe Random.
import java.util.*;

public class EnemyPlayer
{
    // Instance Variables.
    private int targetRow;
    private int targetColumn;

    // Métodos.
    /**
     * Construtor de objetos da classe EnemyPlayer
     */
    public EnemyPlayer()
    {
    }

    /* Escolhe aleatoriamente a linha e coluna da célula alvo do tiro. */
    public void selectTarget() {
        long seed = System.currentTimeMillis();
        Random randomizer = new Random(seed);

        this.targetRow = randomizer.nextInt(Board.ROW_COUNT);
        this.targetColumn =
                randomizer.nextInt(Board.COLUMN_COUNT);
    }

    public int getTargetRow() {
        return this.targetRow;
    }

    public int getTargetColumn() {
        return this.targetColumn;
    }

    // Classe Target ==============================================
    private class Target {
        public int row;
        public int column;

        /** Constructor. */
        public Target(int row, int col) {
            this.row = row;
            this.column = col;
```

```
      }
    }
}
```

Note que a classe Target é declarada com o atributo private. Isso garante que apenas a classe EnemyPlayer pode usá-la. Ë por esse motivo que row e column são declaradas como públicas – sabemos exatamente que quem pode ter acesso a essas variáveis é apenas a classe EnemyPlayer.
Agora, vamos introduzir nosso "saco" e enchê-lo com as células alvo:

```
public class EnemyPlayer
{
    // variáveis de instância.
    private int targetRow;
    private int targetColumn;
     private Vector <Target>bag;

    // Métodos.
    /**
     * Construtor de objetos da classe EnemyPlayer
     */
     public EnemyPlayer()
    {
        this.bag = new Vector <Target> ();

       int iRow = 0;
        while (iRow < Board.ROW_COUNT) {

          int iCol = 0;
           while (iCol < Board.COLUMN_COUNT) {
              Target boardPiece = new Target(iRow, iCol);
              this.bag.addElement(boardPiece);
             iCol++;
          }

          iRow++;
       }
    }
    // O restante da classe entra aqui.
}
```

Não existe muito o que explicar aqui. Você pode notar que criamos um objeto da classe Vector usando o operador new. Note que indicamos, nesse comando, o tipo dos elementos do objeto da classe Vector que está sendo criado – os elementos são objetos da classe Target. Em seguida, usamos um loop para encher o "saco" com células, isto é, criamos um objeto da classe Target para cada possível combinação de linha e coluna e, em seguida o colocamos no saco, usando o comando this.bag.addElement(boardPiece). Você poderá ler mais sobre a classe Vector na documentação de Java (veja http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html)

Tudo o que precisamos fazer agora é retirar uma célula do saco,, quando o computador vai dar seu tiro. Para isso usamos o seguinte método:

```
this.bag.elementAt(x);
```

onde 'x' é um número na faixa de 0 ao número de células restantes no saco menos 1. Esse número pode ser obtido assim:

```
int piecesInBag = this.bag.size();
```

Ë, é claro, sabemos como gerar um número aleatório de 0 ao número de células restantes menos 1:

```
int piece = randomizer.nextInt(piecesInBag);
```

Juntando tudo isso, obtemos nosso novo método selectTarget:

```
/* Seleciona aleatoriamente a linha e coluna da célula alvo do tiro. */
public void selectTarget() {
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

   // Pega uma célula no saco.
   int piecesInBag = this.bag.size();
    int piece = randomizer.nextInt(piecesInBag);
    Target target = this.bag.elementAt(piece);

   this.targetRow = target.row;
    this.targetColumn = target.column;

   // Remove a célula do saco.
    this.bag.removeElementAt(piece);
}
```

Isso é tudo.
Finalmente, note que removemos a célula do saco usando o seguinte comando:

```
this.bag.removeElementAt(piece);
```

Portanto, garantimos que o computador nunca irá atirar sobre uma célula na qual já tenha atirado anteriormente.

OK – temos um bocado de código a ser testado. Vamos compilar e executar o programa, e verificar se ele funciona como esperado (um bom teste: clique sobre cada célula, na ordem em que aparecem no tabuleiro; o computador deverá concluir os tiros sobre o seu tabuleiro ao mesmo tempo que você – a não ser, é claro, que ele atinja todos os seus navios antes....).

Se você tiver algum problema para executar o seu programa, confira o código da sua classe EnemyPlayer com o código 24 da seção de códigos incluída no final deste tutorial.

Professor Zorg fala...

Muito bem! Você completou sua primeira versão do jogo de Batalha Naval! É claro que ela não é assim perfeita – mas dá para jogar e tem todos os elementos do jogo de Batalha Naval original. Esse não é um feito a ser desprezado! Note que nunca usamos algumas das classes que planejamos no início: FriendlyPlayer, WhitePeg, e RedPeg. Isso freqüentemente acontece no projeto de um sistema – o projeto do sistema é refinado, passo a passo, e inicialmente não temos uma idéia completa da sua versão final. Não se incomode – é sempre possível (e muito fácil) remover classes não utilizadas.

Por enquanto, divirta-se com o seu jogo. Pense em como você poderia melhorar o "cérebro do computador", para que ele adote uma estratégia de tiros mais inteligente. É isso o que vamos procurar fazer no próximo módulo. Antes, porém, vamos rever o que aprendemos neste módulo.

Conclusão

Uau, você acaba de concluir sua primeira versão completa do jogo de Batalha Naval! Isso envolveu um bocado de trabalho: tratar eventos do mouse, construir uma interface gráfica para o jogo, implementar uma estratégia de jogada para o computador... Nada mal para algumas poucas lições, não é mesmo?

É claro que o nosso jogo tem algumas imperfeições. A tela tende a piscar (em inglês, flicker), e você apenas pode jogar uma vez. Não se preocupe, ainda vamos procurar corrigir alguns desses problemas, nos próximos módulos. Até lá, não se sinta intimidado – você fez um excelente trabalho e deve sentir-se orgulhoso do seu programa!

De fato, só o que você fez neste modulo já foi um bocado: adicionou uma janela de estado, implementou a jogada do jogador humano e escreveu código (não muito inteligente, devemos reconhecer) para a jogada do computador.

No próximo módulo vamos procurar melhorar a estratégia de jogadas do computador. Veremos então se você ainda será capaz de vencê-lo tão facilmente! Portanto, aproveite e ganhe bastante!

Você pode também experimentar brincar com o código do seu programa, experimentando modificá-lo um pouco: Tabuleiros maiores? Mais navios? Número limitado de tiros?. Você pode inventar várias regras novas e tentar implementar diferentes versões do jogo. O que pode ser mais divertido que isso?

Módulo 9

Mais sobre Tratamento de Eventos

Introdução

É claro que, na atual versão do jogo, está muito fácil vencer o computador. A estratégia de jogada implementada não é muito inteligente... aleém disso podemos ver o tabuleiro do inimigo e, portanto, nunca errar um tiro! Vamos tratar de corrigir esses problemas e tornar nosso oponente mais inteligente.

Vamos aproveitar para aprender mais uma nova forma de comando de repetição – o comando for.

E então, vamos lá?

A nova estratégia de jogada do inimigo

Um melhor pulo do gato (ou pulo do rato)...

Nosso programa não está mal... mas a inteligência do nosso oponente – o computador – ainda deixa muito a desejar. Além disso, fica difícil errar um tiro, quando se pode ver o tabuleiro do oponente e, portanto, saber onde seus navios estão posicionados. Vamos então tratar de corrigir esse problema, antes de mais nada. Para isso, basta introduzir a seguinte modificação, na classe BattleshipApp:

```
private void drawGrids() {
   /* UM BOCADO DE CÓDIGO ENTRA AQUI */

    // Desenha os navios nas grades do tabuleiro.
    this.redBoard.drawShips(gfx, Color.gray,
                            RED_GRID_X,
                            RED_GRID_Y);

