

# Orientação a Objetos

Carinhosamente chamada OO

2

# Revisão OO mais Jogos

- Passamos rapidamente por uma grande quantidade de conceitos nas últimas 8 aulas;
- Nesta aula iremos revisá-los, um a um, para perceber a dimensão do que estudamos e fortalecer o significado de cada novo termo.

# Agenda

- Revisão Conhecimentos da Disciplina;
- Conceitos de Jogos 2D;
- Conceitos de OO;
- Onde aprender mais?

## Ferramentas Base

- C16) Controle de Versões;
- C10) Interface de desenvolvimento (IDE);

## OO básica

- C1) Classes, objetos e instâncias;
- C4) Construtores;
- C2) Comentários;
- C9) Métodos e atributos;

## Elementos de Jogos 2D

- C17) Game Design
- C18) Sprites e TileMaps;
- C19) Animação e desenho 2D;
- C25) Captura Eventos Teclado;

## Reuso com OO

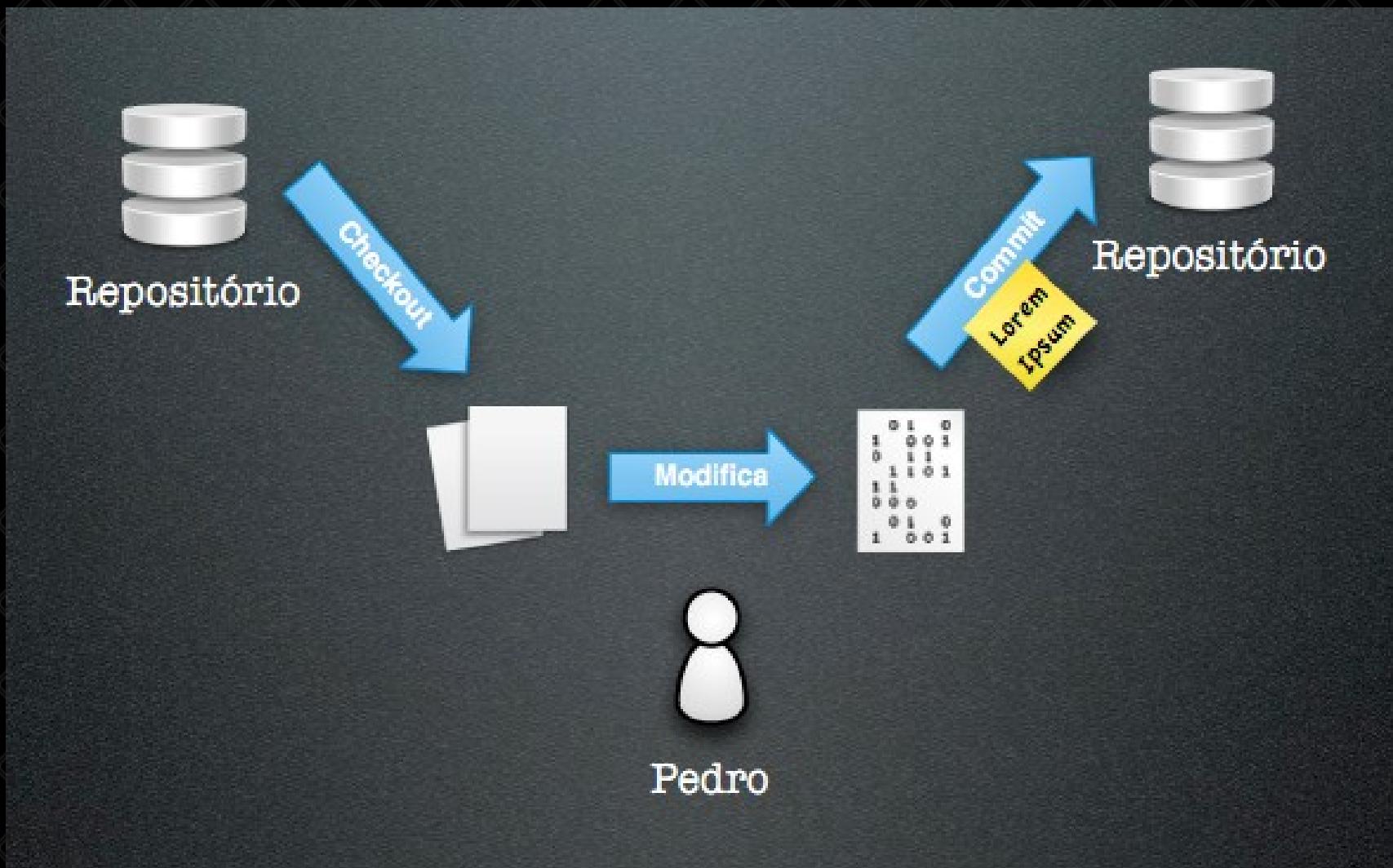
- C12) Padrões de desenvolvimento (frameworks);
- C5) Encapsulamento;
- C14) Reutilização de código;
- C6) Herança;
- C3) Composição;

## OO básica 2

- C7) Polimorfismo;
- C21) Classes abstratas,
- C22) Classes Interface;
- C23) Classes Enum;
- C24) Métodos e atributos estáticos;
- C20) Tratamento de Exceção;

# Controle de Versão

# Controle de Versão



# Controle de Versão

- **Resgate de versões antigas;**

Todos os commits e tags são registrados e é sempre possível voltar para o código de um commit ou tag anterior.

