Plataforma Java

Java Orientação a Objetos

Orientação a Objetos

1. Objeto
2. Classe
3. Construtores
4. Herança
5. Interface
6. Encapsulamento
7. Composição e Agregação
8. Classe Abstrata
9. Classe Final e Metodo Final
10. Static
11. Polimorfismo
12. Package

Orientação a Objetos

13. Classe Object
14. Clonagem de Objetos
15. Classe String
16. Nested Classes

Orientação a Objetos

Objeto

Objeto é uma *“unidade de software”* caracterizadp por: *“Comportamento”* e *“Estado”*.

Frequentemente Objetos são utilizados para modelar a *“Realidade”*.

Exemplos:

Cachorro: estado(nome, cor, raça), comportamento(latir, dormir, correr)

Estados são armazenados em forma de atributos da classe (fields).

Comportamento são implementados através de métodos.

Objetos são *“Instâncias de Classes”*

Orientação a Objetos

Objeto

Benefícios da Orientação a Objeto:

- Modularidade: O código de um objeto pode ser feito de forma independente de outro objeto
- Encapsulamento: Interagindo apenas através dos métodos do objeto, os detalhes de suas implementações e seus atributos internos se mantém oculto para o mundo externo
- Reutilização de Código: Podemos utilizar objetos construídos por outros programadores, e também herdar capacidades (estado e comportamentos) de outros objetos, diminuindo a necessidade de implementá-las
- Plugabilidade e facilidade de corrigir problemas: Objetos problemáticos podem ser removidos facilmente e outros objetos podem ser plugados em seu lugar

Orientação a Objetos

Classe

Classe é um protótipo da qual objetos podem ser criados

Classes representam uma *“Categoria”* de Objetos

Orientação a Objetos

Classe

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

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

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

```
class Bicycle {

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

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

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

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

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

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

Orientação a Objetos

Construtores

Métodos especiais que possuem o mesmo nome da classe

São acionados imediatamente quando um objeto é instanciado através do operador *“new”*

Podem receber parâmetros

Podem receber sobrecarga de métodos (Overloading)

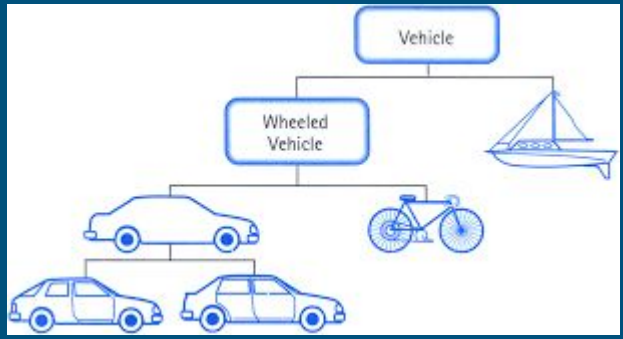
Orientação a Objetos

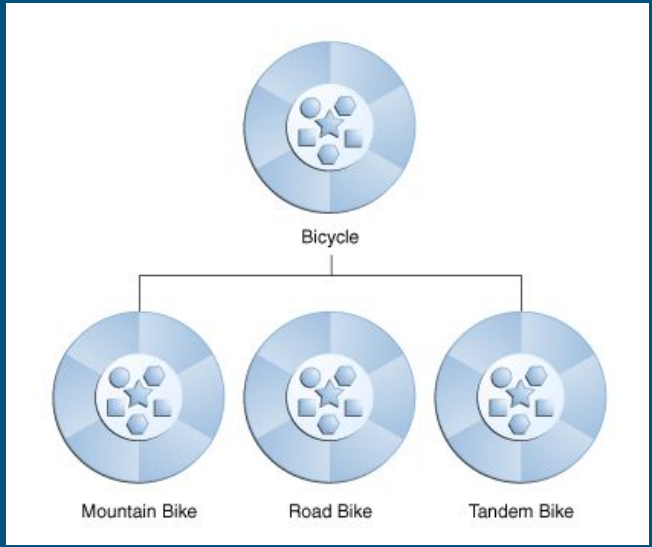
Herança

Frequentemente Objetos diferentes possuem *“Semelhanças”* entre si.

Através da *“Herança”* é possível aproveitar comportamentos e estados comuns, e implementar apenas o que é diferente.

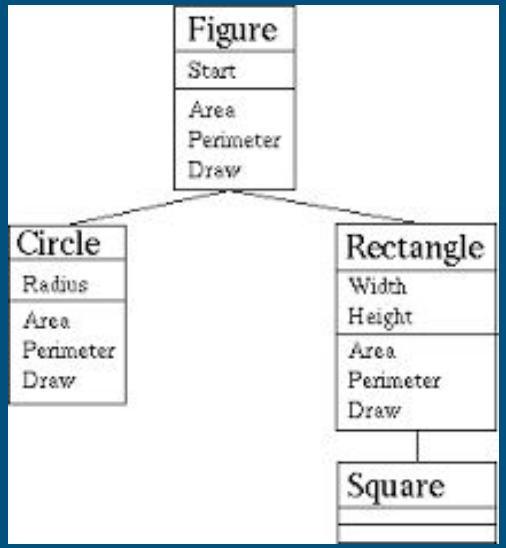
```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here

