SCC0504 – Programação Orientada a Objetos

Herança e Polimorfismo

Luiz Eduardo Virgilio da Silva
ICMC, USP

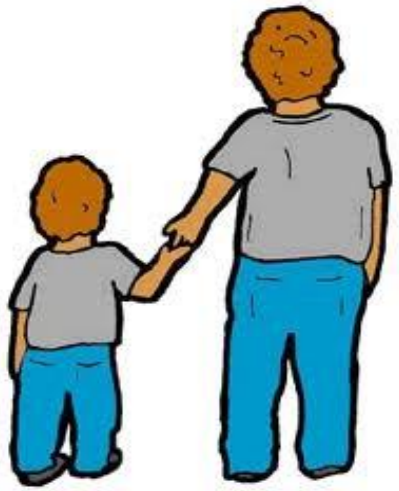
Material baseado nos slides dos professores:
Fernando Paulovich (ICMC/USP)
José Fernando Jr (ICMC/USP)

Sumário

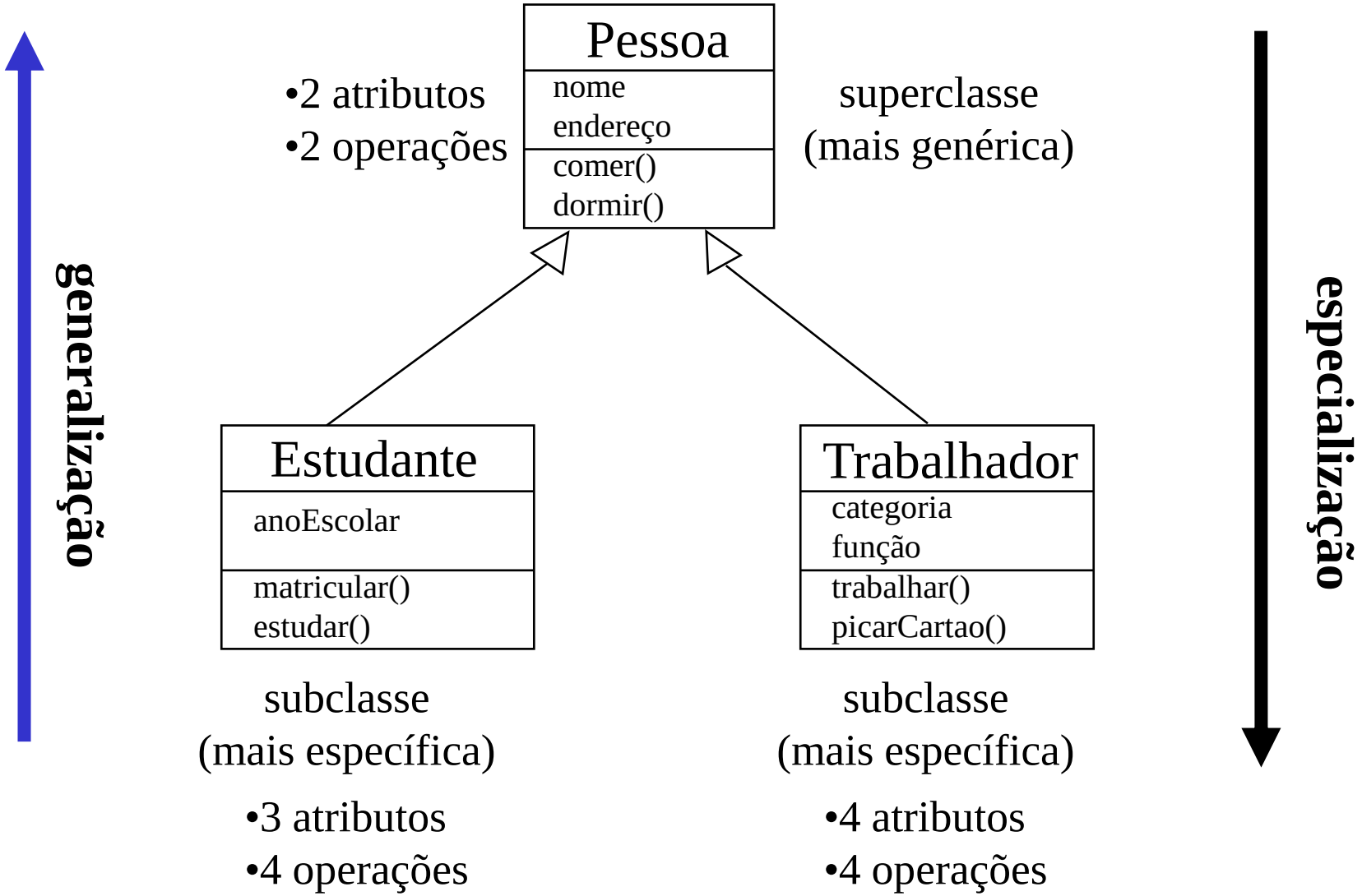
- Revisão de conceitos
- Herança de membros da classe
- Encapsulamento
- Polimorfismo
- Palavra-chave *super*
- Conversão de tipos (*casting*)