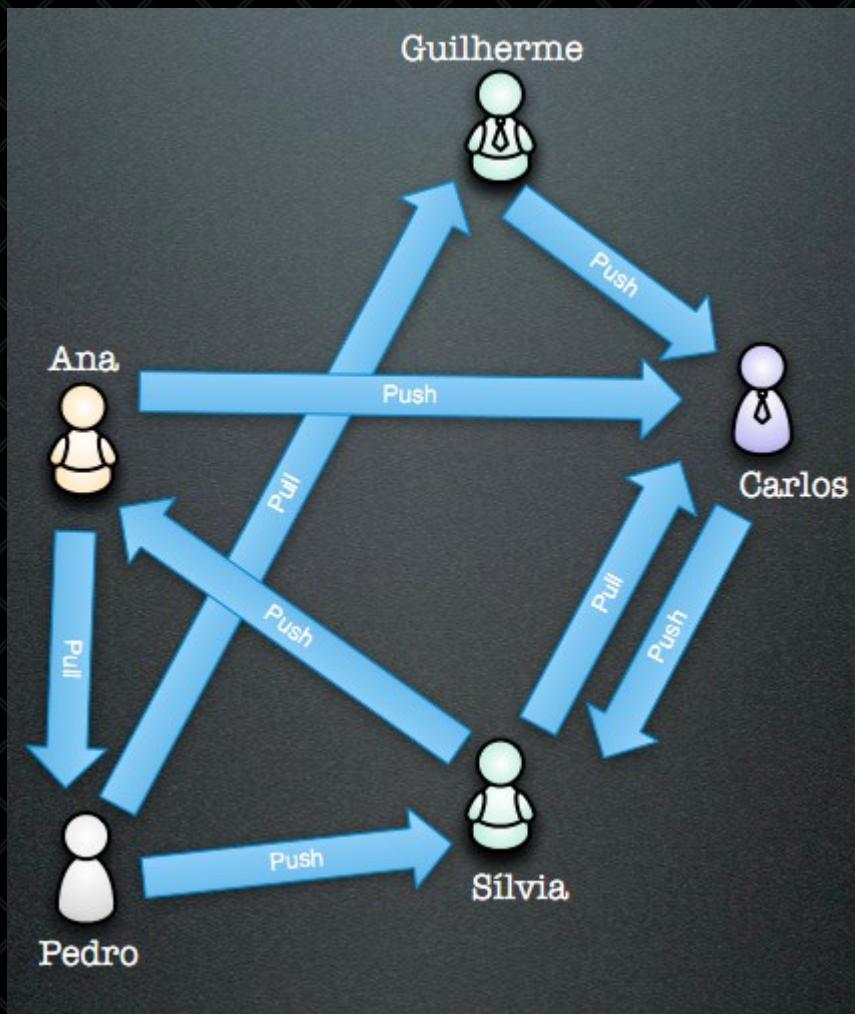
- **Trabalho em equipe;**

O código é compartilhado entre toda a equipe. A ferramenta automaticamente realiza "merge" nos arquivos para que todos tenham a versão mais atual. E quando duas pessoas alteram o mesmo código, é possível corrigir usando resolução de conflitos.

- **Registro de todo o Histórico de alterações;**

Se um erro ocorrer no software é possível ver quais as alterações feitas desde a última versão e analisá-las para encontrar o que aconteceu.

# Controle de Versões Distribuído



# Controle de Versões Distribuído

- Todos possuem o Histórico completo do projeto;

Basicamente, se o projeto estiver sendo usado por 5 PC's e 4 resolverem queimar o HD ao mesmo tempo, o projeto ainda estará a salvo.

- Commits podem ser feitos offline;

Ou seja, você pode trabalhar por um mês no projeto em um sítio em Paulo Lopes, e quando voltar, sincronizar todas as suas modificações com as realizadas pela equipe.

- Commits podem ser transferidos facilmente para outros repositórios;

Novas pessoas podem entrar no projeto rapidamente, acessar seus commits e "codar"

# Git

- **Controle de Versões Distribuído;**

Ou seja, possui todas as características anteriores;

- **Projeto Open Source;**

Isso significa que se os atuais responsáveis pelo projeto resolverem mudar de profissão, qualquer pessoa ou empresa poderá dando continuidade e suporte ao seu desenvolvimento;

- **Suporte para desenvolvimento não-linear;**

Isso significa que se você tiver trabalhando em uma nova funcionalidade do sistema e você precisar parar para corrigir um bug da versão anterior você pode facilmente: 1) salvar o que está fazendo em um branch; 2) acessar o versão que precisa ser corrigida; 3) corrigir o bug; 4) salvar a versão corrigida; 5) Deixar seus clientes felizes; e 6) voltar exatamente onde estava antes;

# Git e GitHub

- Amplamente utilizado pela comunidade de desenvolvimento de software;

Cada vez mais empresas e pessoas estão trabalhando com Git principalmente porque...

- Existem diversos repositórios online para você mostrar seu código para o mundo:

Isso permite que você crie um projeto nesta aula, publique ele no GitHub no final da tarde, seus amigos programadores o vejam e realizem algumas alterações bacanas e amanhã de tarde você pode seguir trabalhando já com as modificações feitas.

- Um destes repositórios é o **GitHub**

E ele é tão forte que algumas empresas além do currículo, pedem o GitHub dos desenvolvedores.