}
```

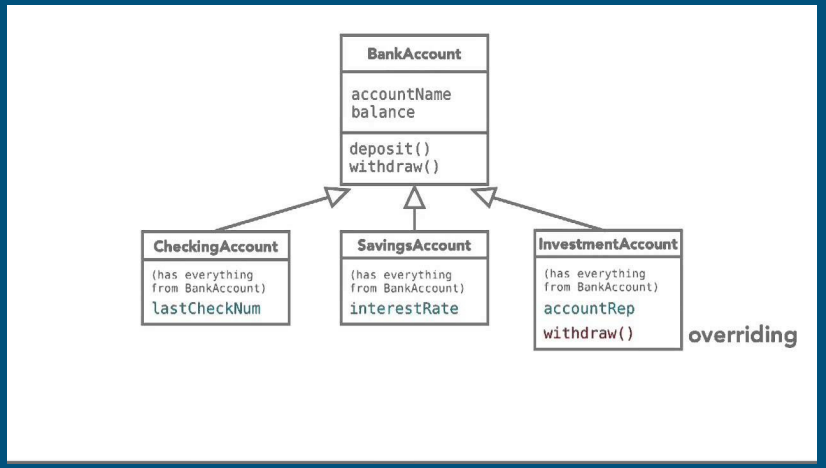


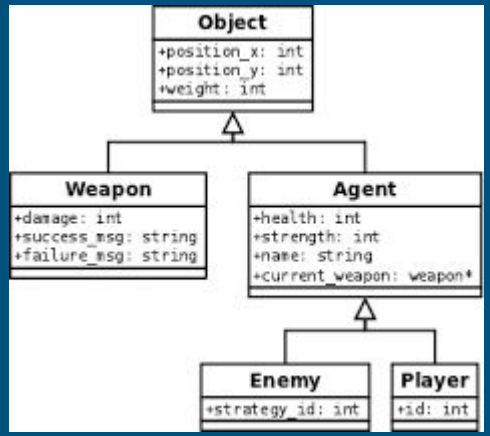


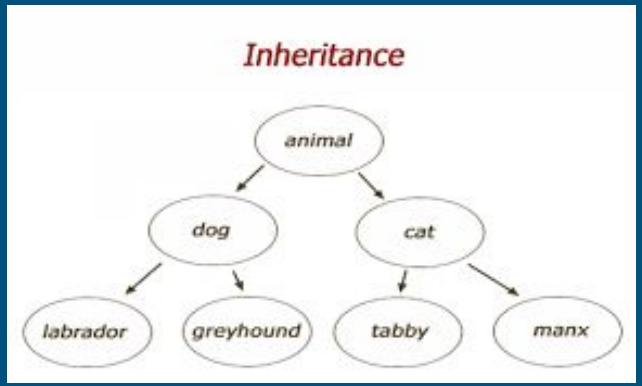
Orientação a Objetos

Herança









Orientação a Objetos

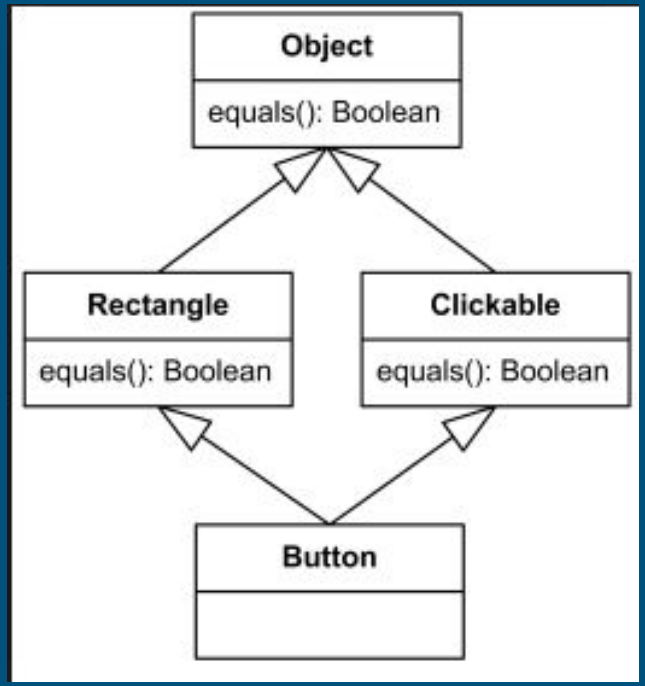
Herança

Superclasse é a classe base da qual se está herdando. Chamada as vezes de *“Classe Pai ou Classe Mãe”*

Subclasse é a classe que está herdando de uma Superclasse. Chamada as vezes de *“Classe Filha”*

Herança Múltipla é a capacidade de que uma classe possa herdar de mais de uma classe.

Java não possui Herança Múltipla



Orientação a Objetos

Herança - Overriding, Dynamic Binding

"Overriding" é quando uma Subclasse "sobrescreve" um método de uma Superclasse, customizando desta forma o comportamento herdado.

O processo de ligar uma chamada de método ao corpo do método é chamado *"binding"*. Devido ao overriding o bind só poderá ser resolvido em tempo de runtime, quando a VM saber exatamente qual o tipo da instância, por este motivo este processo é chamada de Dynamic Binding ou Late Binding.

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
   }
}
```