Herança

- No mundo real, por meio da Genética, é possível herdarmos certas características de nossos pais
  - Atributos: cor dos olhos, cor da pele, doenças, etc.
  - Comportamentos?
- De forma similar, em POO as classes podem herdar
  - Atributos (propriedades)
  - Métodos (comportamento)
- Chamamos este processo de **herança**

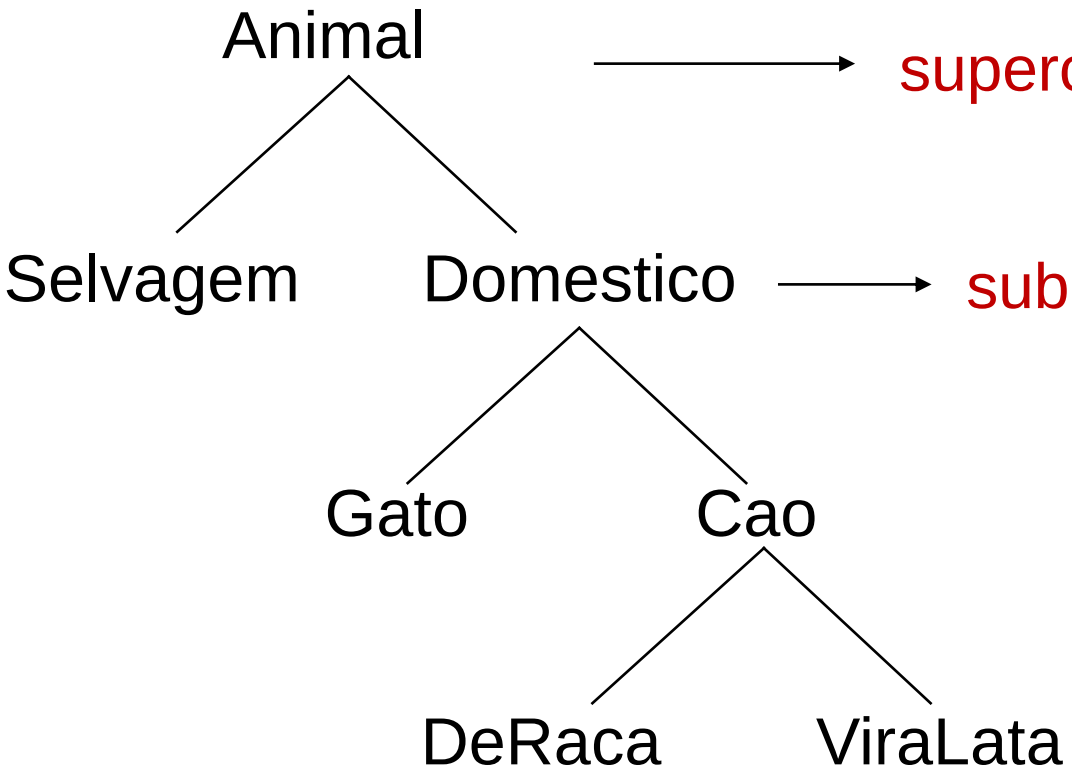


Herança

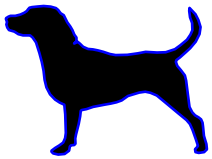

generalização