    this.blueBoard.drawShips(gfx, Color.black,
                             BLUE_GRID_X,
                             BLUE_GRID_Y);
}
```

Agora podemos jogar um pouco mais honestamente. Mas, mesmo sem poder ver o tabuleiro do inimigo, ainda podemos vencer com enorme facilidade. Isso não é muito divertido...

Precisamos melhorar a estratégia de jogo do nosso oponente. Como sempre, isso pode ser feito de diversas maneiras. Freqüentemente, jogos irão "trapacear" – afinal seria fácil examinar o array gridCells e implementar uma estratégia de jogada para o computador que só atira em uma célula que contém parte de um navio. Nesse caso, nosso oponente seria invencível! Mas, novamente, isso não seria nada divertido.

É claro que poderíamos também pensar nas mais mirabolantes e complexas estratégias de jogada. Mas o que vamos fazer aqui é implementar uma estratégia mais simples, a

qual é descrita a seguir.

A nova estratégia de jogada

Como dissemos anteriormente, em um jogo de Batalha Naval geralmente atiramos aleatoriamente, até que atingimos parte de um navio. Então, como um navio é posicionado ao longo do eixo vertical ou do eixo horizontal, procuramos, nas jogadas seguintes, atirar em células imediatamente acima, abaixo, à esquerda, ou à direita da posição onde acertamos um navio. Portanto, as células em torno do ponto onde se atingiu um navio são "muito boas" (com grande probabilidade de conter outra parte do navio), aquelas logo em volta dessas primeiras são "boas" (podem conter parte do navio, se ele não for um bote), etc. Essa idéia é ilustrada na figura a seguir:

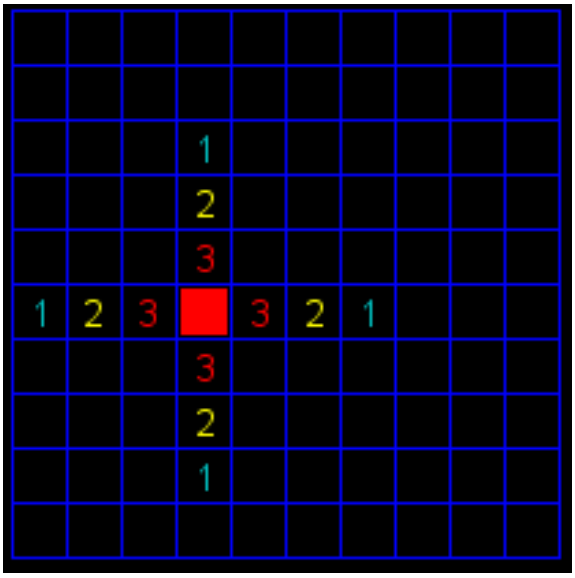


Figura 1 – Alvos potenciais em torno de uma posição onde se atingiu um navio.

Nessa figura, as células maracad com '3' representam as melhores opções para a próxima jogada, aquelas com '2' representam as próximas melhores opções, e as marcadas com '1' as melhores opções depois das anteriores. Vamos supor que todas as demais células têm valor '0' (exceto a célula onde ocorreu o acerto, marcada em vermelho na figura, que tem valor -99).

Agora o computador tem alguma idéia de qual deve ser seu objetivo – ele deve atirar em uma das células com maior valor de alvo potencial. Muito bem... as coisas estão melhorando...

Imagine que, cada vez que o computador atinge um alvo, calculamos valores tais como mostrado na Figura 2, armazenando-os no array gridCells. Suponha que o próximo tiro do computador é dado na célula marcada com '3, imediatamente acima da célula onde foi atingido um navio, e isso resulta em novo acerto. Ao recalcular o esquema de valores '3,2,1', tal como mostrado acima, devemos obter o seguinte:

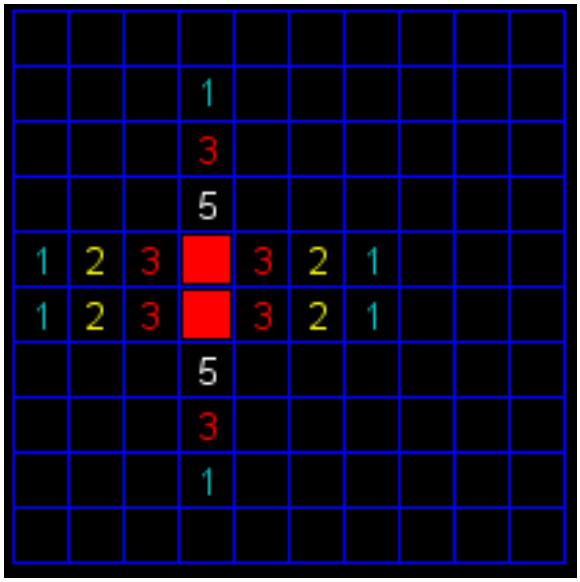


Figura 2 – Valores alvo após o segundo tiro certo.

Nesse caos, os quadrados marcados com '5' indicam as melhores probabilidades de se atingir um alvo. Parece que estamos chegando lá... Mas, o que fazer quando o segundo tiro não resulta em acerto?

Nesse caso, vamos usar números negativos, para indicar posições onde não existe grande probabilidade de se atingir parte de um navio (note que dois navios não podem estar "emendados" no tabuleiro). Então teríamos a seguinte situação:

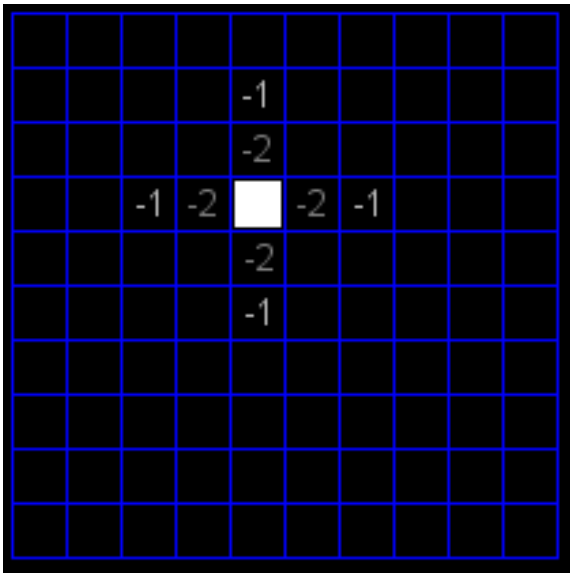


Figura 3 – Valores alvo após um tiro errado.

Novamente, supomos que as células não marcadas na figura têm valor 0. Se o computador sempre procurar atirar nas células com valor alvo mais alto, ele irá ignorar todas as células em torno de uma posição onde não atingiu nenhum navio. Isso faz sentido – como a maioria dos navios tem tamanho igual ou maior que 3, essa estratégia favorece que o computador atire em uma célula que está a uma distância de pelo menos 3 células da posição onde o tiro resultou em erro.

Vamos retornar ao caso anterior, onde o computador acertou dois tiros. Vamos supor que ele agora atira na posição marcada com '5', logo abaixo da área de acerto, e esse tiro

resulta em erro. Como deveriam ficar os valores da grade, nesse caso? Veja a figura a seguir.

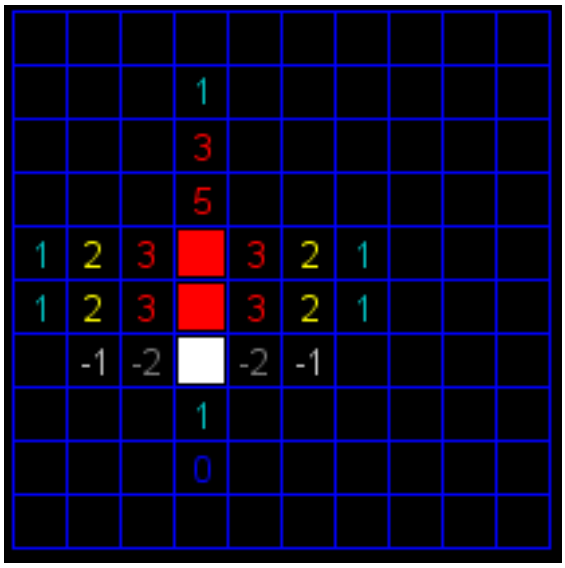


Figura 4 – Aplicando o padrão de erro de tiro, após dois tiros certos anteriores.

Isso parece promissor – o computador deverá atirar, na próxima jogada, na célula marcada com '5', tal como desejamos.

Agora suponha que o computador tenha destruído o bote do inimigo. Isso significa que ele não mais deveria procurar atirar próximo das posições de acerto. Se não recalcularmos os valores das células da grade, o computador irá perder várias jogadas atirando em torno das células de acerto, provavelmente sem nenhum sucesso. Embora isso não seja um problema muito sério, um jogador inimigo humano certamente logo perceberia esse comportamento e tiraria proveito disso. Portanto, temos que tomar alguma providência para evitar esse problema: precisamos "limpar" os valores de alvo calculados anteriormente, quando um navio é afundado.

Vamos fazer isso da maneira mais simples – apenas restabelecer os valores alvo '3,2,1' que tínhamso calculado anteriormente. Em outras palavras, ao afundar um navio, visitamos cada uma das células próximas e subtraímos o padrão 3,2,1. Aplicando essa lógica ao tabuleiro da figura 4, obtemos:

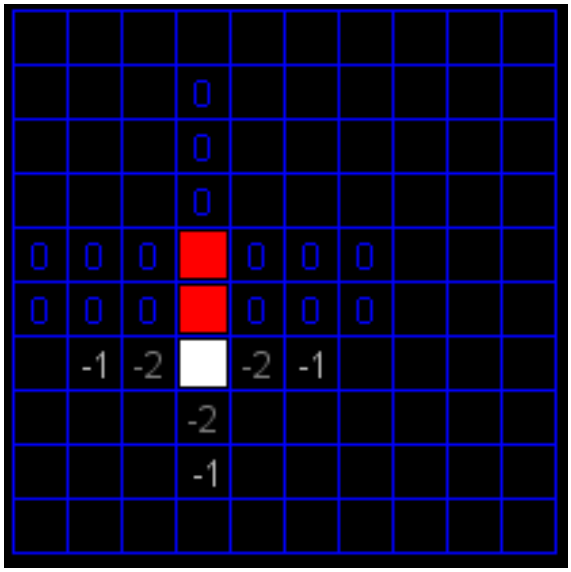


Figura 5 – Valores alvo recalculados, depois de se ter afundado um navio.

Uau – isso não poderia ser mais perfeito! Parece que temos a base do novo cérebro do nosso jogador inimigo.

Loop 'For'

Como você já deve ter adivinhado, vamos precisar de vários loops para implementar nossa nova estratégia de jogada do computador. Toda vez em que o computador atirar, faremos uso de quatro loops: cada um para calcular os valores alvo – ou pesos – das células da grade, em uma das quatro direções (para cima, para baixo, para a esquerda e para a direita). É claro que poderíamos usar nosso velho conhecido while loop, mas vamos economizar um pouco de espaço, usando uma nova forma de loop – o for loop. Vamos ver como isso funciona, antes de usá-lo em nosso código.
Aqui está um exemplo do nosso velho conhecido while loop:

```
int i = 0;
while (i < 10) {
   // Faça alguma coisa.
   i++;
}
```

Note que usamos cores para destacar cada parte importante do loop: inicializar a variável do loop, testair a condição de término do loop, e incrementar a variável do loop. Vamos agora ver o for loop correspondente (que é apenas uma forma abreviada de se escrever o while loop):

```
for (int i=0; i<10; i++) {
   // Faça alguma coisa.
}
```

Uau – realmente bem mais conciso! Note, entretanto, que essa notação mais concise pode causar alguma confusão (pelo menos para os novatos): embora a inicialização, o teste e o incremento da variável do loop apareçam todos na mesma linha, eles serão executados na mesma ordem que no while loop acima.

OK, munidos dessa nova forma de loop, vamos partir para implementar nossa nova estratégia de tiro. Isso será feito na seção a seguir.

Código da nova estratégia de jogada do inimigo

Finalmente estamos prontos para implementar a nova estratégia de jogada do computador. Aqui está uma descrição do código que teremos que escrever:

- Criar e manter informação de peso das células em uma grade (isto é, probabilidade de acerto em cada célula).
- Procurar nessa grade pelas células de maior valor.
- Selecionar uma das células de maior valor.
- Aplicar o esquema 3,2,1 quando um navio do usuário for atingido – "hit".
- Aplicar o esquema -2,-1 quando o tiro não atingir nenhum navio – "miss".

- Desfazer o esquema 3,2,1 sempre que for afundado um navio do usuário.

Puxa, uma porção de tarefas! Vamos começar então.

Um novo começo

Primeiramente, precisamos criar uma estrutura de dados para a representar a informação de probabilidade de se atingir um navio em cada uma das células da grade – ou seja, que contenha um "peso" para cada célula da grade. É claro que essa grade terá as mesmas dimensões da grade visível do tabuleiro, e conterá valores inteiros. Cada célula deverá ter, inicialmente, o valor 0. O valor -99 será usado para indicar células sobre as quais o computador já atirou. Com base nessa idéias, adicionamos os seguinte, à classe EnemyPlayer:

```
public class EnemyPlayer
{
   // variáveis de classe.
   private static final int AI_GRID_EMPTY     = 0;
   private static final int AI_GRID_TARGETED   = -99;

   // variáveis de instância.
   private int targetRow;
   private int targetColumn;
   private Vector bag;
   private int[][] aiGrid;

   // Métodos.
   /**
    * Construtor de objetos da classe EnemyPlayer
    */
    public EnemyPlayer()
   {
      this.bag = new Vector();

      this.aiGrid =
            new int[Board.ROW_COUNT][Board.COLUMN_COUNT];
      this.initAIgrid();

      int iRow = 0;
       while (iRow < Board.ROW_COUNT) {
         int iCol = 0;
          while (iCol < Board.COLUMN_COUNT) {
             Target boardPiece = new Target(iRow, iCol);
             this.bag.addElement(boardPiece);
            iCol++;
         }
         iRow++;
      }
   }

    /** Inicializa os pesos das células da grade de AI. */
    public void initAIgrid() {
       for (int i=0; i<Board.ROW_COUNT; i++) {
          for (int j=0; j<Board.COLUMN_COUNT; j++) {
```

```
                this.aiGrid[i][j] = AI_GRID_EMPTY;
            }
        }
    }

    /* RESTANTE DA CLASSE AQUI... */
```

Note que usamos for loops para inicializar as células da nova grade. Observe como o código ficou bem mais compacto!
Bem, vamos então redefinir o nosso método selectTarget.

```
/* Escolhe aleatoriamente uma linha e coluna
  * para a célula alvo do tiro.
*/
public void selectTarget() {
    long seed = System.currentTimeMillis();
    Random randomizer = new Random(seed);

    // Obtém a célula de maior peso do tabuleiro.
    int maxWeight = AI_GRID_TARGETED;
    for (int i=0; i<Board.ROW_COUNT; i++) {
        for (int j=0; j<Board.COLUMN_COUNT; j++) {
            if (this.aiGrid[i][j] > maxWeight) {
                maxWeight = this.aiGrid[i][j];
            }
        }
    }

    // Cria um alvo para cada célula, com o valor máximo de
    // peso e adiciona esses alvos aos saco.
    for (int i=0; i<Board.ROW_COUNT; i++) {
        for (int j=0; j<Board.COLUMN_COUNT; j++) {
            if (this.aiGrid[i][j] == maxWeight) {
                Target newTarget = new Target(i, j);
                this.bag.addElement(newTarget);
            }
        }
    }

    // Obtém uma célula do saco.
    int piecesInBag = this.bag.size();
    int piece = randomizer.nextInt(piecesInBag);
    Target target = (Target)this.bag.elementAt(piece);
    this.targetRow = target.row;
    this.targetColumn = target.column;

    // Esvazia o saco .
    this.bag.removeAllElements();

    // Chama o coletor de lixo.
    System.gc();
}
```

Vamos examinar o código deste método mais detalhadamente. Comecemos por aqui:

```
// Obtém a célula de maior peso do tabuleiro.
int maxWeight = AI_GRID_TARGETED;
for (int i=0; i<Board.ROW_COUNT; i++) {
    for (int j=0; j<Board.COLUMN_COUNT; j++) {
       if (this.aiGrid[i][j] > maxWeight) {
          maxWeight = this.aiGrid[i][j];
      }
   }
}
```

Você verá código semelhante a esse freqüentemente: ele implementa a pesquisa por um valor máximo em uma lista de valores (o código para pesquisar por um valor mínimo é semelhante). Começamos atribuindo um valor "extremamente pequeno" à variável 'maxWeight' (nesse caso, AI_GRID_TARGETED, que é igual a -99). Em cada iteração do loop sobre os elementos da lista (nesse caso, usamos loops aninhados para percorrer a lista, uma vez que ela é representada na forma de um array bidimensional), verificamos se o valor corrente da lista é maior que o valor máximo corrente, armazenado em maxWeight:

```
if (this.aiGrid[i][j] > maxWeight) {
    maxWeight = this.aiGrid[i][j];
}
```

Se o valor da célula corrente for maior que o máximo corrente, então o valor máximo é atualizado com o valor da célula. Observe como isso funciona, no nosso caso. Inicialmente, preenchemos todas as células do array que representa a grade com o valor AI_GRID_EMPTY (ou seja, 0). Portanto, quando o loop começa, com i=0 and j=0, a execução do comando:

```
if (this.aiGrid[i][j] > maxWeight)
```

se reduz a if (this.aiGrid[0][0] > -99). Como toda célula contém o valor 0, e 0 > -99, o corpo da cláusula if é executado:

```
maxWeight = this.aiGrid[i][j];
```

Isso atribui a maxWeight o valor AI_GRID_EMPTY (ou seja, 0). Ao terminarmos o loop sobre todas as células da grade, repetindo esse processo a cada iteração, maxWeight conterá o valor máximo contido nas células da grade. Muito fácil, não é?
Depois de obter o maior valor contido na grade, colocamos no saco todas as células que contém esse valor. Isso é implementado pelo seguinte código:

```
// Cria um alvo para cada célula, com o valor máximo de
// peso e adiciona esses alvos aos saco.
for (int i=0; i<Board.ROW_COUNT; i++) {
    for (int j=0; j<Board.COLUMN_COUNT; j++) {
       if (this.aiGrid[i][j] == maxWeight) {
          Target newTarget = new Target(i, j);
          this.bag.addElement(newTarget);
      }
   }
}
```

Note que, mais uma vez, usamos loops aninhados para percorrer as células da grade. Em cada iteração, verificamos se a célula corrente contém o valor desejado: if (this.aiGrid[i][j] == maxWeight. Em caso afirmativo, criamos um alvo correspondente a essa célula e o colocamos no saco.

Ao final da execução desse trecho de código, o saco conterá a lista das células candidatas para alvo do tiro corrente. Selecionamos um desses alvos do saco, usando o código que escrevemos anteriormente. Depois de selecionar a célula alvo, usamos o seguinte comando, para remover do saco todas as células alvo candidatas: this.bag.removeAllElements();

Finalmente, usamos o comando System.gc() para encorajar o sistema de execução de programas Java a "limpar" todos os alvos que acabamos de criar. Esse é um conceito importante. Sempre que criamos objetos, usando o operador new, parte da memória do computador é alocada para armazenar os dados desse novo objeto. Quanto mais objetos forem criados, maior será a área de memória usada pelo programa e, em muitos casos, isso pode, como conseqüência, tornar o programa mais lento. No caso do nosso jogo de Batalha Naval, esse problema não será perceptível. Entretanto, é bom criar o hábito de "limpar" da área de memória os objetos que não mais serão usados. De fato, esses objetos que não mais serão usados são automaticamente removidos pelo "Coletor de Lixo" (em inglês, "Garbage Collector") do sistema de execução de programas Java, o qual é ativado, de tempos em tempos, durante a execução de um programa, para realizar a coleta de lixo. O comando System.gc() simplesmente força a chamada "Coletor de Lixo" de Java, nesse ponto específico do programa. Como criamos uma coleção de possíveis alvos, selecionamos um deles, e não usaremos nenhum dos outros (nossa coleção é esvaziada), é uma boa idéia forçar a chamada do coletor de lixo, nesse ponto, para liberar a área de memória que havia sido alocada para esse objetos, que não mais serão utilizados.

Em geral, não precisamos nos preocupar em chamar "Coletor de Lixo" explicitamente. Entretanto, em alguns casos, isso é recomendável, tal como no caso que do nosso código acima.

OK, já implementamos nosso método para selecionar uma célula alvo do tiro. Agora só falta aplicar nosso esquema 3,2,1 ou -2,-1, conforme o alvo tenha sido atingido ou não, e então teremos completado a implementação da novoa estratégia de jogada do computador. Vamos então fazer isso!

Calculando os pesos das células da grade

Para calcular os pesos das células da grade, devemos fazer o seguinte:

- Aplicar o esquema 3,2,1, para recalcular os pesos das células, caso o tiro tenha atingido parte de um navio do adversário.
- Aplicar o esquema -2,-1, para recalcular os pesos das células, caso o tiro não tenha atingido nenhum navio.
- Desfazer o esquema 3,2,1, caso o tiro tenha afundado um navio do adversário.

Antes de escrever o código para isso, vejamos onde ele se encaixa, no corpo do método mouseReleased, da classe BattleshipApp:

```
else {
```

```
        this.enemyPlayer.selectTarget();
        row = this.enemyPlayer.getTargetRow();
        col = this.enemyPlayer.getTargetColumn();

        if (this.redBoard.didHitShip(row, col)) {
            this.redBoard.applyDamage(row, col);
            this.redBoard.putPegInSquare(row, col, true);

            // Verifica se foi afundado um navio do usuário
            // SE SIM...desfaça o esquema 3,2,1.
            // SE NÃO...calcule um novo esquema 3,2,1.
        }
        else {
            this.redBoard.putPegInSquare(row, col, false);
            // Aplique o esquema -2, -1.
        }

        /* Verifica se o computador venceu */
        if (!this.redBoard.isAnyShipLeft()) {
            this.setMessage("VocÊ perdeu...", true);
            this.gameState = GAME_STATE_GAMEOVER;
        }
    }
```

Vamos começar pela aplicação do esquema -2, -1. Vamos adicionar, à classe EnemyPlayer, uma nova variável de instância – missWeights – para representar os valores a serem aplicados às células no caso do esquema -2,-1, e um novo método – applyMissWeights – para calcular os novos valores das células, nesse caso:

```
// variáveis de instância.
private int targetRow;
private int targetColumn;
private Vector bag;
private int[][] aiGrid;
private int[] missWeights = {-2, -1};

/** Aplica o esquema -2, -1 à grade aiGrid. */
public void applyMissWeights(int row, int col) {
    // Primeiro, preenche todas as células alvo com o valor -99.
    this.aiGrid[row][col] = AI_GRID_TARGETED;

    // Em seguida, tenta aplicar os pesos às células acima.
    for (int i=0; i<this.missWeights.length; i++) {
        int r = row - 1 - i;
        if (r < 0) break;
        this.aiGrid[r][col] = this.aiGrid[r][col] +
                              this.missWeights[i];
    }

    // Em seguida, aplica os pesos às células abaixo.
    for (int i=0; i<this.missWeights.length; i++) {
        int r = row + 1 + i;
        if (r >= Board.ROW_COUNT) break;
        this.aiGrid[r][col] = this.aiGrid[r][col] +
                              this.missWeights[i];
```

```
    }

    // Idem, para as células à direita.
    for (int i=0; i<this.missWeights.length; i++) {
       int c = col + 1 + i;
        if (c >= Board.COLUMN_COUNT) break;
        this.aiGrid[row][c] = this.aiGrid[row][c] +
                        this.missWeights[i];
    }

    // Finalmente, idem para as células à esquerda.
    for (int i=0; i<this.missWeights.length; i++) {
       int c = col - 1 - i;
       if (c < 0) break;
        this.aiGrid[row][c] = this.aiGrid[row][c] +
                        this.missWeights[i];
    }
}
```

Observe a primeira ação descrita no corpo deste método: ela armazena na célula escolhida como alvo o valor AI_GRID_TARGETED (que é igual a -99), para evitar que essa célula seja novamente escolhida como alvo, em uma jogada posterior.
Em seguida, o corpo do método contém quatro seções de código praticamente idênticas, correspondentes às quarto possíveis direções – para cima, para baixo, para a esquerda e para a direita. Vamos examinar apenas a primeira delas:

```
// Em seguida, tente aplicar os pesos às células acima.
for (int i=0; i<this.missWeights.length; i++) {
   int r = row - 1 - i;
    if (r < 0) break;

    this.aiGrid[r][col] = this.aiGrid[r][col] +
                    this.missWeights[i];
}
```

Como isso funciona? Bem, começamos, mais uma vez, com uma for loop, que itera sobre os dois elementos do array missWeights, que contêm os valores -2 e -1. Portanto, o comando acima se reduz ao seguinte:

```
for (int i=0; i<2; i++) {
```

Calculamos, então, a linha da célula cujo conteúdo queremos modificar, usando o comando:

```
int r = row - 1 - i;
```

Suponha que o computador atirou na célula da linha 5, coluna 1 (lembre-se que linhas e colunas são numeradas a partir de 0). Então, na primeira iteração do loop, o comando acima equivale a

```
int r = 5 - 1 - 0; // r = 4 na primeira iteração do loop.
```

Suponha, ao invés disso, que o computador atirou na célula da linha 0, coluna 4. Nesse caso, 'r' seria igual a 0 - 1 - 0, ou seja -1 – que está for a dos limites do tabuleiro!

Felizmente, o comando a seguir trata esse caso:

```
if (r < 0) break;
```

No caso em que o valor de r é negativo (isto é, acima da grade do tabuleiro), o comando break faz com que a execução do loop termine imediatamente.
Se 'r' está dentro dos limites do tabuleiro, é executado o seguinte comando:

```
this.aiGrid[r][col] = this.aiGrid[r][col] + this.missWeights[i];
```

Observe como isso funciona, considerando o nosso exemplo, em que o tiro foi dado na célula da linha 5, coluna 1. Nesse caso, temos: row=5, column=1, e, na primeira iteração do loop, i = 0. Suponha que aiGrid[4][1] contém o valor 1. Então, temos:

r = 5 - 1 - 0 = 4

col = 1

aiGrid[r][col] = aiGrid[4][1] = 1

missWeights[i] = missWeights[0] = -2

Portanto, o comando this.aiGrid[r][col] = this.aiGrid[r][col] + this.missWeights[i]; se reduz a this.aiGrid[4][1] = this.aiGrid[4][1] + this.missWeights[0]; ou seja, this.aiGrid[4][1] = 1 + -2 = -1. Nesse ponto, chegamos ao final do corpo do loop, quando é executado o comando i++, fazendo com que i passe a ter o valor 1. Voltamos então ao início do loop, onde é feito o teste (is i < 2?). Como i é igual a 1, o corpo do loop é novamente executado. Desta vez, temos r = 5 - 1 - 1 = 3, e missWeights[i] é igual a missWeights[1], que é igual a -1. Como r ainda é maior que 0 (ainda está dentro dos limites do tabuleiro), o comando break não é executado e, portanto, adicionamos -1 ao valor armazenado em aiGrid[3][1]. Não é muito complicado, não é mesmo?

Como dissemos anteriormente, os três outros trechos de código desse método se comportam de maneira muito semelhante ao primeiro – eles apenas aplicam os "pesos" a direções diferentes na grade. Dê uma olhada nesses outros trechos do código e veja se você consegue entender como funcionam.

O tiro atingiu um navio

Você provavelmente já deve imaginar como deve ser o código para aplicar o esquema 3,2,1, no caso em que o tiro atinge parte de um navio. Entretanto, tenha em mente que devemos "desfazer" esse esquema quando o navio é afundado (isto é, todas as suas partes foram atingidas). O código para implementar esse esquema é apresentado a seguir. Para isso, introduzimos uma nova variável de instância e um novo método.

```
// variáveis de instância.
private int targetRow;
private int targetColumn;
private Vector bag;
private int[][] aiGrid;
private int[] missWeights = {-2, -1};
private int[] hitWeights = {3, 2, 1};

/** Aplica o esquema 3,2,1 à grade aiGrid. */
```

```
public void applyHitWeights(int row, int col, boolean bUndo) {
    // Primeiro, preenche todas as células alvo com o valor -99.
     this.aiGrid[row][col] = AI_GRID_TARGETED;

    // Em seguida, tenta aplicar os pesos às células acima.
    for (int i=0; i<this.missWeights.length; i++) {
       int r = row - 1 - i;
        if (r < 0) break;

        if (bUndo) {
            this.aiGrid[r][col] = this.aiGrid[r][col] -
                          this.hitWeights[i];
       }
       else {
            this.aiGrid[r][col] = this.aiGrid[r][col] +
                          this.hitWeights[i];
       }
    }

   // Em seguida, aplica os pesos às células abaixo.
    for (int i=0; i<this.missWeights.length; i++) {
       int r = row - 1 + i;
        if (r >= Board.ROW_COUNT) break;

        if (bUndo) {
            this.aiGrid[r][col] = this.aiGrid[r][col] -
                          this.hitWeights[i];
       }
       else {
            this.aiGrid[r][col] = this.aiGrid[r][col] +
                          this.hitWeights[i];
       }
    }

     // Idem para a direita.
    for (int i=0; i<this.missWeights.length; i++) {
       int c = col + 1 + i;
        if (c >= Board.COLUMN_COUNT) break;

        if (bUndo) {
            this.aiGrid[row][c] = this.aiGrid[row][c] -
                          this.hitWeights[i];
       }
       else {
            this.aiGrid[row][c] = this.aiGrid[row][c] +
                          this.hitWeights[i];
       }
    }

    // Finalmente, idem para a esquerda.
    for (int i=0; i<this.missWeights.length; i++) {
       int c = col +-1 - i;
       if (c < 0) break;
        if (bUndo) {
            this.aiGrid[row][c] = this.aiGrid[row][c] -
```

```
                        this.hitWeights[i];
        }
        else {
            this.aiGrid[row][c] = this.aiGrid[row][c] +
                        this.hitWeights[i];
        }
    }
}
```

Como o método acima é quase idêntico ao método applyMissWeights, não vamos discuti-lo em detalhe. Você deve entretanto observer que foi usada uma nova variável – bUndo – para indicar quando o esquema 3,2,1 deve ser desfeito. Quando o método applyHitWeights é chamado com o valor de bUndo igual a 'true', os pesos são subtraídos, e não adicionados, as células da grade. Portanto, o método acima é usado, tanto para aplicar, como para desfazer, o esquema 3,2,1.

Finalizando

Estamos quase terminando. Só falta chamar os novos método no corpo do método mouseReleased:

```
else {
    this.enemyPlayer.selectTarget();
    row = this.enemyPlayer.getTargetRow();
    col = this.enemyPlayer.getTargetColumn();
    if (this.redBoard.didHitShip(row, col)) {
        this.redBoard.applyDamage(row, col);
        this.redBoard.putPegInSquare(row, col, true);

        // Verifica se foi afundado um navio do usuário
        // SE SIM...desfaça o esquema 3,2,1.
        else {
            // ...apply a new 3,2,1 pattern.
            this.enemyPlayer.applyHitWeights(row, col, false);
        }
    }
    else {
        this.redBoard.putPegInSquare(row, col, false);
        // aplica o esquema -2,-1
        this.enemyPlayer.applyMissWeights(row, col);
    }

    /* Verifica se o computador venceu */
    if (!this.redBoard.isAnyShipLeft()) {
        this.setMessage("Você perdeu...", true);
        this.gameState = GAME_STATE_GAMEOVER;
    }
}
```

Bem, falta ainda verificar se o navio foi afundado, ou seja, todas as suas partes foram atingidas. Lembra-se que o nosso método applyDamage retorna o tipo do navio? Bem, talvez seja mais útil que ele retorne o próprio navio, para que possamos verificar se ele

foi afundado. Vamos então fazer essa alteração:

```
/** Aplica dano ao navio atingido por tiro na célula indicada por (row, col). */
public Ship applyDamage(int row, int col) {
    int shipType = this.gridCells[row][col];
    Ship hitShip = this.fleet[shipType];
    hitShip.takeHit();

    return hitShip;
}
```

Agora podemos facilmente determinar se o navio já foi afundado:

```
if (this.redBoard.didHitShip(row, col)) {
    Ship hitShip = this.redBoard.applyDamage(row, col);
    this.redBoard.putPegInSquare(row, col, true);