# IDE – Ambiente Integrado de Desenvolvimento

IDE

- Ambiente Integrado de Desenvolvimento;
- Desenvolvimento Rápido de Aplicativos;

# Ambiente Integrado (alguns recursos)

**Editor:** escrita rápida de código, auto-completar, geração de código, etc;

**Compilador:** Gera código de máquina a partir do fonte do projeto automaticamente;

**Depurador:** Facilita a busca por erros no código através da análise cuidadosa de cada passo executado pelo software;

**Testes Automatizados:** Execução de códigos para garantir integridade do software após modificações;

**Deploy:** Realiza a publicação da aplicação, geração de executáveis, etc.;

**Código Limpo e Otimizado:** facilita o entendimento do projeto por todos os envolvidos;

# Atalhos Selecionados (18)

Foco do Cursor		Edição de Código	
Ctrl + 0	Editor de Código	Ctrl + Shift + C	Editor de Código
Ctrl + 1	Janela Projetos	Ctrl + Shift + menos	Janela Projetos
Ctrl + 6	Navegação na Classe	Ctrl + Shift + baixo	Navegação na Classe
Ctrl + 7	Janela de Tarefas	Alt + Shift + F	Janela de Tarefas
Ctrl + Shift + 0	Resultados da Pesquisa	Ctrl + Q	Vai para última edição de código
Edição Mágica de Código		Geral	
Ctrl + R	Refatorar / Renomear	F6	Executa proj. principal
Alt + Insert	Gerar código	Ctrl + F12	Exibe janela para encontrar métodos e atributos de uma classe
Alt + F7	Encontra ocorrências de uma determinada variável ou classe.	Alt + Shift + Enter	Tela Cheia
Ctrl + Espaço	Exibe auto-completar.	Ctrl + Shift + F	Busca em Projetos

OO Básica

# Classes e Objetos

- Em OO, **modelamos o mundo (do nosso problema) utilizando classes**;
  - `public class Personagem { .. }`
  - `public class Tiro { .. }`
- **E criamos objetos (ou instâncias) destas classes que irão interagir para resolver o problema do nosso projeto:** ex:
  - `new Personagem( "Ryu", 100 );`
  - `new Personagem( "Vegeta", 100 );`
  - `new Tiro( 376, 480, Direcao.ESQUERDA );`

# Métodos e Atributos

- Quando criamos uma classe (ou seja, uma abstração do mundo real), **modelamos características e comportamentos** destas classes;
- **Características** são as nossas variáveis de instância;
  - `protected String nome;`
  - `protected int vida;`
- **Comportamentos** são os métodos que alteram ou acessam os valores das variáveis de instância, alteram ou acessam as características do objeto;
  - `public boolean temColisao( Personagem outroPersonagem);`
  - `public int getVida( );`
  - `public boolean estaVivo( );`
  - `public void perdeVida(int pontos);`

# Característica / Variável de Instância

`protected String nome;`



MODIFICADOR  
DE ACESSO



`protected int`



IDENTIFICADOR



`vida;`

# Comportamento / Método

public boolean temColisao(Personagem outro);



MODIFICADOR  
DE ACESSO



TIPO DE  
RETORNO



IDENTIFICADOR



PARÂMETROS



public void perdeVida( int pontos );

```
public class Personagem {  
    protected String nome;  
    protected int vida;  
    public void temColisao( Personagem outro ){  
        return this.getRetangulo().intersects(outro.getRetangulo());  
    }  
    public int getVida() {  
        return this.vida;  
    }  
    public boolean estaVivo( ){  
        return ( this.vida <= 0 );  
    }  
    public void perdeVida(int pontos){  
        this.vida -= pontos;  
    }  
}
```

# Construtores

- Um Construtor é o método executado quando um objeto é construído;
- Ou seja, toda vez que você usa o operador **new** o Java executa o construtor da classe que você está utilizando.
- O construtor sempre tem o mesmo nome da classe e nenhum tipo de retorno;
- Por exemplo, vamos ver o que acontece quando criamos o objeto:

```
Personagem ryu = new Personagem("Ryu", 100);
```

```
public class Personagem {  
    protected String nome;  
    protected int vida;  
  
    /**  
     * O java procurará o construtor com os mesmos parâmetros  
     * passados na inicialização do objeto.  
     * a função básica do construtor é inicializar as variáveis de  
     * instância da classe, para que o objeto seja válido.  
     * EX: se a vida não fosse inicializada, o método estaMorto()  
     * sempre retornaria verdadeiro.  
     */  
    public Personagem( String nomeP, int vidaP){  
        this.nome = nomeP;  
        this.vida = vidaP;  
    }  
  
    ...  
}
```

# Conceitos básicos de Jogos 2D

# O GameLoop

- Todo jogo ocorre dentro de um loop que realiza três operações básicas:
  - **Controle** (Eventos Teclado, Mouse, Rede, etc);
  - **Processamento** (Movimento, colisão, etc);
  - **Desenho**;

# No javaPlay...

- O loop é feito dentro da classe GameEngine;
- Para cada rodada do loop o javaPlay executa o método **step** e o método **draw** da fase ativa;
- O método **step** contém as etapas de **controle** e **processamento**;
- O método **draw** apenas desenha;

```
public class GameEngine {  
    // ...  
  
    public void run()  {  
        // ...  
        while(engineEstaRodando)  {  
            // ...  
            faseAtual.step( currentTime – lastTime );  
            faseAtual.draw( canvas.getGameGraphics() );  
  
            canvas.swapBuffers();  
            sleep();  
            // ...  
        }  
        // ...  
    }  
    // ...
```

# Sprite

Um único arquivo de imagem



Várias  
Fatias

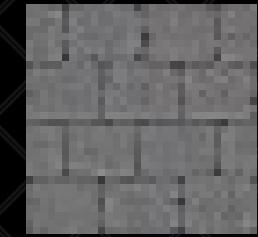
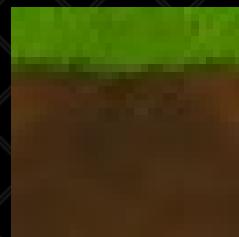
# no javaPlay...