Pessoa
nome
endereço
comer()
dormir()
•2 atributos
•2 operações
superclasse
(mais genérica)
especialização
Estudante
anoEscolar
matricular()
estudar()
Trabalhador
categoria
função
trabalhar()
picarCartao()
subclasse
(mais específica)
•3 atributos
•4 operações
subclasse
(mais específica)
•4 atributos
•4 operações

Herança

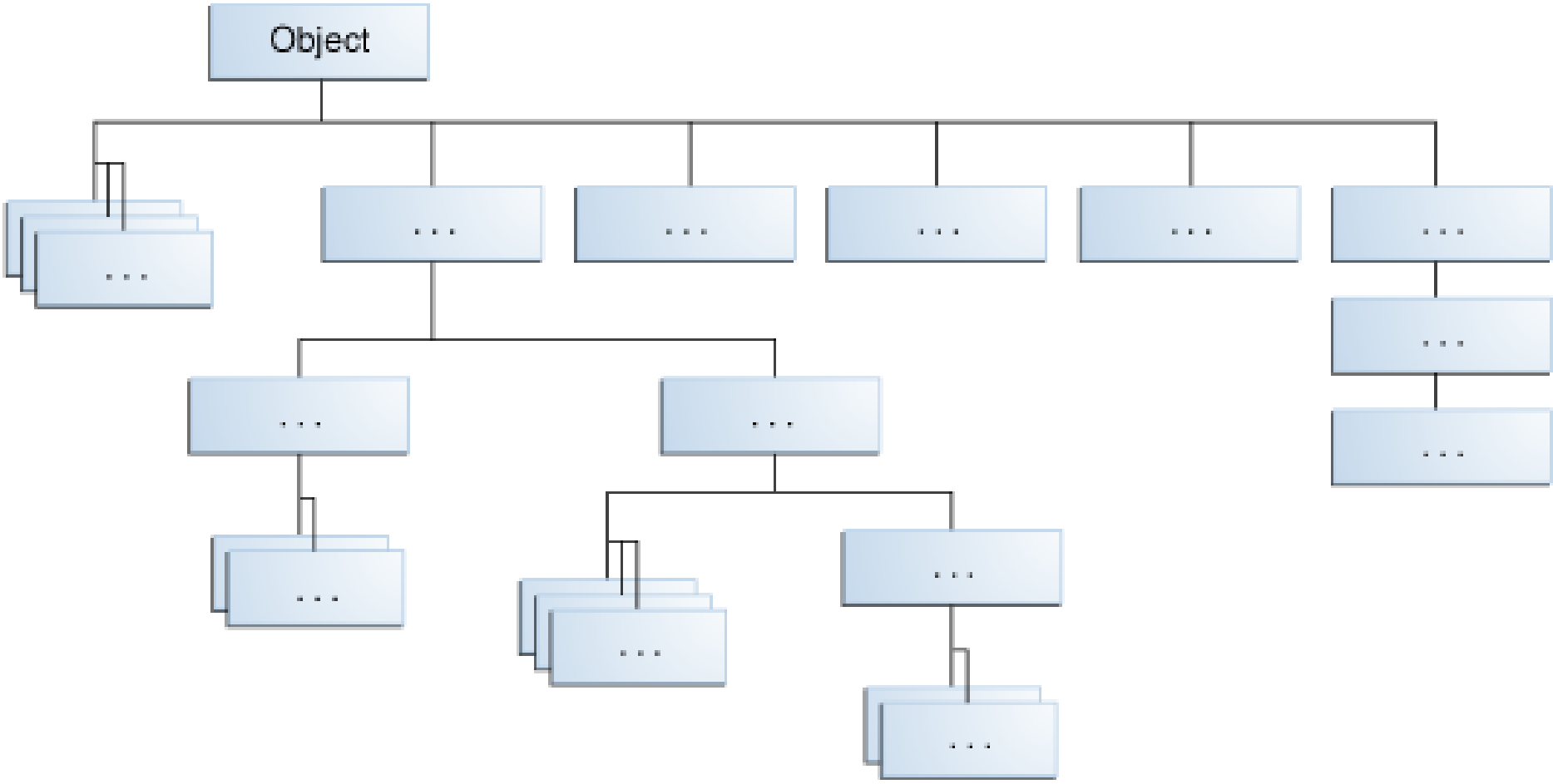
- A herança pode ser repetida em cascata, criando várias gerações de classes