    // Verifica se foi afundado um navio do usuário
    if (hitShip.getHits() == hitShip.getSize())
   {

      // SE SIM...desfaça o esquema 3,2,1.
       this.enemyPlayer.undoWeights(hitShip);
   }
   else {
      // ...apply a new 3,2,1 pattern.
       this.enemyPlayer.applyHitWeights(row, col, false);
   }
}
```

Agora, basta escrever a definição do método undoWeights, na classe EnemyPlayer. Como esse método deve funcionar? Bem, ele deve iterar sobre cada célula onde está posicionado o navio e "desfazer" a informação de pesos nessas células. Como o navio pode estar posicionado segundo uma das quatro possíveis orientações, temos que examinar cada um desses casos:

```
public void undoWeights(Ship ship) {
   int row = ship.getRow();
   int col = ship.getColumn();
   int size = ship.getSize();

    if (ship.getOrientation() == Ship.ORIENTATION_UP) {
      for (int i=0; i<size; i++) {
         this.applyHitWeights(row - i, col, true);
      }
   }
    else if (ship.getOrientation() == Ship.ORIENTATION_DOWN) {
      for (int i=0; i<size; i++) {
         this.applyHitWeights(row + i, col, true);
      }
   }
    else if (ship.getOrientation() == Ship.ORIENTATION_RIGHT) {
      for (int i=0; i<size; i++) {
         this.applyHitWeights(row, col + i, true);
      }
```

```
    }
     else if (ship.getOrientation() == Ship.ORIENTATION_LEFT) {
        for (int i=0; i<size; i++) {
            this.applyHitWeights(row, col - i, true);
        }
    }
}
```

Pronto! Acredite ou não, concluímos a implementação da nova inteligência do nosso jogador computador. Tente compilar o seu programa. Se você tiver problemas, confira os códigos das suas classes BattleshipApp (código 25a), EnemyPlayer (código 25b) , Ship (código 25c) e Board (código 25d), na seção de códigos incluída no final deste tutorial.

Quando seu programa estiver funcionando, aproveite para jogar um pouco contra o computador. Veja se você consegue ganhar. Quando você já tiver se cansado de brincar, reveja o que aprendemos neste módulo. .

Conclusão

Parece que o nosso jogo agora está mais divertido, não é? Acabamos de implementar a nova estratégia de jogada do computador. Agora já não é tão fácil ganhar do nosso adversário. O que você aprendeu de novo, ao implementarmos a nova estratégia de jogada?

Bem, você aprendeu a usar uma nova forma de comando de repetição – o comando for. Como dissemos, esse comando é simplesmente uma forma mais abreviada de escrever o nosso velho conhecido comando while, mas é mais adequado, em determinados casos, pois resulta em um código mais conciso.

Você aprendeu também como implementar o algoritmo de pesquisa por um valor máximo (ou mínimo) em uma lista de valores. Esse tipo de problema ocorre freqüentemente em diversas aplicações.

Se você ainda não brincou com o código do seu jogo, está na hora. Seja criativo: tabuleiros maiores... maior número de navios... número limitado de tiros... Você pode inventar várias regras novas e tentar implementar diferentes versões do jogo.

No próximo módulo, vamos refinar um pouco mais o nosso jogo. Vamos lá? .

Módulo 10

Aprimorando o jogo

Introdução

Este é o último módulo do nosso tutorial. Neste módulo, vamos nos dedicar a dar alguns toques finais no nosso jogo.

Já criamos o código para um jogo completo, mas ainda bastante simplificado. Neste módulo, vamos começar revendo o nosso projeto, para eliminar as classes que não foram utilizadas. Depois, vamos introduzir algumas modificações para tornar o jogo mais amigável para o usuário: possibilitar que o usuário recomece o jogo, depois de ter terminado uma batalha e melhorar a janela de estado do jogo.

Finalmente, vamos sugerir alguns aspectos que você poderá aprimorar, em uma versão mais sofisticada do seu jogo.

Vamos começar então?

Limpando o código

Antes de começar a adicionar novas facilidades ao nosso jogo, precisamos limpar um pouco o nosso projeto. Mencionamos anteriormente que o nosso projeto continha várias classes que, de fato, não foram utilizadas e deviam, portanto ser removidas. Vamos fazer isso agora.

Se você abrir seu projeto no navegador de projeto do BlueJ, verá que ele tem as seguintes classes, que não foram utilizadas: RedPeg, WhitePeg e FriendlyPlayer. Não adicionamos nenhum código a essas classes, e elas não são referenciadas em nenhum ponto do resto do projeto. Remover cada uma dessas classes é muito simples. Basta clicar sobre uma das classes, na janela do navegador de projeto, e selecionar a opção 'Remove', como mostra a figura a seguir..

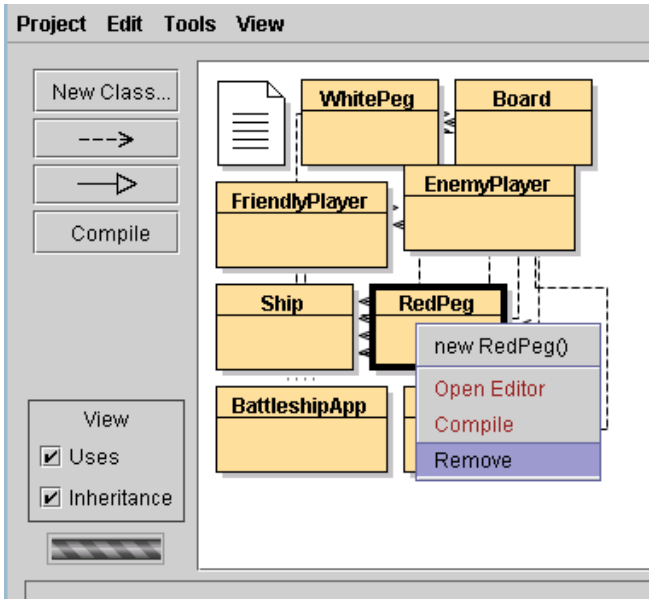


Figura 1 — Removendo uma classe do projeto.

Faça isso então – remova as classes Go ahead and remove RedPeg, WhitePeg e FriendlyPlayer. Você precisará também remover a referência à classe FriendlyPlayer existente na classe BattleshipApp:

```
// variáveis de instância.
    private Board redBoard;
    private Board blueBoard;
    private FriendlyPlayer friendlyPlayer;      // Remova esta linha!
     private EnemyPlayer enemyPlayer;
    private int gameState;
     private boolean bNeedPositionClick;
    private String userMessage = null;
```

Além disso, devemos remover as referências a RedPeg e WhitePeg da classe Board:

```
public class Board
{
   // variáveis de classe.
   // ...algumas variáveis declaradas aqui...
// variáveis de instância.
   private RedPeg[] hitMarkers;          // Remova!
    private WhitePeg[] missMarkers;     // Remova!
   private Ship[] fleet;
   // ...restante da classe.
}
```

Compile e execute novamente o seu projeto, apenas para certificar-se de que tudo funciona corretamente. Se ocorrer algum problema, é porque você provavelmente removeu a classe errada. Se isso acontecer, você poderá baixar o projeto completo, a partir do endereço http://see.photogenesis.com.br/lucilia/projetos/battleship9.zip.

Além de limpar o código, seria bom torná-lo mais amigável para o usuário. EM particular, gostaríamos que o jogador pudesse jogar mais de uma partida, sem ter que fechar e reiniciar o jogo. Além disso, escondemos um pequeno erro do nosso jogo: quando fechamos o jogo, ele de fato não termina – sua janela apenas é fica invisível. Vamos começar corrigindo esse pequeno erro.

Corrigindo um pequeno erro

Vamos agora corrigir o erro que evita que a janela do jogo de fato seja destruída, quando o jogo é terminado. Na verdade, isso não é exatamente um erro – em versões anteriores da linguagem Java, ao terminar um programa, sua janela não era destruída, mas apenas fechada (tornando-se invisível). Além disso, quando o programa é executado dentro do ambiente BlueJ, tudo funciona corretamente. Mas se o programa for executado a mártir da linha de comando – isto é, do promt da janela de comandos do sistema operacional – você irá notar o problema. Quer ver?
Abra uma janela de terminal do seu sistema Unix. Modifique o diretório corrente para o diretório que contém o seu projeto, tal como no exemplo a seguir:

```
>cd c:/BlueJ/projects/Battleship9
```

Agora, execute o compilador Java, indicando onde ele deve encontrar as classes do seu projeto (ou seja, definindo a variável 'classPath') e indicando o nome do arquivo que contém a classe principal do seu projeto (isto é, a classe que contém o método main):

```
>java -classpath c:/BlueJ/projects/Battleship9 BattleshipApp
```

(no comando acima, a expressão 'c:\BlueJ\projects\battleship9', imediatamente depois de –classpath, indica para o interpretador Java onde encontrar as classes do seu projeto; o argumento 'BattleshipApp' especifica a classe que contém o método 'main').

Ao executar esse comando, você deverá ver seu jogo ser exibido na tela normalmente. Clique então sobre o ícone de fechar janela, no canto superior direito da mesma. O jogo irá desaparecer, mas você não poderá entrar com nenhum outro comando na janela de comandos do sistema. Isso ocorre porque o programa do jogo de Batalha Naval, de fato, ainda está sendo executado. Se você pressionar simultaneamente as teclas 'ctrl-break', você verá uma mensagem de erro na janela do sistema. Se você pressionar 'ctrl-c', em seguida, o programa será terminado e novamente você poderá entrar com um comando na janela do sistema.

Esse não é um comportamento muito desejável!

Felizmente, a solução é simples. Quando a janela do jogo é fechada, é sinalizado um evento correspondente, o qual pode ser tratado no programa. O tratamento deste evento deve conter as ações necessárias para encerrar a execução do programa. Lembre-se que, em módulos anteriores, usamos um 'mouseListener' para tratar eventos relacionados ao mouse. A mesma estratégia será usada aqui, exceto que vamos usar um 'windowListener', uma vez que o evento que desejamos tratar é um evento de janela.

Para implementar essa solução, vamos precisar do seguinte:

1. Escolher uma classe que será nosso 'windowListener'.

2. Adicionar os métodos requeridos pela interface 'windowListener'.

3. Definir a classe como 'ouvinte' de eventos de janela.

4. Programar o método que trata o evento de fechar a janela, de maneira que ele finalize a execução do programa.

Como sempre, isso parece mais complicado do que de fato é.

Ouvindo o evento!

Primeiro, precisamos escolher a classe que sera nosso "windowListener". Poderia ser qualquer classe, mas o mais razoável, no nosso projeto, seria escolher a classe BattleshipAp – afinal, ela já a ouvinte de eventos do mouse. Além disso, ela é a classe que gerencia todo o jogo, o que a torna a escolha ideal para ouvinte de eventos de janela, uma vez que ele pode propagar esses eventos para qualquer outra classe do jogo, se desejado.
Precisamos então preparar essa classe para ser o ouvinte de eventos de janela. Para isso, ela deve implementar os métodos da interfae windowListener, que são os seguintes:

```
public void windowActivated(WindowEvent event);
public void windowClosed(WindowEvent event);
```

```
public void windowClosing(WindowEvent event);
public void windowDeactivated(WindowEvent event);
public void windowDeiconified(WindowEvent event);
public void windowIconified(WindowEvent event);
public void windowOpened(WindowEvent event);
```

Puxa! Uma porção de métodos! E estamos interessados em apenas um deles – 'windowClosing'. Como modificaríamos a classe BattleshipApp para ser ouvinte de eventos de janela? Aqui está:

```
public class BattleshipApp extends JFrame
                  implements MouseListener, WindowListener
{
   // Um bocado de código entra aqui...
 ////////////////////////////////////////////////////////////////////////

   // WindowListener Interface
    ////////////////////////////////////////////////////////////////////////
   public void windowActivated(WindowEvent event) {}
   public void windowClosed(WindowEvent event) {}
   public void windowClosing(WindowEvent event) {}
   public void windowDeactivated(WindowEvent event) {}
   public void windowDeiconified(WindowEvent event) {}
   public void windowIconified(WindowEvent event) {}
   public void windowOpened(WindowEvent event) {}
}
```

Com essa modificações, deve ser possível compilar a classe. Entretanto, ainda não registramos a classe como ouvinte de eventos de janela, nem inserimos qualquer código para tartar o evento de fechar a janela –"windowClosing". Vamos fazer isso agora. Começaremos modificando o contrutor da classe BattleshipApp, de modo a registrar essa classe como ouvinte de eventos de janela:

```
/**
 * Construtor de objetos da classe BattleshipApp
 */
public BattleshipApp()
{
    this.setSize(GAME_WIDTH, GAME_HEIGHT);
  this.setVisible(true);
 this.addMouseListener(this);
this.addWindowListener(this);
 this.gameState = GAME_STATE_INTRO;

// Inicializa os tabuleiros do jogo.
 this.redBoard = new Board(PLAYER_STATUS_X, STATUS_Y, Color.red);;
this.blueBoard = new Board(COMPUTER_STATUS_X, STATUS_Y, Color.blue);

 // Inicializa o jogador inimigo – o computador.
  this.enemyPlayer = new EnemyPlayer();
}
```

Para concluir, devemos escrever o código do método windowClosing, de maneira que o jogo seja terminado e a janela do jogo seja fechada:

```
public void windowClosing(WindowEvent event) {
   this.dispose();
}
```

Compile e execute o seu programa, a partir da janela de comandos do sistema. Clique sobre o botão de fechar a janela. Você deverá ver que o prompt e o cursos reaparecem na janela de comandos do sistema, o que indica que a execução do programa foi terminada.

Você poderá estar se perguntando: "Porque escolhemos implementar o método 'windowClosing', e não 'windowClosed'? Bem, o sistema de execução de programas Java envia uma notificação do evento 'windowClosing' quando o usuário clica sobre o botão de fechar a janela, ou usa o menu do sistema para fechar a aplicação. O evento 'windowClosed' é notificado sempre que alguém realize uma chamada do método 'dispose' sobre a janela. Portanto, no nosso caso, devemos implementar o método windowClosing, uma vez que o que desejamos é tomar alguma ação quando o usuário clica sobre o botão de fechar a janela.

E agora? Vamos jogar de novo?

Código para reiniciar o jogo

Queremos dar ao usuário a oportunidade de jogar várias vezes, sem que, para isso, ele tenha que terminar o jogo e reiniciar sua execução. Vamos então adiciona a facilidade de "jogar de novo". Em particular, vamos modificar o código do programa, de maneira que, se o usuário clicar sobre a janela do jogo, ao final de uma partida, o jogo de Batalha Naval inicie outra vez, no estado de preparação do jogo.
A primeira parte dessa alteração é fácil – vamos modificar a mensagem de 'fim de jogo' para "Você ganhou! Clique o mouse, para jogar de novo", ou "Você perdeu...clique o mouse, para jogar de novo." Em seguida, vamos adicionar código ao estado de fim de jogo – GAME_OVER – do método mouseListener, para retornar ao estado de preparação do jogo – GAME_SETUP:

```
public void mouseReleased(MouseEvent event) {
   if (this.gameState == GAME_STATE_INTRO) {
      // Intro code here.
  }
   else if (this.gameState == GAME_STATE_SETUP) {
      // Setup code here.
  }
   else if (this.gameState == GAME_STATE_PLAYING) {
      // More code goes here...

         /* Verifica se você venceu o jogo */
         if (!this.blueBoard.isAnyShipLeft()) {
             this.setMessage("Você venceu! Clique o mouse para jogar de novo", true);
            this.gameState = GAME_STATE_GAMEOVER;
        }
        else {
           // ...and here...

            /* Verifica se o computador venceu o jogo */
```

```
                if (!this.redBoard.isAnyShipLeft()) {
                  this.setMessage(
                      "Você perdeu... Clique o mouse para jogar de novo.", true);
                   this.gameState = GAME_STATE_GAMEOVER;
                }
              }
            }
            else {
                this.setMessage("Por favor, clique sobre uma célula vazia.", true);
            }

             // Atualiza o tabuleiro, de maneira que os novos pinos sejam exibidos.
            this.repaint();
        }
         else if (this.gameState == GAME_STATE_GAMEOVER) {
            this.gameState = GAME_STATE_SETUP;
        }
    }
```

Compile e execute o seu programa. Brinque um pouco com o seu jogo e verifique se ele funciona ok.
Como você talvez já esperasse, temos alguns problemas. Você deve ter recebido, na janela do terminal do BlueJ, uma mensagem semelhante à seguinte:

```
java.lang.ArrayIndexOutOfBoundsException: 5
        at Board.setFirstRowColumn(Board.java:177)
        at BattleshipApp.mouseReleased(BattleshipApp.java:204)
```

(além de uma enorme quantidade de texto adicional).

Nosso programa de Batalha Naval não funciona corretamente quando o jogo é reiniciado. Vejamos o que está ocorrendo de errado.

Interpretando a mensagem de erro

Observe a primeira linha da mensagem de erro:

```
java.lang.ArrayIndexOutOfBoundsException: 5
```

O que isso quer dizer? Bem, para um principiante, está dizendo que o programa tentou fazer acesso ao elemento de índice [5] de um dos arrays usados no programa, mas esse array não possui tantos elementos (isto é, o índice usado está fora dos limites do array). Para ver onde ocorreu o problema, observe a segunda linha da mensagem de erro:

```
at Board.setFirstRowColumn(Board.java:177)
```

Ela nos diz que o problema ocorre na linha 177 do arquivo Board.java. Mas como podemos encontrar essa linha rapidamente? Será que teremos que posicionar o cursor no início do arquivo e ir contando as linhas, uma a uma, até 177? A resposta, naturalmente, é não. O blueJ pode mostrar para nós os números das linhas do arquivo. Abra o arquivo Board.java no editor de textos do BlueJ e selecione a opção "Preferences" do menu "Options", como mostrado na figura a seguir.

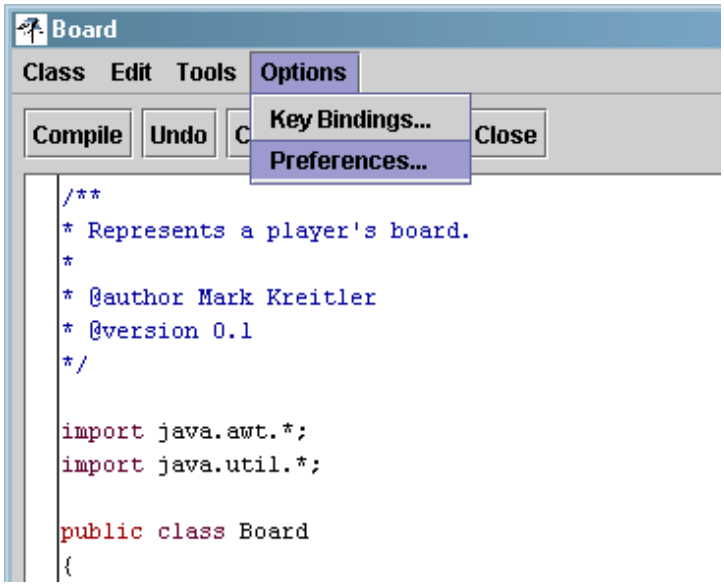


Figura 2 – selecionando a opção "preferences" no editor de textos do BlueJ.

Isso fará com que seja aberta uma nova janela de diálogo, com diversos botões de opção. Clique sobre o botão "Display Line Numbers", para selecionar a opção de que sejam, mostrados os números das linhas do texto (veja a figura a seguir). Em seguida, feche a janela "preferences".

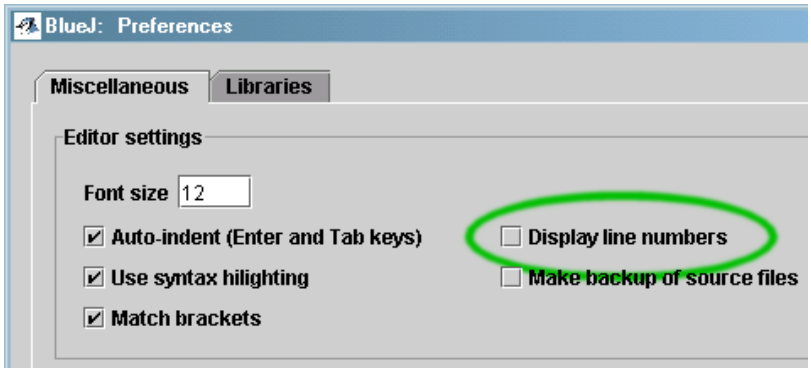


Figura 3 – mostrando números de linhas no BlueJ.

Olhe a margem esquerda da janela do editor de textos que mostra o conteúdo do arquivo Board.java, e você verá a numeração das linhas. Agora fica fácil localizar a linha 177 (veja a figura a seguir 4).

Figura 4 – numeração das linhas do programa.

Agora você pode ver o problema claramente: o valor de this.currentUserShip é muito grande. De acordo com a seguinte linha da mensagem de erro – java.lang.ArrayIndexOutOfBoundsException: 5 – essa variável tem o valor 5. Isso faz sentido, pois existem 5 navios no jogo. Parece que nos esquecemos de retornar o valor da variável this.currentUserShip para 0 quando reiniciamos o jogo. Note que this.currentUserShip é inicializada com 0 no construtor da classe Board. De fato, outras

inicializações de variáveis feitas aí devem também ocorrer quando reiniciamos o jogo.

Existem duas maneiras de se fazer isso. A primeira seria mover o código correspondente a essas inicializações, do construtor da classe Board para um método separado. (que poderia ser chamado 'Init'). Então poderíamos chamar esse método sempre que um novo jogo fosse reiniciado.

A segunda maneira á mais simples, mas menos eficiente – poderíamos simplesmente criar um novo tabuleiro quando reiniciamos o jogo. Apesar disso, vamos usar essa estratégia, para tornara as coisas mais simples.
Como estamos reiniciando um novo jogo na seção GAMEOVER do método mouseReleased (da classe BattleshipApp), vamos então modificar o código desse mátodo:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        // Um *bocado* de código entra aqui.
   }
    else if (this.gameState == GAME_STATE_GAMEOVER) {
        // Inicializa os tabuleiros do jogo.
         this.redBoard = new Board(PLAYER_STATUS_X, STATUS_Y, Color.red);;
         this.blueBoard = new Board(COMPUTER_STATUS_X, STATUS_Y, Color.blue);
       System.gc();

        this.gameState = GAME_STATE_SETUP;
       this.repaint();
   }
}
```