Orientação a Objetos

Herança - Overload, Static Binding

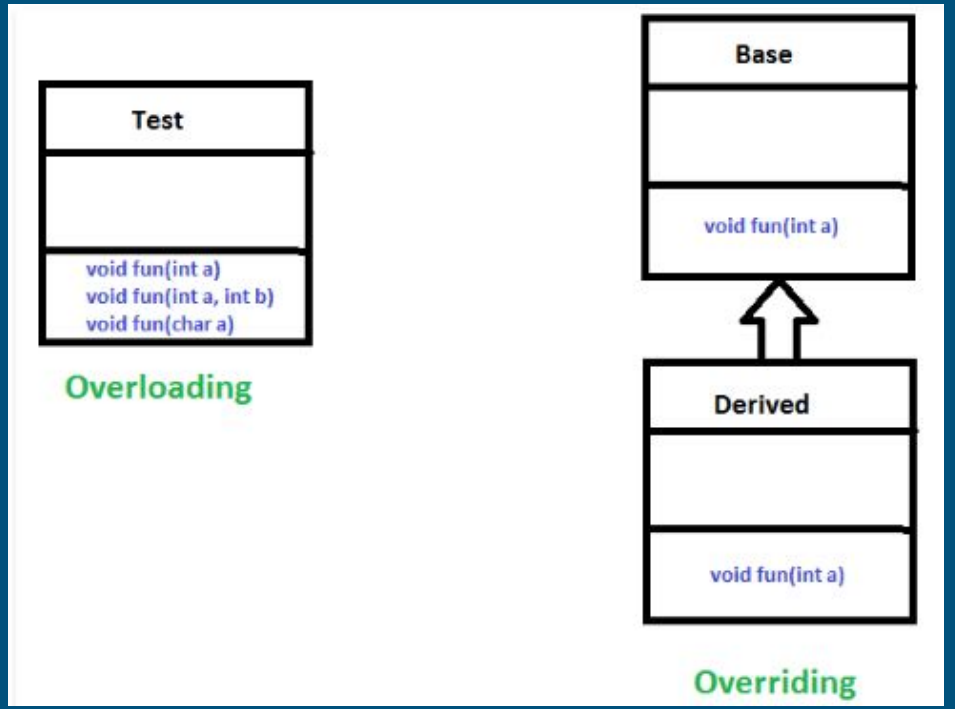
"Overload" é quando existem mais de um método com o mesmo *"Nome"* porém com *"Assinatura"* diferente.