ViraLata rex;

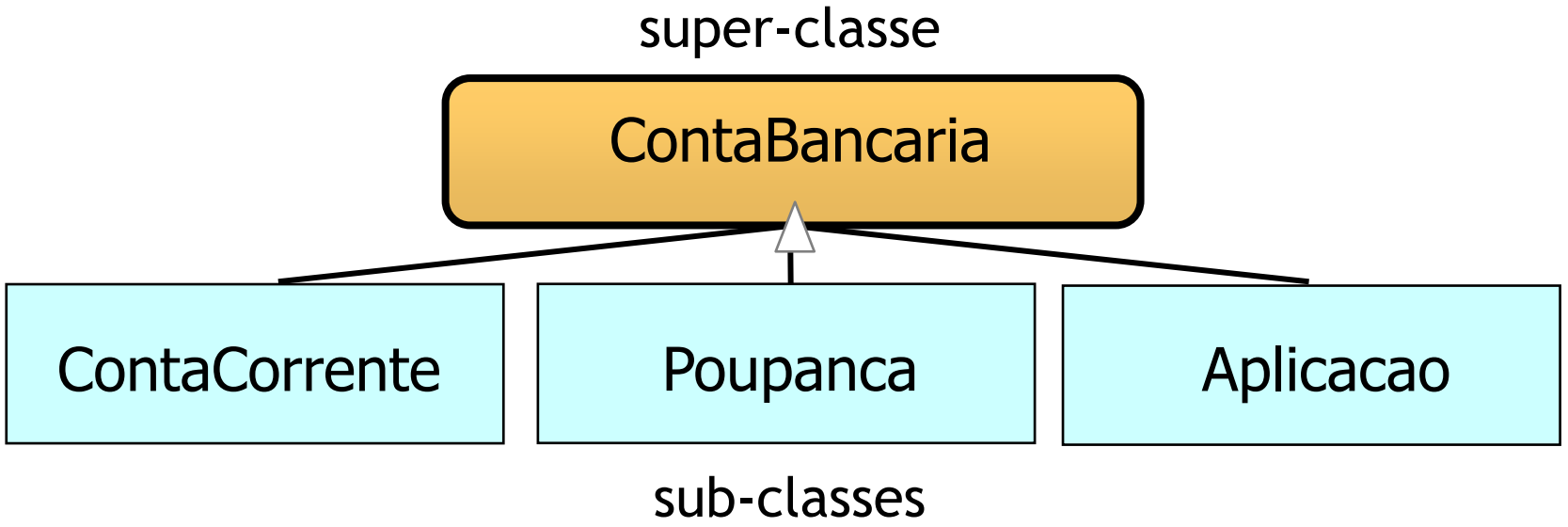


Herança



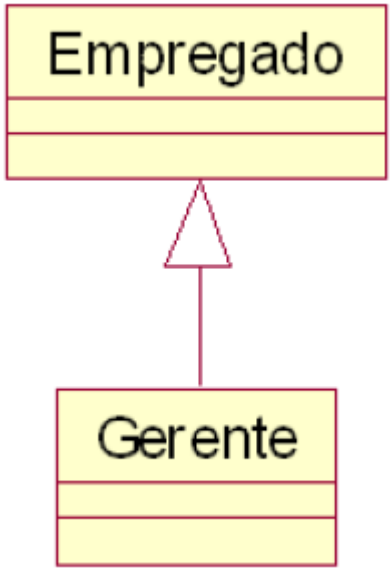
Herança

- **Classe mãe, superclasse, classe base**: A classe mais geral, a partir da qual outras classes herdam membros (atributos e métodos)
- **Classe filha, subclasse, classe derivada**: A classe mais especializada, que herda os membros de uma classe mãe



Herança

- A herança é feita pela palavra-chave extends




```
public class Empregado {

}

public class Gerente extends Empregado {

}
```

Herança

- Herança permite a criação de classes com base em uma classe já existente
  - Proporcionar o **reuso** de software
  - Não é preciso escrever (e degubar) novamente
  - Especialização de soluções genéricas já existentes
- A ideia da herança é “ampliar” a funcionalidade de uma classe
- Todo objeto da subclasse também é um objeto da superclasse, mas NÃO o contrário

Herança - Encapsulamento

- Subclasse herda todos os membros da superclasse
  - Construtores não são membros da classe
    - Contudo, da subclasse é possível chamar um construtor da superclasse
  - Membros privados (-): ocultos na subclasse
    - Acessíveis apenas por métodos
  - Membros protected (#): acessíveis na subclasse (e outras classes do mesmo pacote)
  - Membros package-private: acessíveis se a subclasse estiver no mesmo pacote da superclasse
  - Membros public (+): acessíveis na subclasse (e por qualquer outra classe)

  - Os membros herdados visíveis podem ser usados diretamente, como os membros da própria classe