Note a chamada de método 'System.gc()' depois de serem criados os tabuleiros do jogo. Isso sugere ao sistema de execução de programas Java que chame o coletor de lixo, para liberar a área de memória associada aos antigos tabuleiros.

Note também que chamamos os método repaint, para garantir que os novos tabuleiros sejam exibidos na janela do jogo. Desse modo, os jogadores reiniciam com novos tabuleiros limpos, na fase seguinte de SETUP.

Compile e execute o seu programa, mais uma vez, e brinque um pouco para ver se você consegue reiniciar um novo jogo.

Quase lá...

Como você pode ver, nosso programa está funcionando melhor, mas nem tudo está correto ainda. Quando terminamos o primeiro jogo e iniciamos um segundo, os tabuleiros são limpos e podemos posicionar os navios novamente. Entretanto, parece que os navios inimigos não são exibidos corretamente, e fica fácil vencer o segundo jogo. O que está acontecendo?
Quando estamos procurando encontrar um erro em um programa, é uma boa idéia usar todas as dicas que tivermos á mão. Nesse caso, nossa primeira dica é óbvia – os navios do inimigo (o computador) não são exibidos na janela de estado do jogo. A segunda dica é também importante – o usuário vence facilmente com o primeiro tiro! Se verificarmos, no código da classe BattleshipApp, onde ocorre a mensagem "Você venceu!", veremos o seguinte:

```
if (!this.blueBoard.isAnyShipLeft()) {
    this.setMessage("Você venceu! Clique o mouse para iniciar um novo jogo.", true);
```

Isso parece indicar que não existem navios no tabuleiro azul, quando o jogo é reiniciado! Isso está coerente com o que observamos na janela de estado do jogo – nenhum navio inimigo é mostrado na janela de estado. Portanto, se pudermos identificar qual é o trecho de código que posiciona os navios inimigos no tabuleiro, ficará fácil solucionar o problema existente no nosso programa.
Você ainda deve lembrar-se do método 'placeComputerShips' da classe BattleshipApp. Esse método é chamado na seção correspondente ao estado AME_SETUP do método mouseReleased. Reveja o seguinte trecho de código:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        this.gameState = GAME_STATE_SETUP;
         this.blueBoard.placeComputerShips();
       this.repaint();
       this.initializeSetupState();
   }
   // Mais código entra aqui...
}
```

Como você já deve ter adivinhado, devemos também incluir uma chamada do método 'placeComputerShips' na seção correspondente ao estado GAMEOVER, no método mouseReleased. Além disso, também temos que incluir uma chamada do método 'initializeSetupState' (reveja o código deste método para ver o que ele faz). Aqui está o novo código para o estado GAMEOVER do método mouseReleased:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
        // Um *bocado* de código entra aqui.
   }
    else if (this.gameState == GAME_STATE_GAMEOVER) {
        // Inicializa os tabuleiros do jogo.
         this.redBoard = new Board(PLAYER_STATUS_X, STATUS_Y, Color.red);;
         this.blueBoard = new Board(COMPUTER_STATUS_X, STATUS_Y, Color.blue);
       System.gc();

        this.gameState = GAME_STATE_SETUP;
         this.blueBoard.placeComputerShips();
       this.repaint();
       this.initializeSetupState();
   }
}
```

Se você agora compilar e executar novamente o programa, verá que pode jogar várias partidas. Você poderá pensar que tudo está funcionando corretamente agora – mas, depois de jogar algumas partidas, perceberá um problema... a atuação do seu adversário (o computador) piora muito, depois da primeira partida. Qual será o problema?
Bem, esse problema não é difícil de identificar. Sabemos onde reside o código correspondnete à "inteligência" do computador – na classe EnemyPlayer. Dando uma rápida olhada no construtor dessa classe, vemos o seguinte:

```
public  EnemyPlayer()
{
   this.bag = new Vector();

    this.aiGrid = new int[Board.ROW_COUNT][Board.COLUMN_COUNT];
   this.initAIgrid();

  int iRow = 0;
   while (iRow < Board.ROW_COUNT) {

     int iCol = 0;
      while (iCol < Board.COLUMN_COUNT) {
         Target boardPiece = new Target(iRow, iCol);
         this.bag.addElement(boardPiece);
        iCol++;
     }
     iRow++;
  }
}
```

Note que o método 'initAIgrid' apenas é chamado no corpo do construtor dessa classe. Esse m´todo inicializa os valores dos "pesos" usados para determinar as posições alvo de tiro. Como esse métodos apenas é chamado no construtor da classe, ele é executado uma única vez, quando o jogador inimigo é criado. Isso significa que, em uma segunda partida, os "pesos" usados para selecionar as posições alvo de tiro permanecem os mesmos da partida anterior. Isso não é uma boa estratégia...

Assim como no caso dos tabuleiros, precisamos "limpar" os dados do jogador inimigo ao final de cada partida. Portanto, ou temos que mover o código de inicialização desses dados para um método separado e chamar esse método sempre que quisermos inicializar os dados do jogador, ou temos que criar um novo jogador inimigo ao reiniciarmos uma nova partida. Mais uma vez, vamos adotar essa última solução.
Note que o seguinte trecho de código está contido no corpo do construtor da classe EnemyPlayer:

```
int iRow = 0;
while (iRow < Board.ROW_COUNT) {

   int iCol = 0;
    while (iCol < Board.COLUMN_COUNT) {
       Target boardPiece = new Target(iRow, iCol);
        this.bag.addElement(boardPiece);

      iCol++;
   }
   iRow++;
}
```

Esse trecho de código de fato pertencia ao nosso método original para "escolher aleatoriamente uma posição alvo de tiro". Esse código não tem nada a ver com a nova (e esperta) inteligência que programamos para o jogador inimigo. Portanto, vamos remover esse trecho de código, antes de corrigir o erro relacionado à criação de um novo EnemyPlayer. Removendo esse trecho de código, o construtor da classe fica assim:

```
public EnemyPlayer()
{
   this.bag = new Vector();

    this.aiGrid = new int[Board.ROW_COUNT][Board.COLUMN_COUNT];
   this.initAIgrid();
}
```

Oba! Bem menor!
Agora, vamos criar um novo jogador inimigo, quando é iniciada uma nova partida:

```
public void mouseReleased(MouseEvent event) {
    if (this.gameState == GAME_STATE_INTRO) {
       // Um *bocado* de código entra aqui.
  }
    else if (this.gameState == GAME_STATE_GAMEOVER) {
       // Inicializa os tabuleiros do jogo.
        this.redBoard = new Board(PLAYER_STATUS_X, STATUS_Y, Color.red);
        this.blueBoard = new Board(COMPUTER_STATUS_X, STATUS_Y, Color.blue);
       this.enemyPlayer = new EnemyPlayer();

      System.gc();
       this.gameState = GAME_STATE_SETUP;
        this.blueBoard.placeComputerShips();
      this.repaint();
      this.initializeSetupState();
   }
}
```

Mais uma vez, compile e execute o seu programa. Dessa vez, tudo deve funcionar corretamente. Se você tiver algum problema, confira ao seu código com as nossas versões das classes BattleshipApp (código 26a) e EnemyPlayer (código 26b), na seção de códigos incluída no final deste tutorial.

Você pode estar pensando: "Argh! Não aprendi muita coisa nesta seção. Tudo o que fiz foi procurar e corrigir erros". Você tem razão, isso pode não ser muito divertido, mas é uma tarefa muito freqüente no desenvolvimento de programas. É claro que tudo seria mais fácil se tivéssemos planejado melhor o nosso código, previamente. Se tivéssemos planejado que o jogo deveria permitir iniciar uma nova partida, não estaríamos agora procurando e corrigindo erros. Esse é também um padrão freqüente em programação – um bom projeto de sistema evita grande perda de tempo com depuração do programa.

Bem, concluímos nosso tutorial e nossa tarefa. Talvez você queira rever o que aprendemos e dar uma olhada em algumas dicas de como melhorar ainda mais o seu programa. Veja então nossas conclusões.