No Overload o compilador resolve o *binding* em tempo de compilação pois basta a assinatura do método. Portanto esta operação é chamada de *"Static Binding"*

```java
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z) {
         return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y) {
         return (x + y);
    }

    // Driver code
    public static void main(String args[]) {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Orientação a Objetos

Herança - Override x Overload



Orientação a Objetos

Polimorfismo

Capacidade de termos comportamentos diferentes (poli=muitos, morf=forma) para diferentes objeto dentro de uma mesma semântica.

Subtype Polimorphism: Overriding

Parametric Polimorfism: Generics

Orientação a Objetos

Subtype Polymorphism, Inclusion Polymorphism

A função letsHear() foi feita para receber Animal, e funciona corretamente se receber um Cat ou um Dog

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

static void letsHear(final Animal a) {
    println(a.talk());
}

static void main(String[] args) {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

Orientação a Objetos

Parametric Polymorphism

Polimorfismo paramétrico permite que funções e tipos seja escrito genericamente e funcione de maneira uniforme sem dependência dos tipos parametrizados

Em Java temos este tipo de polimorfismo com Generics

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List<B> map(Func<A, B> f, List<A> xs) {
    ...
}
```

Orientação a Objetos

Interface

Interface são *“Declarações”* de métodos que serão *“Implementados”* por classes

Podemos entender interfaces como *“Contratos”*, no sentido de que quando uma classe implementar a interface deverá implementar todos os métodos definidos pela interface

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

```
class ACMEBicycle implements Bicycle {

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

   // The compiler will now require that methods
   // changeCadence, changeGear, speedUp, and applyBrakes
   // all be implemented. Compilation will fail if those
   // methods are missing from this class.

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

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

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

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

Orientação a Objetos

Encapsulamento

As Classes podem manter seu *Estado Interno* (atributos) fora do alcance do *“Mundo Externo”* (outros objetos), fornecendo a este métodos específicos.

Isto torna mais seguro a manutenção do estado do objeto

Exemplo: setters

Orientação a Objetos

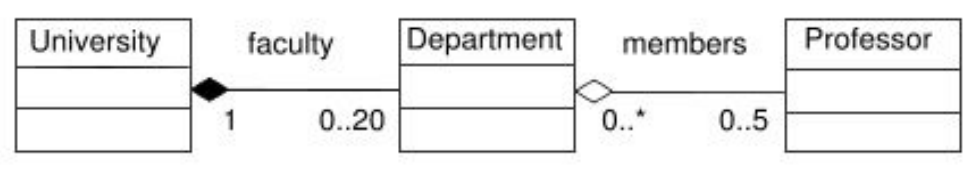
Composição e Agregação

Objetos podem possuir outros objetos em seu estado interno (atributos).

Numa composição quando o objeto composto é destruído, todos os objetos que o compõem também são destruídos.

Agregação é um tipo especial de composição em que os objetos que compõem o objeto *"container"* possuem ciclo de vida independente.

```
class Professor;

class Department
{
  // Aggregation: vector of pointers to Professor objects living outside the Department
  std::vector<Professor*> members;
};

class University
{
  std::vector<Department> faculty;