- Criamos objetos da classe Sprite:
  - `new Sprite("resources/nave.png", 1, 64, 64);`
- O construtor de uma Sprite exige 4 parâmetros:
  - `public Sprite(String file, int frameInicial,  
larguraFrame, alturaFrame);`
- Alteramos o frame atual da Sprite quando desejarmos:
  - `this.sprite.setCurrAnimFrame( 6 );`
- E desenhamos a sprite:
  - `this.sprite.draw( g, this.x, this.y );`

# TileMap

- Grandes cenários exigiriam imagens muito grandes que poderiam tornar o carregamento do jogo lento;
- Além disso, é preciso ter uma forma de mapear quais pontos do cenário tem colisão com um jogador;
- A solução foi utilizar **pequenas imagens quadradas** para montar cenários através de um **arquivo de configuração**.

# As imagens pequenas, os tiles...



O Cenário...e o arquivo de configuração...



4 → número de Tiles

resources/img\_cenario/terra.png

resources/img\_cenario/grama.png

resources/img\_cenario/terra\_pedras.png

resources/img\_cenario/placa.png

1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1,0,0,1,1,1

1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1,1,0,0,1,1,1

1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,1,1,1,0,4,1,1,1

**1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,1,1,1,1**

% → O fim da matriz  
resources/img\_fundo.png → Uma imagem de fundo

# Animações

- Podemos fazer animações com 3 técnicas:
  - Mudando a Sprite ou imagem desenhada a cada determinado **número de frames**;
  - Mudando a Sprite ou imagem desenhada a cada **fatia de tempo** passada;
  - Usando **Gifs**;

# Exemplo de animação baseada em tempo.

```
public class Explosao extends GameObject {  
    protected Sprite sprite;  
    private int frame;  
    private int timeElapsedInMilliseconds;  
  
    public Explosao(int x, int y){ ... }  
  
    public void step(long timeElapsed) {  
        if(this.frame >= 16){  
            return; //Parou animação  
        }  
  
        this.timeElapsedInMilliseconds += timeElapsed;  
        if(this.timeElapsedInMilliseconds > 100){  
            this.frame++;  
            this.sprite.setCurrAnimFrame(this.frame);  
            this.timeElapsedInMilliseconds -= 100;  
        }  
    }  
  
    public void draw(Graphics g) {  
        this.sprite.draw(g, this.x, this.y);  
    }  
}
```

# Exemplo de animação baseada em frames.

```
public class Explosao extends GameObject {  
    protected Sprite sprite;  
    private int frame;  
    private int framesElapsed;  
  
    public Explosao(int x, int y){ ... }  
  
    public void step(long timeElapsed) {  
        if(this.frame >= 16){  
            return; //Parou animação  
        }  
  
        this.framesElapsed += 1;  
        if(this.framesElapsed > 10){  
            this.frame++;  
            this.sprite.setCurrAnimFrame(this.frame);  
            this.framesElapsed = 0;  
        }  
    }  
  
    public void draw(Graphics g) {  
        this.sprite.draw(g, this.x, this.y);  
    }  
}
```

# Exemplo de animação baseada em gifs.

```
public class Explosao extends GameObject {  
    protected Imagem img;  
    private int timeElapsed;  
  
    public Explosao(int x, int y){ ... }  
  
    public void step(long timeElapsed) {  
        this.timeElapsed += timeElapsed;  
    }  
  
    public void draw(Graphics g) {  
        if(this.timeElapsed < 1000){  
            //Se o gif tiver um segundo,  
            //ele sumirá e não será mais desenhado  
            //Se você quiser o gif em loop,  
            //basta apenas desenhá-lo a cada rodada  
            this.img.draw(g, this.x, this.y);  
        }  
    }  
}
```

# Desenho de formas geométricas

- O Java permite que desenhemos **círculos, quadrados e polígonos**;
- Fazemos isso utilizando os métodos do objeto **Graphics**;
- Este objeto não pode ser instanciado manualmente, porém **todo objeto Image ou JFrame** é composta por um objeto da classe **Graphics**;
- **No javaPlay**, é criada dinamicamente uma imagem e passados para nossos métodos draw uma referência do objeto **Graphics** desta imagem;
- Para saber mais estude a API Java 2D;

# Utilizando o objeto da classe Graphics

```
public void draw( Graphics g ) {  
    //configura cor  
    g.setColor( Color.blue );  
  
    //desenha círculo  
    g.fillOval(x, y, largura, altura);  
    //desenha Retângulo  
    g.fillRect(x, y, largura, altura);  
    //desenha Linha  
    g.drawLine(x1, y1, x2, y2);  
  
    // desenha polígono  
    Polygon triangulo = new Polygon();  
    triangulo.addPoint(this.x, this.y);  
    triangulo.addPoint(this.x+10, this.y+20);  
    triangulo.addPoint(this.x-10, this.y+20);  
    triangulo.addPoint(this.x, this.y);  
    g.fillPolygon(triangulo);  
}
```

# Captura de Eventos do Teclado

- Em Java, **eventos de teclado e mouse estão sempre ligados a janela em foco**;
- Para pegar eventos é preciso:
  - Criar uma classe que implementa a interface KeyListener;
  - Criar uma janela;
  - Associar a classe KeyListener com a janela;
  - Associar todos os objetos que tem algum comportamento alterado com o teclado à classe que implementa o KeyListener;

# no javaPlay existe a classe Keyboard

- Internamente a GameEngine associa esta classe com o GameCanvas(a janela do jogo);
- E para acessar a classe basta pegar a referência para o objeto ativo da classe Keyboard:

Keyboard k = GameEngine.getInstance().getKeyboard();