Herança - Encapsulamento

	Classe	Subclasse no mesmo pacote	Pacote (mesmo pacote)	Subclasse em outro pacote	Exterior (pacotes diferentes)
public	OK	OK	OK	OK	OK
protected	OK	OK	OK	OK	Não
<ausente>	OK	OK	OK	Não	Não
private	OK	Não	Não	Não	Não

Herança - Encapsulamento

	Classe	Subclasse no mesmo pacote	Pacote (mesmo pacote)	Subclasse em outro pacote	Exterior (pacotes diferentes)
public	Pode tudo				
protected	Pode tudo no mesmo pacote, e em subclasses em qualquer lugar				
<ausente>	Pode tudo no mesmo pacote				
private	Não pode nada				

Herança - Encapsulamento

- É possível declarar um campo na subclasse com o mesmo nome de um campo da superclasse
  - Mesmo que os tipos sejam diferentes
  - Ocultamento de campo (não recomendado)
- É possível sobrescrever um método da superclasse, declarando um método com a mesma assinatura
  - Polimorfismo
- É possível declarar novos campos e métodos na subclasse
  - Especialização

Herança - Encapsulamento

- Os membros herdados de uma classe podem ter seu acesso relaxados, mas não o contrário
- Por exemplo, um método protected na superclasse pode ser reescrito como public na subclasse, mas não como private

```
public class Bicycle {
    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Herança - Encapsulamento

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
     }

     // the MountainBike subclass adds one method
     public void setHeight(int newValue) {
         seatHeight = newValue;
     }
}
```