  University()  // constructor
  {
    // Composition: Departments exist as long as the University exists
    faculty.push_back(Department("chemistry"));
    faculty.push_back(Department("physics"));
    faculty.push_back(Department("arts"));
  }
};
```



Orientação a Objetos

Classes Abstratas

Em Java Classes Abstratas são classes que não podem ser *“instanciadas”*.

Normalmente possuem um ou mais métodos abstratos.

Orientação a Objetos

Classes Final, Metodo Final

Classes final não permite extensão da mesma (Subclasse)

Metodo final não pode ser sobrescrito.

Classes Abstratas e Classes Final são uma o oposto da outra. A primeira é criada para extensão e a segunda é criada para evitar a extensão.

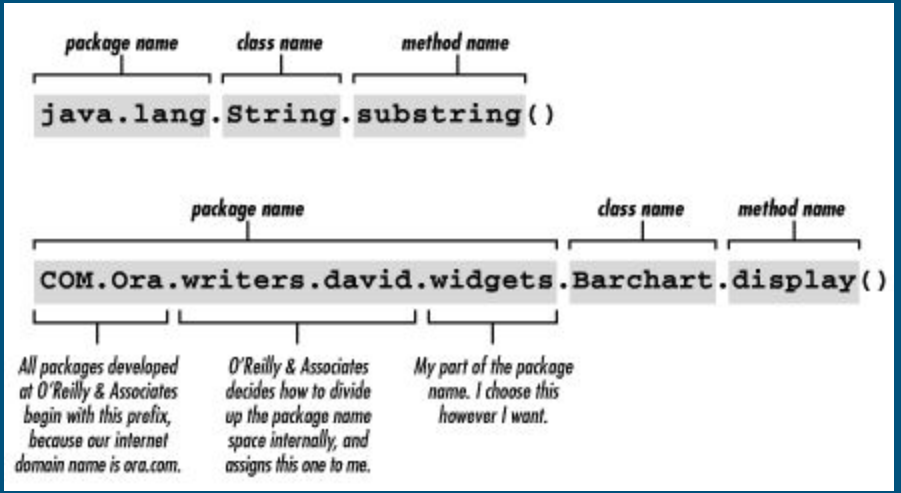
Orientação a Objetos

Package

Package são *“Namespaces”*, utilizado para organizar o código de um projeto

Namespace são utilizados para evitar *“Colisões de Nomes”*, conflitos entre nomes de classes

br.org.cpb.Pessoa != com.indra.Pessoa



Orientação a Objetos

static

Métodos *“static”* são métodos que pertencem a Classe e não ao Objeto

Campos *“static”* são atributos que pertencem a Classe e não ao Objeto, portanto só existe um por classe

Constantes em Java são atributos *“final”* e *“static”*

Orientação a Objetos

Classe Object

Classe base da hierarquia de classes da Linguagem Java

Métodos

- boolean equals(Object)
- String toString()
- Object clone()
- protected void finalize()
- Class<?> getClass()
- int hashCode()

Orientação a Objetos

boolean equals(Object)

Permite criar uma *“semântica”* de igualdade para o tipo em questão

Orientação a Objetos

String toString()

Permite definir uma representação no formato String para o tipo em questão

Orientação a Objetos

Object clone()

Permite que seja definido um processo de clonagem de objetos do tipo em questão

Implementação do Design Pattern GoF chamado de Prototype

Orientação a Objetos

protected void finalize()