- E perguntar se uma determinada tecla está pressionada:

```
if( k.keyDown( Keys.DIREITA ) ) {  
    this.moveDireita(12);  
}
```

# Game Design

- O Game Design é o projeto do jogo. Afinal, o que será o jogo. O Game Desig é criado por um Game Designer:
  - Um Game Desiner proporciona uma **experiência**;
  - Um Game Designer toma **decisões** sobre o jogo;
  - Um Game Designer deve ter **múltiplas habilidades**;

As 4 Áreas de um Jogo

Mecânica Históri  
a  
Tecnologia Estética

# Mecânica

- **Regras:** o que pode e não pode acontecer no jogo;
- **Jogabilidade:** quais elementos podem ser adicionados ao jogo para permitir um maior número de possibilidades;
- **Diversão:** Equilíbrio entre descoberta e dificuldade;

# História

- Em qual **contexto** a mecânica será inserida para que **faça sentido na mente** do jogador?
- Desde o início dos tempos a principal forma de transmissão de experiências é através da tradição de **Contar Histórias**;
- Qual a história que **dará sentido à experiência**?
- Seres humanos gravam melhor as situações que envolvem histórias com **começo, meio e fim..**

# Tecnologia

- O que é preciso para fazer a mecânica do jogo acontecer?
- Colisão especial? Placa de vídeo? Reprodução de Vídeos? Fontes especiais? Bibliotecas externas?
- O jogo será 2D ou 3D?
- Quais as limitações tecnológicas?

# Estética

- **Estilo Visual:** o jogo terá estilo futurista ou como os primeiros vídeo-games?
- **Animações:** quais animações o jogo terá para torná-lo mais agradável?
- **Imagens** de personagens e cenários;

## Ferramentas Base

- C16) Controle de Versões;
- C10) Interface de desenvolvimento (IDE);

## OO básica

- C1) Classes, objetos e instâncias;
- C4) Construtores;
- C2) Comentários;
- C9) Métodos e atributos;

## Elementos de Jogos 2D

- C17) Game Design
- C18) Sprites e TileMaps;
- C19) Animação e desenho 2D;
- C25) Captura Eventos Teclado;

## Reuso com OO

- C12) Padrões de desenvolvimento (frameworks);
- C5) Encapsulamento;
- C14) Reutilização de código;
- C6) Herança;
- C3) Composição;

## OO básica 2

- C7) Polimorfismo;
- C21) Classes abstratas,
- C22) Classes Interface;
- C23) Classes Enum;
- C24) Métodos e atributos estáticos;
- C20) Tratamento de Exceção;

# Reuso em OO

# Frameworks

- Frameworks constituem uma abordagem de desenvolvimento que visa maximizar o reuso de software (Ricardo Pereira e Silva);
- Definição: "Conjunto de classes que incorpora um projeto abstrato que soluciona uma família de problemas relacionados". (Fayad, citado por Ricardo Pereira e Silva)
- JavaPlay:
  - Conjunto de 9 classes;
  - Onde cada fase é um GameStateController e cada objeto do jogo é um GameObject;
  - Para jogos 2D;

# Encapsulamento

- Encapsular é fundamental para que seu sistema seja sucatível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada (Caelum);
- Podemos encapsular uma determinada regra ou lógica dentro de um **método** se ela for pequena, dentro de uma **classe** se for mais complexa ou dentro de uma **API** ou **framework** se for ainda maior;

# Metáfora da Cápsula

- Encapsular pode significar literalmente "Colocar dentro de uma cápsula";
- A partir disso podemos pensar em um comprimido para alguma doença;
- Você sabe que ele é bom para aquela doença e então, pode tomá-lo para este fim sem saber o que há dentro da cápsula;
- Porém, o médico precisará saber exatamente o que estiver lá dentro para não correr riscos de lhe oferecer algo que irá prejudicá-lo depois.

# A metáfora na programação

- Se o seu objetivo for se tornar um excelente programador de jogos em Java, é fundamental que você estude por dentro o framework javaPlay e outros ao invés de apenas usá-lo;
- Se o seu objetivo é desenvolver jogos para diversas plataformas, será mais inteligente se preocupar em aprender os melhores frameworks de cada linguagem, sem se preocupar com a forma que ele está implementado;

# Exemplos de Encapsulamento

- (Encapsulamento de framework) **javaPlay**:
  - Framework que encapsula a lógica de construção de jogos 2D;
- (Encapsulamento de classe) **Classe CenarioComColisao**:
  - Encapsula a lógica de construção de um cenário baseado em Tiles e colisão com objetos;
- (Encapsulamento de método) **Método temColisao de um GameObject com um Retângulo**:
  - Encapsula a lógica de colisao entre um GameObject e um retângulo;

# Reuso com Herança e Composição

- Frameworks fornecem classes e uma arquitetura para você desenvolver sua solução;
- Encapsulamento é apenas um conceito que pode ajudá-lo a identificar quais regras podem ser encapsuladas;
- Para realizarmos **reuso em nível de código**, as duas formas básicas que temos são **Herança** e **Composição**;

# Herança

- A herança permite definir uma classe como uma extensão de outra (Barnes);
- Quando uma classe extende outra, dizemos que ela herda todos os métodos e atributos (desde que não sejam private) da classe superior;
- Podemos fazer isso em java através da palavra-chave extends:
  - `public class Bola extends GameObject { ... }`
  - `public class MacaVenenosa extends Item { ... }`

# Mau uso de Herança

- Durante anos, o uso de herança foi considerado como o grande mote da orientação a objetos. O mais importante na construção de um sistema era a definição de uma boa hierarquia de herança (Raul Sidnei Wazlawick)
- Alguns desenvolvedores usavam herança mesmo quando queriam apenas um método de uma determinada classe.
- Por exemplo, considere a classe Random que é a classe utilizada para gerar números aleatórios. E digamos que o seu sistema tem uma nave que precisa de números aleatórios para lançar tiros. Um erro grave seria:
  - `public Nave extends Random{ ... }`

