

JPlay

Jplay é uma framework (Possui um conjunto de classes que colaboram para realizar uma determinada tarefa) desenvolvida na UFRJ voltada para o desenvolvimento rápido e fácil de jogos. Contem vários métodos e objetos auxiliares que o ajudarão a criar jogos 2D usando a linguagem Java em pouquíssimo tempo.

Resumo

Nesta breve introdução, iremos falar sobre a configuração do ambiente de programação, alguns conceitos de POO e a função de métodos das classes utilizadas nos exemplos.

Programas utilizados

IDE Eclipse: [<http://www.eclipse.org/downloads/>]

Framework JPlay: [<http://www2.ic.uff.br/jplay/zip/jplay3.rar>].

Importando um projeto Eclipse

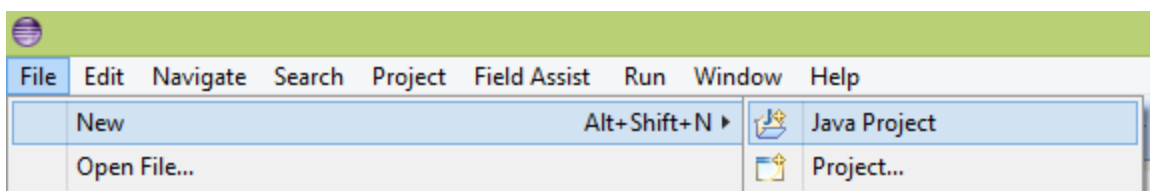
Caso já se tenha um projeto, não há necessidade de criar um do zero e configurar, basta importar. Para isso vamos seguir uma série de etapas bem definidas:

- No Eclipse, clique em **File > Import > Existing Project Into Workspace** e clique em **next**.
- Em **Select root directory** clique em "**Browse...**" e vá até a pasta onde se encontra o projeto desejado, selecione e dê **OK**.
- Na área **Projects** irão ser listados todos os projetos que existem dentro da pasta que você selecionou na etapa anterior. Deixe marcado o ou os projetos desejados e clique em **finish**.

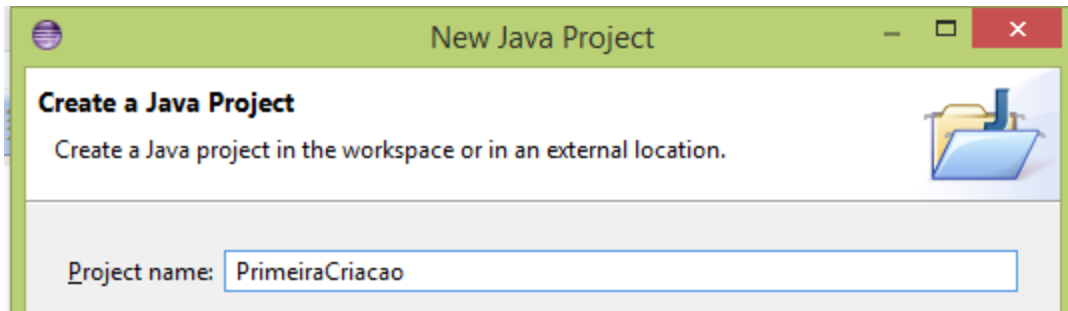
Se o projeto importado possuir o *JPlay* adicionado ao *Build Path*, pule os 2 próximos tópicos.

Criando um projeto Java

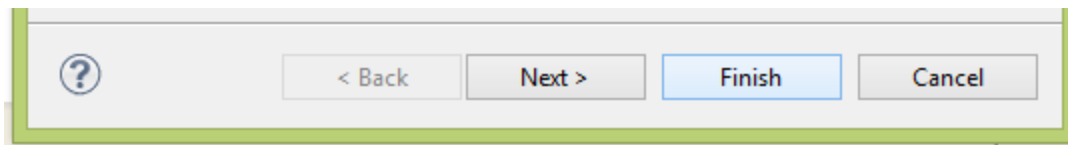
Primeiramente vá em **File > New > Java Project**.



