Java - Classes e Métodos

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

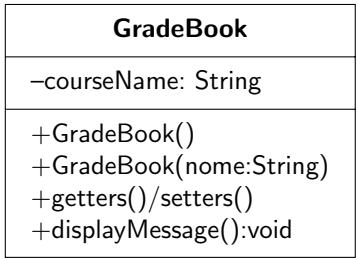
Construtores

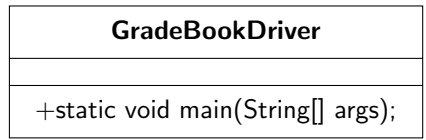
- Java também utiliza construtores para inicializar objetos assim que são criados
  - Por padrão, o compilador fornece um construtor *default*, que não possui argumentos e inicializa os membros de um objeto
    - Tipos primitivos são zerados;
    - O construtor de objetos internos são chamados automaticamente.

Construtores (cont.)

- O operador **new** invoca o construtor;
- A chamada do construtor é indicada pelo nome da classe seguida de parênteses
  - O construtor deve ter o mesmo nome da classe;
  - Não retorna nada e não possui tipo.
- Exemplo: Diario de classe: *GradeBook*

Construtores (cont.)





Construtores (cont.)

```
public class GradeBook{
   private String courseName;

   // construtor
   public GradeBook(String name){
      setCourseName(name);
   }

   // método para definir o nome da disciplina
   public final void setCourseName( String name ){
      courseName = name;
   }
```

Construtores (cont.)

```
    // método para recuperar o nome da disciplina
    public String getCourseName() {
        return courseName;
    }

    // mostra a mensagem de bem vindas
    public void displayMessage() {
        // mostra a mensagem
        System.out.printf("Welcome to the grade book
          for\n%s!\n", getCourseName() );
    }
} // fim classe GradeBook
```

Construtores (cont.)

```
public class GradeBookTest{
    // método principal que inicia a execução
    public static void main( String[] args )
    {
        // cria um objeto da classe GradeBook
        GradeBook myGradeBook = new GradeBook("BCC221");

        System.out.printf( "Initial course name is:
          %s\n\n", myGradeBook.getCourseName() );
     }
}
```

Exemplo

Construir uma classe *Conta* que matém o *saldo* de uma conta bancária além do *nome* do cliente.

Exemplo (cont.)

```
public class Conta {
    private String nome;
    private double saldo;

    public Conta(){
    }
    public Conta(String nome, double saldo){
        setNome(nome);
        setSaldo(saldo);

    }

    public final void setNome(String nome){
        this.nome = nome;
    }
    public String getNome(){
        return nome;
    }
```

Exemplo (cont.)

```
    public final void setSaldo(double saldo){
        this.saldo = saldo > 0 ? saldo : 0;
    }

    public double getSaldo(){
        return saldo;
    }

    public void deposita(double valor){
        if (valor > 0){
            setSaldo(getSaldo()+valor);
        }
    }

    public void print(){
        System.out.println("Nome : " + getNome() + "\nSaldo: " +
            getSaldo());
    }
}
```

Exemplo (cont.)

```
public class DriverConta {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        Conta c1 = new Conta();
        Conta c2 = new Conta("Leonardo", 1500.56);

        System.out.print("Digite nome: ");
        String nome = input.nextLine();
        System.out.print("Digite saldo: ");
        double saldo = input.nextDouble();

        c1.setNome(nome);
        c1.setSaldo(saldo);
        c1.print();
        c2.print();
    }
}
```

Exemplo (cont.)

```
Digite nome: Miriam
Digite saldo: 2346.9

Nome: Miriam
Saldo: 2346.9

Nome: Leonardo
Saldo: 1500.56}
```

Finalizadores e Coleta de Lixo Automática

- Toda classe em Java deriva da classe **Object**;
- Um dos métodos herdados é o **finalize**
  - Raramente utilizado;
  - Dos 6500 códigos da API Java, somente 50 o utilizam.

Finalizadores e Coleta de Lixo Automática (cont.)

- Cada objeto criado consome recursos de sistema, como memória
  - A JVM realiza a coleta de lixo automática (*garbage collection*);
  - Quando não há mais referências a um objeto na memória, tal objeto é marcado para a coleta de lixo;
  - Quando o coletor de lixo (*garbage collector*) for executado, os recursos utilizados por aquele objeto serão liberados.

Finalizadores e Coleta de Lixo Automática (cont.)

- Desta forma, estouros de memórias, comuns em C e C++, são menos propensas a ocorrer
- O método *finalize* é **chamado pelo coletor de lixo** para realizar a **terminação do objeto** antes que seus recursos sejam liberados;

Finalizadores e Coleta de Lixo Automática (cont.)