# Herança = É um

- Uma forma de verificar se um caso é herança, é utilizar o termo "é um".
- Por exemplo. Considere uma Classe **Cachorro** com os atributos **corDoPelo** e **velocidadeCorrida**.
- Para criá-la eu poderia extender uma classe **Animal**. Este relacionamento é correto, **um cachorro é um animal**.
- Agora imagine que exista a necessidade de uma classe **Gato** que tenha também **corDoPelo** e **velocidadeCorrida**. Seria correto extender a classe Cachorro?
- Neste caso, não, porque **um gato não é um cachorro**;
- Se no futuro a classe Cachorro fosse modificada para ter o método latir. A classe gato automaticamente teria o mesmo método, estranho não?

# Composição ao invés de Herança

- Uma outra forma de **realizar reuso de classes** é utilizando composição;
- A **composição consiste** basicamente em uma classe ter uma variável de instância de outra classe;
- O temo chave para verificar se o seu sistema tem composição é verificando se um classe "tem um" objeto de outra classe.

# Exemplo de Composição

- Considere uma fase de um jogo de naves, podemos dizer que uma determinada fase tem um jogador, uma nave inimiga e tiros sendo lançados de um lado para o outro.
- Nesta relação a classe Fase é composta por objetos de três outras classes;
- Em código isso se reflete no seguinte:

```
public class Fase implements GameStateController {  
  
    //fase é composta pelo jogador  
    NaveJogador1 naveJogador1;  
    //fase é composta por tiros  
    ArrayList<Tiro> tirosJogador;  
    //fase é composta por uma naveInimiga  
    NaveInimiga naveInimiga;  
  
    public void load() {  
        //Inicializa os objetos que compõe a fase  
        this.naveJogador1 = new NaveJogador1();  
        this.tirosJogador = new ArrayList<Tiro>();  
  
        this.naveInimiga = new NaveInimiga();  
    }  
}
```

# Composição

- Fechando com o exemplo da Nave e da Classe Random.
- Podemos dizer que uma nave tem um gerador de números aleatórios;
- Em código isso poderia ser refletido assim:

```
public class Nave {  
    private Random geradorNumeros;  
    public Nave(){  
        this.geradorNumeros = new Random();  
    }  
}
```

# OO Básica Parte 2

# Polimorfismo

- Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas.  
(cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).
- Considere as classes MacaVerde e MacaVenenosa que extendem a classe Item;

```
MacaVerde umaMaca = new MacaVerde(); // normal  
Item umItem          = new MacaVerde(); // Polimorfismo  
Item umItem          = new MacaVenenosa(); // Polimorfismo
```

- Isso é possível porque uma maçã verde é um ítem também;

# Polimorfismo na prática

- Na prática, duas boas utilidades para polimorfismo são no **uso de listas e como parâmetros**;
- **No caso de listas**, imagine uma fase que possue ao mesmo tempo maçãs verdes e maçãs venenosas. Teríamos que ter uma lista para cada uma delas;
- **Com polimorfismo, podemos ter apenas uma lista de Itens** que conterá maçãs verdes, venenosas e quaisquer outros ítems que venham a existir;

# Exemplo de Polimorfismo com Listas

```
public class Fase3 implements GameStateController {  
    //...  
    // O poder do polimorfismo  
    // Os dois tipos de maçãs são também Itens,  
    // Então, posso armazená-las todas no mesmo lugar.  
    private ArrayList<Item> itens;  
  
    public void load() {  
        // ...  
        //Cria o array lista para os ítems  
        this.itens = new ArrayList<Item>();  
  
        this.itens.add( new MacaVerde() );  
        this.itens.add( new MacaVenenosa() );  
        this.itens.add( new Chocolate() );  
  
        // ...
```

# Polimorfismo como Parâmetros

- Considerando um jogo onde um personagem anda por um cenário e encontra ítems como maçãs será necessário verificar a colisão deste personagem com os diversos ítems.
- A solução **sem polimorfismo** seria implementar um método para detectar a colisão com cada ítem do cenário:

```
public class Personagem {  
    //...  
    public boolean temColisao(MacaVerde item) { .. }  
    public boolean temColisao(MacaVermelha item) { .. }  
    public boolean temColisao(Chocolate item) { .. }  
    // ...  
}
```

# Polimorfismo como Parâmetros

- A solução **com polimorfismo** é muito mais elegante  
seria implementar um método para detectar a colisão  
com objetos da classe Item:

```
public class Personagem {  
    //...  
    public boolean temColisao(Item item) { .. }  
    // ...  
}
```

- Agora poderíamos inclusive detectar a colisão com qualquer  
ítem que estivesse em uma lista de ítems:

```
//...  
for (Item item : this.listaItens) {  
    if( personagem.temColisao( item ) ) { .. }  
}
```

# Classes Abstratas

- Uma classe Abstrata é uma classe que não foi concebida para criar instâncias. Seu propósito é servir como superclasse para outras classes. As classes abstratas podem conter método abstratos (Barnes);
- Classes abstratas modelam idéias abstratas, que não podem ser vistas, tocadas ou executadas sozinhas em um software. Um exemplo de idéia abstrata é um Animal. Não existe um exemplar específico de um animal, o que existem são macacos, cachorros, gatos.
- Para aprofundarmos, considere como exemplo a classe GameObject:

```
public abstract class GameObject {  
    // . . .  
    protected int altura;  
    protected int largura;  
  
    public abstract void step(long timeElapsed);  
    public abstract void draw(Graphics g);  
    // . . .  
    public int getAltura() {  
        return altura;  
    }  
  
    public void setAltura(int altura) {  
        this.altura = altura;  
    }  
  
    public Rectangle getRetangulo() {  
        return new Rectangle(x, y, largura, altura);  
    }  
  
    public boolean temColisao(GameObject obj) {  
        return this.getRetangulo().intersects(obj.getRetangulo());  
    }  
}
```