Em *Project name* defina um nome para seu projeto, seguindo sempre este padrão: **PalavraComposta**, ou seja, sem símbolos, espaços, caracteres especiais, acentos e a primeira letra de cada palavra deve ser maiúscula.




Não é necessário alterar mais nenhuma configuração, sendo assim, clique em **Finish**.



Importando a biblioteca do JPlay

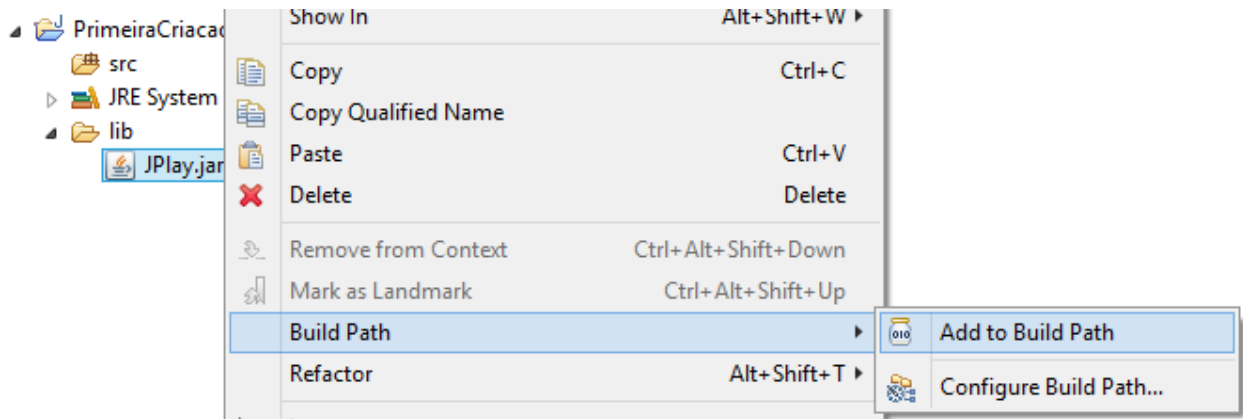
Criado o projeto e baixado a biblioteca do JPlay, agora nos resta importá-la. Depois de descompactado o arquivo, vá até a pasta **JPlay\store** e copie o arquivo JPlay.jar.

Nome	Tipo	Tamanho	Pasta
 JPlay	Executable Jar File	184 KB	store (D:\Downloads\Compressed\JPlay)

Vá até a pasta de seu projeto, que fica dentro do seu *workspace*, crie 2 novas pastas: Uma chamada **lib**, que é abreviação de *library*, é aqui onde iremos colocar nossa biblioteca. A outra chamada **resources**, onde iremos colocar todos os recursos do jogo: imagens, sons...



Com isto feito, vá até o Eclipse, abra seu **projeto > lib** clique com o botão direito em cima da biblioteca e adicione ao projeto. **Build Path > Add to Build Path**.



Pronto, com isto, agora já estamos com nosso ambiente de programação configurado.

Alguns conceitos de POO

Alguns conhecimentos básicos de POO que serão bastante utilizados, tais como: pacotes, variáveis locais e globais, herança, bem como as palavras reservadas *this* e *super*.

Package - Package ou pacotes, nada mais são do que simples pastas com o intuito de organizar nosso projeto. De forma resumida, o uso de pacotes considera que:

- Em um pacote podem existir várias classes;
- Evitam-se conflito de nomes;
- Utilizam-se a notação de pontos em vez de “\” ou “/”.

Ex: o pacote **jplay.Window** é na verdade um diretório **\jplay\Window**.