- Um problema com o método *finalize* é que não há garantia de que o coletor de lixo será executado antes de o programa terminar
  - Logo, a execução de sua aplicação não deve depender dele.
- De fato, desenvolvedores profissionais indicam que o método *finalize* não é útil em aplicações Java para empresas
  - Outras técnicas de liberação de recursos devem ser utilizadas;
  - Geralmente, as classes relacionadas à manipulação de recursos providenciam outras maneiras de liberá-los

Membros *static*

- Assim como em C++, o modificador static define membros que serão instanciados uma única vez
  - Ou seja, não haverá uma cópia para cada objeto;
  - Todos os objetos compartilham uma única cópia.
- O exemplo a seguir mostra a utilização de membros *static* e também do método *finalize*.

Membros *static* (cont.)

```
public class Rec {
    private static int n = 0;
    public Rec(){
        n++;
    }
    protected void finalize() throws Throwable{
        try{
            n--;
            System.out.println("Finalizou um objeto");
        }
        finally{
            super.finalize();
        }
    }
    public static int getRec(){
        return n;
    }
}
```

Membros *static* (cont.)

```
public class JavaStatic {

    public static void main(String[] args) {

        System.out.println("Variavel estatica: "
                           + Rec.getRec());
        Rec r1 = new Rec();
        Rec r2 = new Rec();
        Rec r3 = new Rec();
        System.out.println("Variavel estatica: "
                           + Rec.getRec());
        r1 = null; r2 = null; r3 = null;
        System.gc();
        System.out.println("Variavel estatica: "
                           + Rec.getRec());

    }

}
```

Membros *static* (cont.)

```
Variavel estatica: 0
Variavel estatica: 3
Variavel estatica: 3
Finalizou um objeto
Finalizou um objeto
Finalizou um objeto
```

Geração de Números Aleatórios

- Existe duas formas de gerar número aleatórios
  - A classe *Random* gera números inteiros, reais em diferentes intervalos
  - O método estático *Math.random()* gera números reais entre 0 e 1.

Geração de Números Aleatórios (cont.)

- Classe Random
  - *nextDouble()*: número real [0,1]
  - *nextFloat()*: número real [0,1]
  - *nextGaussian()*: número com distribuição normal (média = 0, desvio padrão = 1)
  - *nextInt(int n)*: números inteiros [0, n[

Geração de Números Aleatórios (cont.)

Gerar 10000 números aleatórios e determinar a frequência de cada um;

Geração de Números Aleatórios (cont.)

```
import java.util.*;

public class ExemploRandom {

    public static void myRandomMath(int m_ini, int m_final){
        int tot = (m_final - m_ini) + 1;
        int n;
        int hist[] = new int[tot];
        for (int i = 0; i < 100000; i++){
            n  = (int)( Math.random()*(m_final-m_ini+1) ) ;
            hist[n]++;
        }
        print(hist);
    }
```

Geração de Números Aleatórios (cont.)

```
public static void myRandomClass(int m_ini, int m_final){
    int tot = (m_final - m_ini) + 1;
    int n;
    int hist[] = new int[tot];
    Random numRandom = new Random();
    for (int i = 0; i < 100000; i++){
        n  = numRandom.nextInt(tot);
        hist[n]++;
    }
    print(hist);
}
```

Geração de Números Aleatórios (cont.)

```
    public static void print(int[] v){
        for (int elem : v)
            System.out.println(elem + " ");
    }

    public static void main(String[] args) {
        System.out.println("Frequencia com Math.random");
        myRandomMath(5, 10);
        System.out.println("Frequencia com Random class");
        myRandomClass(5, 10);
    }

}
```

Geração de Números Aleatórios (cont.)

```
Frequencia com Math.random
16673
16500
16818
16750
16729
16530
Frequencia com Random class
16789
16639
16702
16625
16420
16825
```

Final e static

- Classe final: esta classe não pode ser extendida;
- Método final: método que não pode ser sobrescrito nas subclasses;
- Atributo final: seu valor pode ser atribuido uma única vez, seja na própria declaração ou no construtor.
- Classe static: é uma classe declarada dentro de outra classe, que pode ser usada sem a necessidade de uma instância;
- Método static: é um método que pode ser chamado sem que seja preciso instanciar o objeto da classe que o contém;
- Atributo static: possui o mesmo valor em todas as instâncias do objeto.