Conclusão

Finalmente concluímos a implementação do nosso jogo. Ao longo dessa tarefa, você teve oportunidade de ter uma visão geral sobre conceitos importantes de programação orientada a objetos: classes, objetos, métodos, variáveis de classe e variáveis de instância, herança, sobrecarga, associação dinâmica de métodos...

Você também teve oportunidade de lidar com conceitos comuns a todas as linguagens de programação imperativa: variáveis, expressões, comandos de atribuição, comandos de controle do fluxo de execução de programas – comandos condicionais e comandos de repetição –, tipos de dados – tipos primitivos, como os de valores inteiros, booleanos e caracteres, e tipos estruturados, como arrays e vectors.

Talvez você ainda não se sinta seguro em relação a todos esses novos conceitos. Isso é natural, afinal, são muitos conceitos, vistos pela primeira vez. Se você gostou de programar, ainda há muito a aprender e praticar. Para saber mais sobre a linguagem Java, consulte as referências indicadas no apêndice "Sobre Java". Se você quer aprender mais sobre implementação de jogos, consulte as referências indicadas no apêndice "Sobre Jogos".

Bem, agora vamos deixar você brincando – com o jogo de Batalha Naval e também com o código do programa... Você fez um excelente trabalho!

Nos vemos de uma próxima vez.

Referência sobre Java

Sites na Internet:

- http://java.sun.com .download do sistema de desenvolvimento de programas Java, informações, tutoriais, etc.
- http://java.sun.com/j2se/1.4.1/docs/api referências para bibliotecas de classes do ambiente Java 2 SE versão 1.4.1.
- http://www.java.sun.com/docs/books/tutorial tutorial sobre Java
- http://java.sun.com/docs/books/tutorial/uiswing/ tutorial sobre como criar GUIs usando AWT/Swing
- http://www.mindview.net/Books documentação sobre a linguagem

Ambientes integrados de programação em Java:

http://www.blueJ.org

htttp://www.eclipse.org

htttp://www.jcreator.com

htttp://www.jbuilder.com

Livros

Existem milhares de livros sobre a linguagem Java. Relacionamos abaixo apenas alguns deles.

- Programação de Computadores em Java,

Carlos Camarão e Lucília Figueiredo. Editora LTC, 2003.

Livro didático de programação de computadores em Java, para cursos introdutórios de programação, de nível universitário.

- The Java Language Specification

James Gosling, Bill Joy, Guy Steele Jr., Gilad Bracha. Addison-Wesley, 1996.

Definição original da linguagem Java. Livro de referência sobre a sintaxe e semântica da linguagem. Não é um livro voltado para introdução ao ensino da linguagem, nem para o ensino de programação de computadores. O livro está disponível, tanto em HTML quanto PDF e PostScript, na página http://java.sun.com/docs/books/jls/.

- The Java Programming Language

Ken Arnold, James Gosling, David Holmes. Addison-Wesley, 2000 (3a. edição).

Introdução à linguagem Java, escrita pelos projetistas da linguagem.

- Java: How to Program

Livro bastante extenso e abrangente, incluindo: introdução à programação orientada a objetos, ao uso de bibliotecas da linguagem Java – como bibliotecas para interface gráfica (Swing e AWT) e bibliotecas para definição e uso de estruturas de dados (Collection) – introdução à programação concorrente em Java, à programação de aplicações em redes de computadores, ao desenvolvimento de aplicações de multimídia, ao uso de bancos de dados usando, etc. Traduzido para o português.

- Core Java 2: Fundamentals (vol. 1), Advanced Features (vol. 2)

Gary Cornell, Cay S. Horstmann, Prentice-Hall, 2002 (6a. edição).

Bons livros de referência sobre Java, que cobrem extensivamente a programação na linguagem (o primeiro contém 752 páginas e o segundo 1232 páginas). Traduzido para o português.

- Thinking in Java

Bruce Eckel, Prentics-Hall, 2002 (3a. edição).

Livro gratuitamente disponível na Web a partir do endereço http://www.mindview.net/Books/ , aborda diversos aspectos da programação em Java e inclui comparação de Java com outras linguagens, particularmente com C++.

- Java Professional Programming

Brett Spell. Wrox Press Ltda. (2a. edição), 2000.

Livro de programaçcão avançada em Java, abordando diversas bibliotecas da linguagem.

- Livros - Introdução à Programação Orientada a Objetos Usando Java

RAFAEL SANTOS. Editora Campus

- Livros - Programação Orientada a Objetos com Java

DAVID J. BARNES & MICHAEL KOLLING. Makron Books

Algumas Referências sobre Jogos em Java na Internet

http://www.sleepinggiantsoftware.com/FGJ/games.htm

http://www.kidsdomain.com/games/java.html

http://www.addictinggames.com/

http://www.free-game.com/Java_Games/

TUTORIAL

BENVINDO!

Você provavelmente já deve ter tido oportunidade de usar diversos tipos de programas em seu computador: programas que implementam jogos, editores de texto, programas para envio mensagens eletrônicas através da Internet, navegadores web... Como um computador pode ser capaz de executar tantas tarefas, tão diferentes? O segredo é que um computador é uma máquina bastante versátil: pode ser programado para executar um grande número de tarefas, de vários tipos!

Mas o que é, afinal, um programa de computador? Como podemos escrever um programa? Como um computador é capaz de entender e executar um programa? Neste tutorial, procuramos responder a essas perguntas, de maneira simples e introdutória, e também introduzir alguns conceitos básicos de linguagens de programação modernas: classes, objetos, métodos, interfaces... Esperamos também que você possa aprender a utilizar esses conceitos no desenvolvimento de programas e que, ao final deste tutorial, já tenha sido capaz de desenvolver seu primeiro projeto de programação.

Vamos utilizar, neste curso, a linguagem de programação Java. Java é uma linguagem orientada a objetos, desenvolvida pela Sun Microsystems, e hoje empregada no desenvolvimento de programas para inúmeras aplicações, especialmente aquelas voltadas para a rede mundial de computadores – a Internet. O ambiente de programação da linguagem Java inclui compiladores, interpretadores e um rico conjunto de bibliotecas de classes, além de outras ferramentas úteis para o desenvolvimento de programas. Para auxiliar em nossas tarefas de programação, vamos também utilizar um ambiente integrado de desenvolvimento de programas – o BlueJ.

Puxa, quantas palavras novas e complicadas! Não se assuste. Ao longo deste tutorial esperamos poder esclarecer o significado de cada uma delas. Para isso, além de textos explicativos, utilizaremos vários exemplos de programas e proporemos que você desenvolva algumas atividades de programação. Os exemplos e atividades propostas consistem, em sua maioria, de partes de um programa que implementa um jogo de "Batalha Naval" (*), o qual você irá aprender a desenvolver, passo a passo. Ao final, além de ter aprendido bastante, você terá um novo jogo em seu computador! Legal, não?

Antes de começar, talvez você queira ter uma idéia de como está organizado o conteúdo deste tutorial:

Módulo 1: Vamos começar procurando entender os conceitos de programa, linguagem de programação, edição, compilação, depuração e execução de programas. Você será instruído sobre como instalar, em seu computador, o ambiente de programação Java e o ambiente integrado de desenvolvimento de programas BlueJ. Ao final, você já terá aprendido a editar, compilar e executar um primeiro programa.

Módulo 2: Neste módulo você vai começar a criar seus primeiros programas, aprendendo um pouco sobre os conceitos fundamentais de linguagens de programação orientada a objetos e, em particular, sobre a linguagem Java. Você vai saber sobre os diferentes tipos de programa que podemos construir em Java: aplicações console, aplicações stand alone com interface gráfica e applets.

Módulo 3: Neste módulo você vai começar a programar o arcabouço do seu jogo de Batalha Naval. Vai aprender sobre os tipos primitivos da linguagem Java, sobre o comando de atribuição e comandos de repetição, em Java.

Módulo 4: Neste módulo você vai começar a programar a interface gráfica do seu jogo de Batalha Naval. Para isso, você precisará aprender um pouco sobre o conceito de herança em linguagens orientadas a objetos e sobre como organizar o seu programa em classes. Você precisará aprender também sobre a idéia de programação dirigida por eventos.

Módulo 5: Neste módulo, vamos ver como tratar, em um programa, eventos relacionados ao mouse – tais como o evento do usuário clicar o mouse sobre a janela do programa. Você vai aprender como escrever o código para fazer o seu programa reagir a tais eventos. Assim, poderá programar no seu jogo, por exemplo, o evento de o usuário atirar sobre uma posição do tabuleiro do jogo, procurando acertar o tiro em um navio do adversário.

Módulo 6: Este módulo trata da interface gráfica que exibe os tabuleiros onde são posicionados os navios dos dois jogadores. Para programar essa interface, você irá aprender sobre arrays de duas dimensões e um pouco mais sobre comandos de repetição.

Módulo 7: A partir deste módulo, vamos desenvolver o código que possibilita aos dois jogadores – o computador e o usuário – posicionarem suas esquadras no tabuleiro do jogo. Os dois casos são bastante diferentes. No caso do usuário, teremos que tratar eventos do mouse, de maneira que o usuário possa selecionar as células do tabuleiro onde deseja posicionar seus navios. Você irá também aprender a exibir mensagens de instrução na janela do jogo.

Módulo 8: Neste módulo vamos desenvolver para posicionar no tabuleiro os navios do computador. Nesse caso, as posições dos navios serão escolhidas aleatoriamente – você precisará aprender sobre como lidar com números aleatórios, em Java.

Módulo 9: Este módulo trata do código para exibir, na tela do jogo, uma janela que informa sobre o estado dos navios dos dois jogadores, isto é, quais os navios já foram atingidos por tiros do jogador adversário. Além disso, vamos tratar do código que possibilita, a cada jogador, atirar sobre navios da esquadra adversária.

Módulo 10: Neste módulo, vamos fazer alguns acertos finais no nosso projeto, concluindo assim nossa tarefa de programação.

(*) Os exemplos utilizados no curso foram extraídos do tutorial disponível em http://www.sleepinggiantsoftware.com/FGJ/index.htm, no qual este tutorial foi baseado.