Pacotes são definidos usando-se a palavra reservada *package* no início do código fonte:

package <nomePacote>;

ou

package <nomePacote>.<nomeSubpacote>;

Ex: **package br.ifpi.poo.jplay.spaceInvader;**

Importando pacotes - Classes de diferentes pacotes não são visíveis entre si e precisam ser importadas; Usa-se a palavra reservada *import*.

Ex: **import jplay.Window;**

package x imports - Em uma classe, deve-se: primeiro declarar a que pacote a classe pertence e posteriormente declarar as importações.

Ex:

```

1 package spaceInvaders;
2
3 import jplay.GameImage;
4 import jplay.Window;

```

Variáveis Globais ou atributos - são aquelas declaradas fora de métodos e vistas em toda a classe e existem enquanto o programa estiver rodando.

Variáveis Locais - são aquelas declaradas dentro de métodos e vistas apenas dentro do método. Permanecem “vivas” apenas enquanto o método é executado.

Um problema comum gerado é quando utilizamos variáveis ou parâmetros com o mesmo nome dos atributos é que o programa não sabe identificar qual você está querendo referenciar.

Perceba que nosso saldo não será alterado, mesmo que o método *definirSaldoInicial()* seja chamado.

```

2 public class Conta {
3     double saldo;
4
5     void definirSaldoInicial(double saldo) {
6         saldo = saldo;
7     }
8 }

```

Uma solução para este problema é a utilização do **this**.

This - *this* é uma palavra reservada que faz referência à classe onde a mesma está inserida, sem ser necessária a instanciação desta classe internamente. Isso é ilustrado no exemplo a seguir:

```

2 public class Conta {
3     double saldo;
4
5     void definirSaldoInicial(double saldo) {
6         this.saldo = saldo;
7     }
8 }

```

Herança - Utilizamos herança para compartilhar atributos e métodos com o objetivo de reaproveitar código e comportamentos. Para isso, usa-se a palavra reservada *extends*. Exemplo:

```

public class Nave extends Sprite {

```

Neste exemplo a classe **Nave** está herdando **Sprite**, agora **Nave** possui o direito de utilizar de todos os métodos e atributos públicos da classe **Sprite**. Dizemos que **Sprite** é mãe de **Nave**.

Sobrescrita de métodos - Quando criamos métodos com o mesmo nome de um método da super classe.

Super - A palavra super, refere-se a super classe. Exemplo:

```
5 public class Nave extends Sprite {  
6  
7     public Nave(String endereco) {  
8         super(endereco);  
9     }  
}
```

No exemplo acima, dentro do construtor da classe Nave, estamos chamando o construtor da classe **Sprite** e passando o parâmetro *endereco*.

Caso queria chamar um método diferente do construtor basta: *super.metodo(parâmetro)*;

Padrão de desenvolvimento

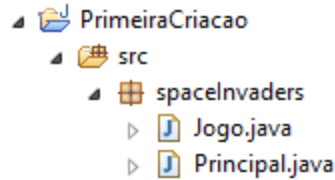
Iremos a seguir apresentar alguns padrões de desenvolvimento com o objetivo de tornar nosso código mais legível, otimizado e facilitando a manutenção e a correção de possíveis erros.

Algumas práticas:

- Possuir um pacote para cada jogo criado;
- Escrita de comentários ao longo do código;
- Métodos e variáveis devem ser iniciadas com letra minúscula, e a partir da segunda palavra iniciar com letra maiúscula. exp: **variavel, metodoComMaisDeUmaPalavra()**;
- Classes devem ser escritas com letras iniciais maiúsculas, seguindo o mesmo padrão do nome do projeto.

Ciclo padrão de desenvolvimento:

- Possuir uma classe Principal, neste será chamado o construtor da classe que contém o comportamento do jogo;
- Possuir uma classe Jogo onde irá conter o que acontece no jogo. Onde são instanciados, inicializados todos os objetos usados no jogo e aplicados todos os métodos que irão definir o comportamento do jogo.
- Para cada personagem e objeto do jogo deve ser criado uma classe.