Exemplo II

Um jogo popular de azar é um jogo de dados conhecido como craps, que é jogado em cassinos e nas ruas de todo o mundo. As regras do jogo são simples e diretas:

1. Você lança dois dados. Cada dado tem seis faces que contêm um, dois, três, quatro, cinco e seis pontos, respectivamente.

2. Depois que os dados pararam de rolar, a soma dos pontos nas faces viradas para cima é calculada.

3. Se a soma for 7 ou 11 no primeiro lance, você ganha.

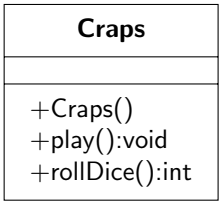
4. Se a soma for 2, 3 ou 12 no primeiro lance (chamado "craps"), você perde (isto é, a "casa" ganha).

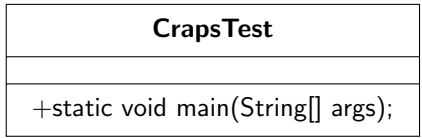
Exemplo II (cont.)

5. Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se sua "pontuação". Para ganhar, você deve continuar a rolar os dados até "fazer sua pontuação" (isto é, obter um valor igual à sua pontuação).

6. Você perde se obtiver um 7 antes de fazer sua pontuação.

Exemplo II (cont.)





Exemplo II (cont.)

```
public class Craps {
    private static final Random randomNumber = new Random();

    private enum Status {CONTINUE, WON, LOST};

    private static final int SNAKE_EYES = 2;
    private static final int TREY = 3;
    private static final int SEVEN = 7;
    private static final int YO_LEVEN = 11;
    private static final int BOX_CARDS = 12;

    public void play(){
        int myPoint = 0;
        Status gameStatus;

        int sumOfDice = rollDice();
```

Exemplo II (cont.)

```
switch(sumOfDice){
    case SEVEN:
    case YO_LEVEN:
        gameStatus = Status.WON; // 7, 11
        break;
    case SNAKE_EYES:
    case TREY:
    case BOX_CARDS:
        gameStatus = Status.LOST; // 2,3,12
        break;
    default:
        gameStatus = Status.CONTINUE;
        myPoint = sumOfDice;
        System.out.println("Pontuacao : " + myPoint);
}
```

Exemplo II (cont.)

```
/* Rule switch
        gameStatus =  switch(sumOfDice){
              case SEVEN, YO_LEVEN -> Status.WON;
              case SNAKE_EYES, TREY, BOX_CARDS -> Status.LOST;
              default -> {myPoint = sumOfDice;
                  System.out.println("Pontuacao: " + myPoint);
                  yield Status.CONTINUE;
                  }
         };
         */
```

Exemplo II (cont.)

```
        while ( gameStatus == Status.CONTINUE) {
            sumOfDice = rollDice();

            if (sumOfDice == myPoint)
                gameStatus = Status.WON;
            else
                if (sumOfDice == SEVEN)
                    gameStatus = Status.LOST;
        }

        if (gameStatus == Status.WON){
            System.out.println("O jogador vence");
        }
        else{
            System.out.println("O jogador perde");
        }
    }
```

Exemplo II (cont.)

```
    public int rollDice(){
        int die1 = 1 + randomNumber.nextInt(6);
        int die2 = 1 + randomNumber.nextInt(6);

        int sum = die1 + die2;

        System.out.println("Jogador lanca :" + sum);
        return sum;
    }
}
```

Exemplo II (cont.)

```
package crapstest;

public class CrapsTest {

    public static void main(String[] args) {
        Craps jogo = new Craps();
        jogo.play();
    }

}
```

Métodos *static*

- Embora os **métodos sejam executados** em resposta a **chamadas de objetos**, isto nem sempre é verdade
  - **Eventualmente**, um **método pode executar ações** que **não são dependentes do conteúdo** de um determinado **objeto**;
  - Tais métodos devem ser declarados static.

Métodos *static* (cont.)

- Métodos *static* podem ser **invocados** utilizando-se o **nome da classe seguido de . e o nome do método**

```
classe.metodo(argumentos);
```

- De fato, esta é uma boa prática, para indicar que o método é *static*.

Classe *Math*

- A classe *Math* está definida no pacote *java.lang*
  - Fornece uma coleção de métodos *static* que realizam cálculos matemáticos comuns;
  - Não é necessário instanciar um objeto da classe para poder utilizar seus métodos;
  - Por exemplo:

```
Math.sqrt(900.00);
```

- Os argumentos destes métodos podem ser constantes, variáveis ou expressões.

Classe *Math* (cont.)