# Classes Abstratas

- Você jamais poderia criar um objeto diretamente da classe `GameObject`, isso não faria sentido pois esta classe possui métodos abstratos sem nenhuma implementação, o Java não pode aceitar que se criem objetos com métodos sem implementação;

```
GameObject obj = new GameObject() // ERRO
```

# Classes Abstratas

- Você sempre poderá extender uma classe abstrata, e aproveitar todos os métodos e atributos que ela oferece;
- Em troca, a subclasse deverá implementar os métodos abstratos, caso contrário, a subclasse também será abstrata;
- Veja um exemplo de uma classe que representa uma bola navegando pela tela:

```
public class Bola extends GameObject {  
  
    private int velocidadeHorizontal;  
    private int velocidadeVertical;  
  
    public Bola(){ ... }  
  
    //Implementa método abstrato step  
    public void step( long timeElapsed ){  
        this.x += this.velocidadeHorizontal;  
        this.y += this.velocidadeVertical;  
    }  
  
    //Implementa método abstrato draw  
    public void draw( Graphics g ) {  
        //Herda atributos x, y, largura e altura  
        g.setColor( Color.red );  
        g.fillRect(this.x, this.y, this.largura, this.altura);  
    }  
    //...  
}
```

# Classes Interface

- Uma interface em Java é uma especificação de um Tipo na forma de um nome e um conjunto de métodos que não define nenhuma implementação para os métodos (Barnes).
- Elas são semelhantes as classes abstratas no sentido de que todos os métodos são abstratos e você não pode criar um objeto de uma classe interface;
- Porém, uma interface não pode ser extendida por uma classe concreta, mas apenas implementada.

# Exemplo de Interface que represente uma Fase de um jogo

```
public interface Fase {
```

```
    public void load();  
    public void start();  
    public void step(long timeElapsed);  
    public void draw(Graphics g);
```

```
}
```

Nome da interface.

Conjunto de Métodos que deverão ser implementados pelas classes que implementarem a interface Fase.

## Exemplo de uso da interface Fase

```
public class Fase1 implements Fase {  
    private Personagem personagem;  
  
    public void load() {  
        this.personagem = new Personagem("Ryu", 100);  
    }  
  
    public void step(long timeElapsed) {  
        this.personagem.step( timeElapsed );  
    }  
  
    public void draw(Graphics g) {  
        g.setColor(Color.CYAN);  
        g.fillRect(0, 0, 800, 600);  
  
        this.personagem.draw(g);  
    }  
  
    public void start() { ... }  
}
```

# Classes Enum

- Uma enumeração é um tipo de dado abstrato, cujos valores são atribuídos a exatamente um elemento de um conjunto finito de identificadores;
- Esse tipo é geralmente usado para variáveis categóricas (como os naipes de um baralho, as 8 direções, etc), que não possuem uma ordem numérica definida;
- Usamos variáveis de um tipo Enumerado para gravar valores como "Naipes Espadas" ou "Direção Esquerda" sem precisar usar Strings;

# Exemplo Naipes Baralho

```
public enum Naipe {  
    ESPADAS,  
    PAUS,  
    COPAS,  
    OUROS  
}
```

```
public enum Direcao {  
    ESQUERDA,  
    DIREITA,  
    CIMA,  
    BAIXO,  
    ESQUERDA_CIMA,  
    DIREITA_CIMA,  
    ESQUERDA_BAIXO,  
    DIREITA_BAIXO  
}
```

# Uso da classe Enum Naipe

```
public class Carta {  
    Naipe naipe;  
    int numero;  
    public Naipe(Naipe n, int num){  
        this.naipe = n;  
        this.numero = num;  
    }  
}
```

```
// uso dos valores do conjunto enumerado Naipe  
Carta c1 = new Carta(Naipe.OUROS, 3);  
Carta c2 = new Carta(Naipe.ESPADAS, 3);  
Carta c3 = new Carta(Naipe.PAUS, 3);
```

# Uso da classe Enum Direcao

```
public class Tiro {  
    Direcao direcao;  
    int velocidade;  
    public Tiro(Direcao d, int num){  
        this.direcao = d;  
        this.velocidade = num;  
    }  
}
```

```
// uso dos valores do conjunto enumerado Naipe  
Tiro c1 = new Tiro(Direcao.DIREITA, 3);  
Tiro c2 = new Tiro(Direcao.ESQUERDA, 5);  
Tiro c3 = new Tiro(Direcao.CIMA, 9);
```

# Uso Enum

- Variáveis do tipo enum podem ser utilizadas para trabalhar com os algoritmos do programa;
- Por exemplo, para realizar o movimento de um tiro, dependendo da direção, o valor do x ou do y deverá ser alterado:

```
switch(this.direcao){  
    case DIREITA:  
        this.moveDireita( this.velocidade );  
        break;  
    case ESQUERDA:  
        this.moveEsquerda( this.velocidade );  
        break;  
}
```