- A classe que possui o comportamento do jogo merece uma observação mais detalhada. Quando formos implementá-la, vamos seguir um padrão que inclui os métodos seguintes métodos:

```
1 package spaceInvaders; //nome do pacote
2
3 public class Jogo {
4
5     // Aqui declaramos os atributos da classe jogo
6
7     public Jogo() { // Este eh o Construtor da classe
8
9         init();
10        loop();
11    }
12
13    public void loop() {
14        // O loop e' o coração do jogo, e' nele que serao colocadas todas as
15        // condicoes de interacoes e as atualizacoes das imagens na tela.
16        // O que ocorre no jogo, esta aqui!
17    }
18
19    public void init() {
20        // Este metodo eh responsavel por inicializar ou instanciar todos os
21        // nossos atributos
22    }
23
24    public void desenha() {
25        // Metodo responsavel por desenhar em tela todos os elementos
26    }
27 }
```

Este é o esqueleto de nossa classe Jogo.

Algumas funcionalidades do JPlay

Classe **Window**. Os métodos que iremos utilizar são os seguintes:

public Window(int largura, int altura) – construtor da classe.

public Keyboard getKeyboard() – retorna uma instância do teclado.

public void update() – mostra a janela atualizada com os desenhos na tela do monitor.

public void delay(long tempo) – O jogo retarda/atrasa pelo tempo passado por parâmetro (em milissegundos), usado principalmente, para atrasar a execução do jogo.

public void drawText(**String** message, **int** x, **int** y, **Color** color) – desenha um texto na tela.
public void exit() – fecha a janela e sai do jogo.

Classe **Sprite**. Possui métodos que podem fazer a imagem se locomover pela tela e de animação. Os métodos que iremos utilizar são os seguintes:

public Sprite(**String** fileName, **int** numFrames) - é passado o endereço da imagem que será exibida e número de frames que ela contém.

public Sprite(**String** fileName) - basta passar o endereço da imagem a ser exibida, o número de frames automaticamente será 1.

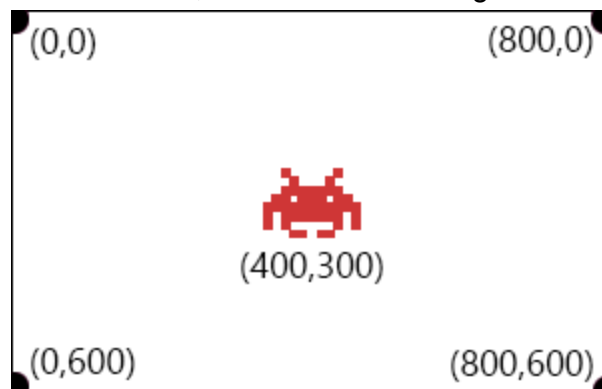
public void moveX(**double** velocidade) – Move o Sprite na tela somente no eixo x. Utiliza o teclado. Por padrão as teclas direcionais.

public void moveY(**double** velocidade) – Move o Sprite na tela somente no eixo y. Utiliza o teclado. Por padrão as teclas direcionais.

Posicionando um Sprite na tela

Para posicionarmos um desenho na tela, primeiramente precisamos conhecer como funciona o posicionamento de objetos pela tela.

Primeiramente, imagine a tela como um plano cartesiano, com 2 eixos: o x para as abscissas e o eixo y para as ordenadas. A única diferença para nosso plano cartesiano é que a posição **(0,0)** fica no canto superior esquerdo, e a medida que o objeto vai se movendo para esquerda ou para baixo, este valor vai aumentando, como ilustrado a seguir:



A posição **x** do personagem acima é 400 e sua posição **y** é 300.

Criando uma janela e definindo um fundo

Não esquecendo das importações de classe, vamos criar 2 atributos:

- janela, do tipo Window. Ou **Window** *janela*;
- fundo, do tipo GamelImage. Ou **GamelImage** *fundo*.