Método	Descrição	Exemplo
abs(*x*)	Valor absoluto de *x*	abs(23.7) é 23.7 abs(0.0) é 0.0 abs(-23.7) é 23.7
ceil(*x*)	Arredonda *x* para o menor inteiro maior que *x*	ceil(9.2) é 10.0 ceil(-9.8) é -9.0
cos(*x*)	Cosseno de *x* (*x* em radianos)	cos(0.0) é 1.0
exp(*x*)	Exponencial e^x	exp(1.0) é 2.71828 exp(2.0) é 7.38906
floor(*x*)	Arredonda x para o menor inteiro não maior que *x*	floor(9.2) é 9.0 floor(-9.8) é -10.0
log(*x*)	Logaritmo natural de *x* (base *e*)	log(Math.E) é 1.0 log(Math.E * Math.E) é 2.0

Classe *Math* (cont.)

Método	Descrição	Exemplo
max(*x*,*y*)	Maior valor entre *x* e *y*	max(2.3, 12.7) é 12.7 max(-2.3, -12.7) é -2.3
min(*x*,*y*)	Menor valor entre *x* e *y*	min(2.3, 12.7) é 2.3 min(-2.3, -12.7) é -12.7
pow(*x*,*y*)	*x* elevado a *y* (x^y)	pow(2.0, 7.0) é 128.0 pow(9.0, 0.5) é 3.0
sin(*x*)	Seno de *x* (*x* em radianos)	sin(0.0) é 0.0
sqrt(*x*)	Raiz quadrada de *x*	sqrt(900.0) é 30.0
tan(*x*)	Tangente de *x* (*x* em radianos)	tan(0.0) é 0.0

Classe *Math* (cont.)

- Declaradas *public final static*
  - Todas as classes podem utilizar;
  - São constantes;
  - Podem ser acessadas pelo nome da classe;

Constante	**Valor**
Math.PI	3.14159265358979323846
Math.E	2.7182818284590452354

Promoção de Argumentos

- A promoção de argumentos consistem em converter o tipo de um argumento
  - Por exemplo, o método *Math.sqrt* espera um *double*, mas pode ser invocado passando-se um *int* como argumento;
  - A promoção é **realizada automaticamente**, desde que se respeite as regras de promoção
  - Especifica quais conversões podem ser realizadas sem a perda de dados.
  - Em uma expressão com dois ou mais **tipos primitivos diferentes**, cada valor é **promovido ao tipo "mais abrangente"**.

Promoção de Argumentos (cont.)

Tipo	Promoções Válidas
double	Nenhuma
float	double
long	float ou double
int	long, float ou double
char	int, long, float ou double
short	int, long, float ou double (mas não char)
byte	short, int, long, float ou double (mas não char)
boolean	Nenhuma (valores booleanos não são considerados números em Java)

Cast

- Considerando a tabela anterior, não é possível realizar a promoção de argumentos de tipos "mais altos" para tipos "mais baixos";
- No entanto, é possível realizar o *cast* explícito
  - Assumindo o risco de erros de truncamento.
- Suponha que o método abaixo só aceita valores inteiros:

```
raizQuadrada((int) valorDouble);
```

Sobrecarga de Construtores

- Java permite que objetos de uma mesma classe sejam inicializados de formas diferentes
  - Através da sobrecarga de construtores;
  - Basta definir múltiplos construtores com assinaturas diferentes

Sobrecarga de Construtores (cont.)

```
public class Tempo
{
    private int h, m, s;

    public Tempo(){
        h = m = s = 0;
    }

    public Tempo(int hora){
        h = hora;
        m = s = 0;
    }

    public Tempo(int hora, int minuto){
        h = hora;
        m = minuto;
        s = 0;
    }
```

Sobrecarga de Construtores (cont.)

```
    public Tempo(int hora, int minuto, int segundo){
        h = hora;
        m = minuto;
        s = segundo;
    }

    public static void main(String args[]){
        Tempo t = new Tempo();
        Tempo t2 = new Tempo(12);
        Tempo t3 = new Tempo(12, 30);
        Tempo t4 = new Tempo(12, 30, 00);
    }
}
```

Sobrecarga de Construtores (cont.)