# O modificador static

- O modificador **static** pode ser utilizado quando precisarmos acessar um método ou atributo diretamente pela classe e não pelos objetos;
- Em métodos, este recurso é útil quando tudo o que o método precisa pode ser passado por parâmetros, ele não precisa manipular variáveis de instância;
- Este recurso também é útil quando precisamos de funções ao invés de classes. Se tudo o que você precisa é uma função utilitária, pode ter uma classe Util com vários métodos estáticos.

```
public class Util {  
  
    //Paralisa todo o programa por alguns milisegundos  
    static public void sleep(int milliseconds) {  
        try {  
            Thread.sleep(milliseconds);  
        } catch (InterruptedException ex) {  
            System.out.println("Erro: " + ex);  
        }  
    }  
  
    //Sorteia um número aleatório  
    static public int random(int max) {  
        Random r = new Random();  
        return r.nextInt(max);  
    }  
}
```

# Uso de métodos estáticos

```
// ...
```

```
int num = Util.random(38);
System.out.println("Aluno sorteado: "+num);
Util.sleep( 1000 ); //Pausa de um segundo
```

```
num = Util.random(38);
System.out.println("Aluno 2 sorteado: "+num);
Util.sleep( 1000 ); //Pausa de mais um segundo
```

```
num = Util.random(38);
System.out.println("Aluno 3 sorteado: "+num);
```

# Atributos Estáticos

- Usar atributos de uma classe como estáticos permite que todas as instâncias de uma classe acessem o mesmo valor;
- Um uso comum para o modificador static em atributos é quando queremos constantes – valores que não mudam ao longo da execução do software – e que podem ser utilizados em vários lugares diferentes;
- Um exemplo é a classe **Keyboard** que fornece diversos valores de constantes que representam, cada um, uma tecla do teclado.

```
public class Keyboard implements KeyListener  
{  
    // . . .  
    public static int UP_KEY = 38;  
    public static int LEFT_KEY = 37;  
    public static int RIGHT_KEY = 39;  
    public static int DOWN_KEY = 40;  
    public static int ESCAPE_KEY = 27;  
    public static int SPACE_KEY = 32;  
    public static int ENTER_KEY = 10;  
    // . . .  
}
```

# Uso dos atributos estáticos

```
// . . .  
if( k.keyDown( Keyboard.RIGHT_KEY ) ){  
    jogador.aumentaVelocidadeHorizontal();  
}  
  
if( k.keyDown( Keyboard.LEFT_KEY ) ){  
    jogador.diminuiVelocidadeHorizontal();  
}  
// . . .
```

# Tratamento de Exceção

- Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema (Caelum).
  - Por exemplo:
    - Divisão por zero;
    - Objeto vazio tentando executar um método (NULL pointer exception);
    - Arquivo de imagem não encontrado;
  - Em Java é possível capturar estas exceções durante a execução do software e tratá-las da forma mais adequada.

# Tratamento de Exceções

- Tratamos exceções com blocos  
Try { ... } Catch( ... ) { ... }
- Se um erro ocorrer dentro do bloco Try, o Bloco Catch pode capturá-lo e tratá-lo da melhor forma possível.
- Também é possível ter vários blocos Catch para um mesmo bloco Try:  
Try { ... } Catch( ... ) { ... } Catch( ... ) { ... }

# Exemplo de tratamento de exceção

- Uma exceção comum em Jogos é tentar usar uma imagem que não existe, geralmente por erro na digitação do nome do arquivo.
- A classe Sprite já lança um erro quando não encontra um arquivo, este erro pode ser capturado usando um bloco Try { ... } Catch( ... ) { ... }
- 

```
try {  
    this.sprite = new Sprite("resources/nave.png", 4, 64, 64);  
} catch (Exception ex) {  
    System.out.println("Erro: "+ ex.getMessage());  
}
```

## ~~Ferramentas Base~~

- C16) Controle de Versões;
- C10) Interface de desenvolvimento (IDE);

## ~~OO básica~~

- C1) Classes, objetos e instâncias;
- C4) Construtores;
- C2) Comentários;
- C9) Métodos e atributos;

## ~~Elementos de Jogos 2D~~

- C17) Game Design
- C18) Sprites e TileMaps;
- C19) Animação e desenho 2D;
- C25) Captura Eventos Teclado;

## ~~Reuso com OO~~

- C12) Padrões de desenvolvimento (frameworks);
- C5) Encapsulamento;
- C14) Reutilização de código;
- C6) Herança;
- C3) Composição, agregação;

## ~~OO básica 2~~

- C7) Polimorfismo;
- C21) Classes abstratas;
- C22) Classes Interface;
- C23) Classes Enum;
- C24) Métodos e atributos estáticos;
- C20) Tratamento de Exceção;

# Bibliografia

- (Caelum) Apostila Caelum FJ-11;
- (Barnes) Programação Orientada a Objetos com Java: Uma Introdução prática usando o BlueJ;
- (Ricardo Pereira e Silva) Como Modelar com UML2;
- (Raul Sidnei Waslawick) Análise e Projeto de Sistemas de Informação Orientados a Objetos;

# Considerações Finais

- Levei 3 anos desde a primeira vez que ouvi falar de classes Interface e Abstratas para entendê-las e mais 2 anos para perceber com clare quando e como utilizá-las;
- Apenas neste último ano que conheci o poder das classes Enum;
- E Tratamento de Exceção com Java para mim continua sendo sempre um desafio conceitual de quando criar minhas próprias exceções;

# Considerações finais

- Aprendi que não há aprendizado melhor do que 1) Ver como outras pessoas usaram os conceitos; 2) Aplicar minhas próprias idéias para resolver alguns problemas;
- Portanto, quando surgirem desafios de programação, tenha por perto bons livros e alguns exemplos de código, uma simples revisada pode lhe ajudar a ver com clareza e uma luz bem lá no fundo irá brilhar com o conceito certo para o momento.