Vamos agora inicializá-los e desenhá-los conforme ilustrado abaixo:

```

16 public void loop() {
17     while(true){
18         desenha();
19
20         janela.update();
21     }
22 }
23
24 public void init() {
25     janela = new Window(800, 600);
26     fundo = new GameImage("resources/fundo.png");
27 }
28
29 public void desenha() {
30     fundo.draw();
31 }
32 }

```

É importante observar o uso de um método desconhecido chamado **update()**. Esse método da classe *Window* é usado para mostrar as atualizações feitas e sempre deve ser chamado por último.

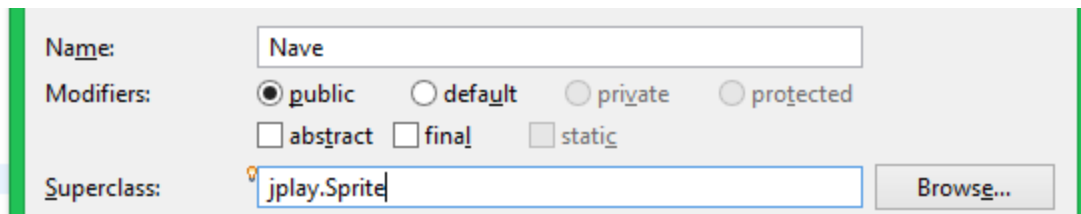
Para rodar nosso jogo basta apertar **CTRL + F11** ou .

Criando nossos personagens


Como dito anteriormente, uma classe deve ser criada para cada personagem e objeto, mas quais são nossos personagens? Como no nosso exemplo tratamos do clássico jogo *Space Invaders*, sabemos que ele possui uma nave e os inimigos/invasores.

Mas além de criar 2 classes representando a nave e o inimigo, vamos criar mais uma para representar o tiro da nave, todas herdando a classe **Sprite**.

Clique com o botão direito em cima do **pacote > new > class**. Em name, insira o nome da classe e em Superclass, insira o endereço de onde está a super classe, assim ele irá importá-la automaticamente.





Para finalizar, clique em *Finish*.

Nossa classe foi criada, mas note que o eclipse aponta um erro, cliquei na notificação que fica na mesma linha do erro (). Ele irá sugerir possíveis soluções.


```

1 package spaceInvaders;
2
3 import jplay.Sprite;
4
5 public class Nave extends Sprite {
6
7 }
8

```

 Add constructor 'Nave(String)'
 Add constructor 'Nave(String,int)'

Este erro foi ocasionado porque todas as classes que herdam de **Sprite** são obrigadas a criar um construtor padrão que passe por parâmetro uma String.

```

1 package spaceInvaders;
2
3 import jplay.Sprite;
4
5 public class Nave extends Sprite {
6
7     public Nave(String fileName) {
8         super(fileName);
9         // TODO Auto-generated constructor stub
10    }
11 }

```

Feito isto, vamos agora fazer o mesmo para os personagens **Inimigo** e **Bala**.

Criando uma pontuação e um contador de tiros

Antes de fazermos a bala e sua colisão, vamos

O que precisamos saber sobre colisões?

Bem, para descobrirmos se um objeto colidiu com outro você pode usar o método: **boolean collided(GameObject)**.

O método **boolean collided(GameObject)**, retorna *true* se houver colisão, ao contrário, retorna *false*. Exemplo:

```

if(bala.collided(inimigo)){
    //Colidiram !!!
}

```

Criando uma bala e fazendo sua colisão

Criado a classe bala e seu construtor, vamos criar um objeto do tipo bala na classe Jogo, inicializar ela e desenhá-la na tela. Assim como a gente fez com os personagens.

Porem, não é necessário inicializar as coordenadas da bala (x,y) pois os mesmos dependem da posição da nave.

Criando os métodos da classe Bala: Precisaremos que a bala suba a tela, que ela receba a posição X inicial