Telescoping Constructor

```
public class Tempo2 {
    private int h, m, s;

    public Tempo2(){
        this(0,0,0);
    }

    public Tempo2(int hora){
        this(hora, 0);
    }

    public Tempo2(int hora, int minuto){
        this(hora, minuto, 0);
    }

     public Tempo2(int hora, int minuto, int segundo){
        h = hora;
```

Sobrecarga de Construtores (cont.)

```
        m = minuto;
        s = segundo;
    }
```

Observação sobre Construtores em Java

- Java permite que outros métodos possuam o mesmo nome que a classe
  - Embora não se tratem de construtores;
  - Não são chamados quando um objeto da classe é criado;
  - Possuem tipo de retorno.
- Um erro comum é colocar um tipo de retorno em um método com o mesmo nome da classe e confundi-lo com um construtor

Observação sobre Construtores em Java (cont.)

```
public class ConstrutorFalso {

    public int ConstrutorFalso(){
        System.out.println("Um objeto foi criado?");
        return 1;
    }

    public ConstrutorFalso(){
        System.out.println("Um objeto foi criado!");
    }
    public static void main(String[] args) {
        ConstrutorFalso obj = new ConstrutorFalso();
    }
}
```

Composição

- Uma classe Java pode ter referências a objetos de outras classes como membros
  - **Composição**, ou **relacionamento tem-um**.
- Por exemplo, um despertador precisa saber o horário atual
  - É razoável embutir duas referências a objetos de uma classe Hora como membros da classe Despertador.

Hora.java

```
package testedespertador;

public class Hora {
    private int h, m, s;
    public Hora(){
        this(0,0,0);
    }
    public Hora(int h, int m, int s){
        setH(h);
        setM(m);
        setS(s);
    }
    public Hora(Hora h){
        this(h.getH(), h.getM(), h.getS());
    }
    public void setH(int hora){
        h = hora > 0 & hora <24 ? hora : 0;
    }
```

Hora.java (cont.)

```
public int getH(){
    return h;
}
public void setM(int minutos){
    m = minutos > 0 & minutos <60 ? minutos : 0;
}

public int getM(){
    return m;
}

public void setS(int segundos){
    s = segundos > 0 & segundos <60 ? segundos : 0;
}

public int getS(){
    return s;
}
```

Hora.java (cont.)

```
    public String toUniversalString(){
        return String.format("%02d:%02d:%02d",
            getH(), getM(), getS());
    }
    @Override
    public String toString(){
        return String.format("%d:%02d:%02d %s",
            (getH() == 0 || getH() == 12) ? 12 :
            getH() % 12, getM(), getS(),
            (getH() < 12 ? "AM" : "PM"));
    }
}
```

Despertador.java

```
package testedespertador;

public class Despertador {
    boolean ligado;
    Hora alarme, horaAtual;

    public Despertador(){
        alarme = new Hora();
        horaAtual = new Hora();
    }

    public final void setAlarme(int h, int m, int s){
        alarme.setH(h);
        alarme.setM(m);
        alarme.setS(s);
    }
```

Despertador.java (cont.)

```
    public final void setHoraAtual(int h, int m, int s){
        horaAtual.setH(h);
        horaAtual.setM(m);
        horaAtual.setS(s);
    }

    public String getAlarme(){
        return alarme.toString();
    }

    public String getHoraAtual(){
        return horaAtual.toUniversalString();
    }
}
```

TesteDespertador.java

```
package testedespertador;

public class TesteDespertador {

    public static void main(String[] args) {
        Despertador d = new Despertador();
        d.setHoraAtual(14, 56, 20);
        d.setAlarme(18, 15, 0);
        System.out.println("Hora atual: " + d.getHoraAtual());
        System.out.println("Alarme: " + d.getAlarme());
        System.out.println("Ligado? " + d.ligado); // certo?
    }

}
```

TesteDespertador.java (cont.)

```
package testedespertador;

public class TesteDespertador {

    public static void main(String[] args) {
        Despertador d = new Despertador();
        d.setHoraAtual(14, 56, 20);
        d.setAlarme(18, 15, 0);
        System.out.println("Hora atual: " + d.getHoraAtual());
        System.out.println("Alarme: " + d.getAlarme());
        System.out.println("Ligado? " + d.ligado); // certo? Sim, porque
            nao foi definido o tipo de acesso de ligado. Portanto, se comporta como público
            dentro do pacote
    }

}
```

TesteDespertador.java (cont.)

```
Hora atual: 14:56:20
Alarme: 6:15:00 PM
Ligado? false
```

FIM

Overridable method calls in constructors

```
public class Main {
    static abstract class A {
        abstract void foo();
        A() {
            System.out.println("Constructing A");
            foo();
        }
    }

    static class C extends A {
        C() {
            System.out.println("Constructing C");
        }
        void foo() {
            System.out.println("Using C");
        }
    }
```

Overridable method calls in constructors (cont.)

```
    public static void main(String[] args) {
        C c = new C();
    }
}
```

Overridable method calls in constructors (cont.)

```
Constructing A
Using C
Constructing C
```