Permite que seja definido um comportamento que será acionado quando a JVM for destruir o objeto (antes de o fazê-lo).

Orientação a Objetos

Class<?> getClass()

O tipo Class define a API de Reflection

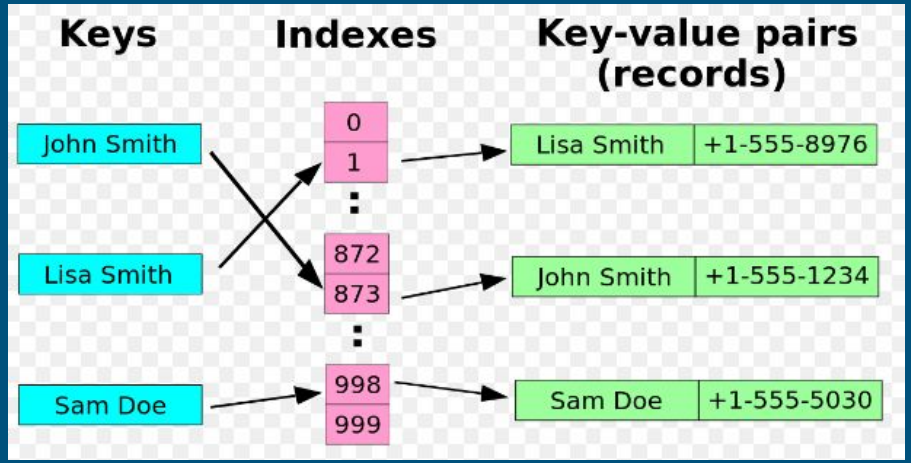
Orientação a Objetos

int hashCode()

Necessário para estruturas de hssh utilizadas no Framework de Collection

Contrato

- Se dois objetos são iguais de acordo com a semântica definida no equals então deverão retornar o mesmo hashCode()
- Não é requerido que se equals() retornar false, os objetos deverão retornar hashCode() distintos



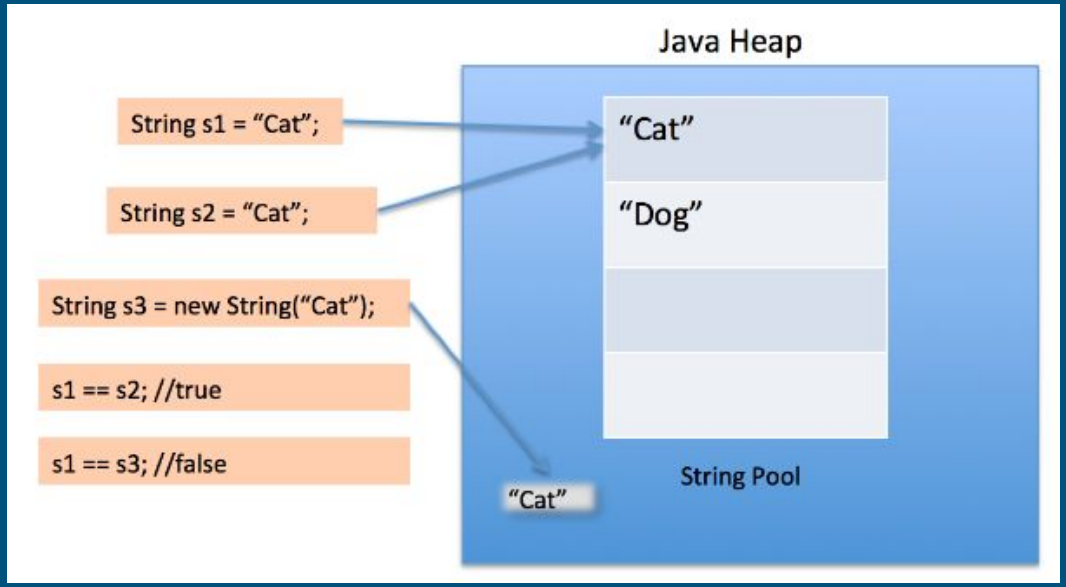
Orientação a Objetos

Classe String

Representa dado do tipo texto

Existe uma estrutura para dar mais performance no tratamento das Strings: Pool de String

Novos objetos String (new) são armazenados no Heap



Orientação a Objetos

Tratamento de Exceções

Situações de erro são representadas com instâncias de classes chamadas Exception

Exemplos

- Divisão por Zero
- Falta de memória para processamento
- Erros de conversão de dados
- Acessar arquivos inexistentes
- Acessar elementos de array além do tamanho do array

Orientação a Objetos

Tratamento de Exceções

A JVM possui um tratador padrão de erros chamado Default Exception Handler

Ações:

- Exibe a mensagem de erro no console
- Exibe o StackTrace, que é a pilha completa de chamadas
- Termina o programa

Orientação a Objetos

Tratamento de Exceções

Quando uma situação de erro é encontrada, temos as seguintes opções para tratar o mesmo:

- Tratar o erro encapsulando o código que pode lançar através do comando *throw* um erro com uma estrutura de *try..catch* ou *try..catch..finally*
- Tratar o erro, mas lançar a exceção para o método chamador
- Não tratar o erro, lançando a exceção para que o método chamador possa tratar o erro

Normalmente o objeto de exceção contém informações sobre o erro, como por exemplo uma mensagem

Orientação a Objetos

Tratamento de Exceções

```
public static void concatenate(String fileName) {
    RandomAccessFile raf = null;

    try {
        raf = new RandomAccessFile(fileName, "r");
    }
    catch (FileNotFoundException fnf) {
        System.err.println("File: " + fileName
                           + " not found.");
    }

}
```

Orientação a Objetos

Tratamento de Exceções

Métodos podem decidir não tratar o erro, simplesmente assinalando que o erro em questão pode ocorrer, e consequentemente o método chamador deverá tratar o problema.

```
public static void concatenate(String fileName)
    throws FileNotFoundException {
    RandomAccessFile raf = null;
```

Orientação a Objetos

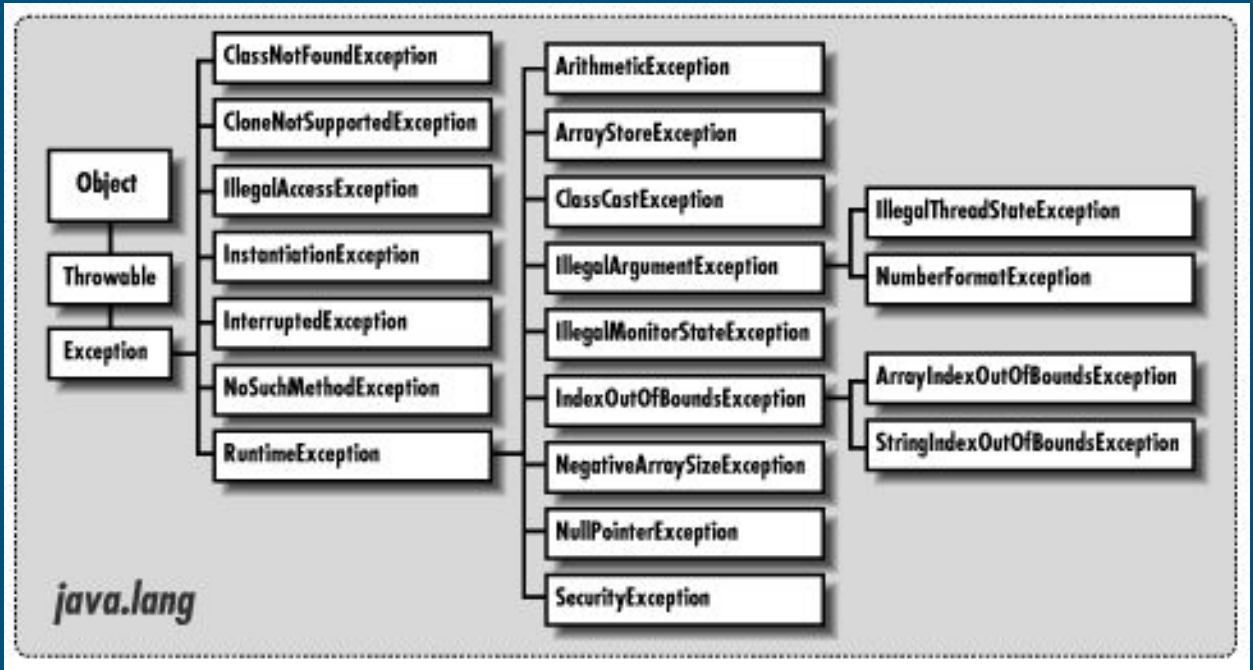
Tratamento de Exceções

A API padrão do Java oferece uma Hierarquia de exceções

- Throwable
  - Classe base da hierarquia
- Exception
  - Classe base para as exceptions que deverão ser tratadas pelos programas do usuário
- RuntimeException
  - O compilador não exige tratamento (Unchecked)
- Error
  - Usado pela JVM para indicar situações de erro no ambiente
  - Unchecked
  - Nada o que o desenvolvedor possa fazer nesta situação

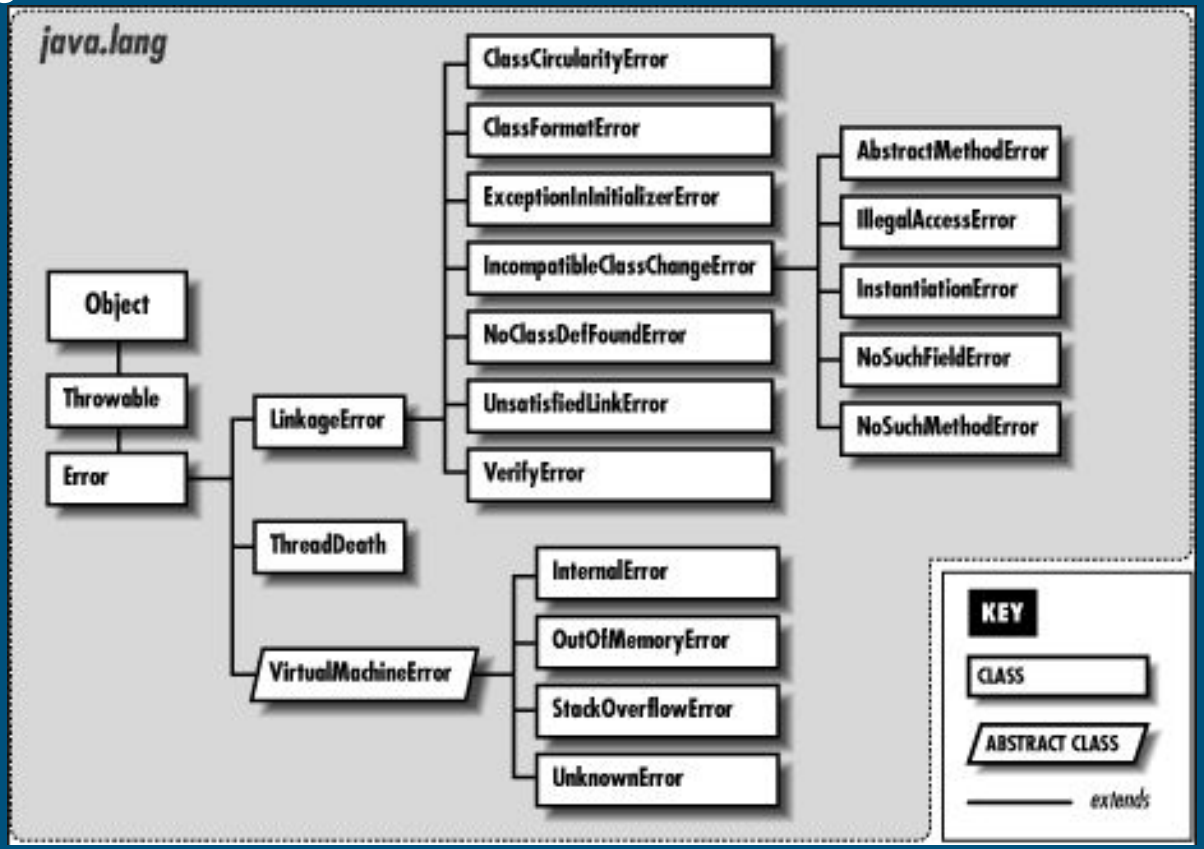
Orientação a Objetos

Tratamento de Exceções



Orientação a Objetos

Tratamento de Exceções



Orientação a Objetos

Assertions

Assertions permitem testar premissas esperadas pelo programador

Forma de utilização:

```
assert expression;

assert expression1 : expression2;
```

Assertions por padrão são desabilitadas

Para habilitar as assertions utilize o parâmetro “ea” ou “enableassertions”:

```
java -ea Test
```

Para desabilitar as assertions utilize o parâmetro “da” ou “disableassertions”:

```
java -da Test
```

Orientação a Objetos

Assertions

Porque utilizar Assertions:

- Se certificar que um código inalcançável é realmente inalcançável
- Verificar se as premissas esperadas são realmente realidade
- Verificar se o default de um switch não é alcançado (qdo isso for desejado)
- Verificar o estado de um objeto
- No início de um método
- Após a invocação de um método

Assertions X Tratamento de Exceções

- Assertions são normalmente utilizadas para verificar situações de impossibilidade lógica
- São utilizadas apenas em fase de desenvolvimento
- Em produção são desabilitadas

Orientação a Objetos

Assertions

Quando utilizar Assertions:

- Verificar argumentos de métodos privados. Argumentos privados são passados pelo próprio programador, e ele poderá verificar se as premissas definidas pelo método são atendidas
- Cases condicionais (default)
- Condições no início do método
- Condições após a execução de um método

Quando NÃO utilizar Assertions

- Não devem ser utilizadas para substituir mensagens de erro
- Não deveriam ser utilizadas para verificar validade de argumentos de métodos públicos, que são informados por usuários, neste caso use o Tratamento de Erro
- Não deve ser utilizado para validar os argumentos de linha de comando

Orientação a Objetos

Assertions

O código do projeto não deve depender dos asserts, pois os mesmos serão desabilitados em produção

Os asserts servem como auxílio na fase de desenvolvimento.

Orientação a Objetos

Nested Classes

Na linguagem Java é possível criar classes dentro de classes ou até de métodos

Nested Classes podem ser static ou não static

Nested classes Não static são chamadas Inner Classes e tem acesso aos outros membros da classe a qual pertencem

Static Nested Classes não possuem acesso aos membros da classe a qual pertencem

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

Orientação a Objetos

Nested Classes

Motivação para usar *Nested Classes*

- Forma de agrupar classes que são logicamente usadas em um único lugar
- Aumentar o encapsulamento
- Pode melhorar a legibilidade e manutenabilidade do código

Orientação a Objetos

Static Nested Classes

Não possuem acesso a métodos e atributos de instância da classe a qual pertencem

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

Orientação a Objetos

Inner Classes

Possuem acesso a métodos e atributos de instância da classe a qual pertencem

Uma instância de InnerClass só pode existir dentro de uma instância de OuterClass

Para instanciar uma InnerClass será necessário primeiramente instanciar uma OuterClass

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

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

Orientação a Objetos

Shadow

Quando uma variável é declarada em diversos escopos (Outer class, inner class, method) com o mesmo nome, temos o efeito de shadow

Orientação a Objetos

Shadow

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

Orientação a Objetos

Local Class

Inner class declarada dentro de um método

Possui acesso aos membros da classe que a contem

```
public class LocalClassExample {

    static String regularExpression = "[^0-9]";

    public static void validatePhoneNumber(
        String phoneNumber1, String phoneNumber2) {

        final int numberLength = 10;

        // Valid in JDK 8 and later:

        // int numberLength = 10;

        class PhoneNumber {

            String formattedPhoneNumber = null;

            PhoneNumber(String phoneNumber){
                // numberLength = 7;
                String currentNumber = phoneNumber.replaceAll(
                  regularExpression, "");
                if (currentNumber.length() == numberLength)
                    formattedPhoneNumber = currentNumber;
                else
                    formattedPhoneNumber = null;
            }
```

Orientação a Objetos

Anonymous Class

Local class declarada sem nome e ao mesmo tempo instanciado um objeto para uso imediato

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }

        HelloWorld englishGreeting = new EnglishGreeting();

        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };
```

Orientação a Objetos

Inner Class Modifiers

É possível utilizar os mesmos modificadores que utilizamos para outros membros de classes (métodos e atributos), ou seja:

public

protected

private