Herança - Encapsulamento

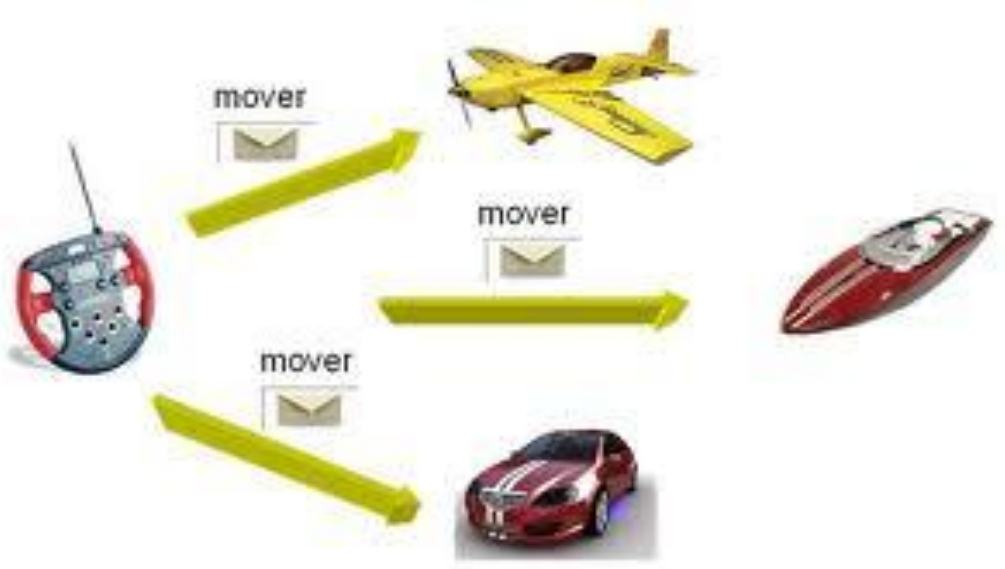
- Se os campos da classe Bicycle fossem privados, subclasses não teriam acesso a eles
  - O acesso poderia acontecer indiretamente, através de métodos públicos ou protegidos da superclasse
- Uma forma de torná-lo acessíveis diretamente na subclasses, mas ainda com restrições, é usar modificador protegido
  - Subclasses e classes do mesmo pacote
- O ideal é manter sempre o mais escondido o possível

Polimorfismo

- Como já discutido, polimorfismo vem da biologia
- É a capacidade de indivíduos ou organismos de uma mesma espécie apresentarem diferentes formas
  - Instâncias

Polimorfismo

- Estendendo para POO
  - Capacidade de objetos responderem a um MESMA MENSAGEM de maneira diferente
  - Mensagem → chamada de método
  - Obtido através da sobreposição (reescrita) de métodos



Polimorfismo

- Quando um método de um objeto é chamado, a JVM procura a implementação mais especializada
  - Hierarquicamente, de baixo (especializado) para cima (geral)
- Se o método não foi definido na classe derivada (subclasse), procura-se pela implementação da classe base (superclasse)
- Quando o método é sobrescrito na subclasse, ele passa a ser o comportamento padrão daquela classe
  - Mas ainda é possível acessar o método da superclasse

Polimorfismo

- A sobrescrita de métodos acontece quando um método da superclasse é redefinido na subclasse
  - Mesma assinatura
  - Mesmo tipo (ou subtipo) de retorno
- Se quisermos aproveitar o comportamento definido pela superclasse, podemos chamar a implementação da superclasse
  - Palavra-chave super
  - Método da superclasse fica sobreposto (*overriding*)

Polimorfismo

- Exemplos

```
public class Bicycle {
    // fields
    private int cadence;
    private int gear;
    private int speed;

    // methods
    // getters and setters

    public void printDescription() {
        System.out.println("\nBike is " + "in gear " +
                           this.gear + " with a cadence of " +
                           this.cadence + " and travelling at speed” +
                           this.speed + ". ");
    }
}
```

Polimorfismo

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike( int startCadence, int startSpeed,
                         int startGear, String suspensionType) {
        super(startCadence, startSpeed, startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension() {
        return this.suspension;
    }
    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }
    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
                           getSuspension() + " suspension.");
    }
}
```

Polimorfismo

```
public class RoadBike extends Bicycle {
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence, int startSpeed,
                    int startGear, int newTireWidth) {
        super(startCadence, startSpeed, startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth() {
        return this.tireWidth;
    }
    public void setTireWidth(int newTireWidth) {
        this.tireWidth = newTireWidth;
    }
    public void printDescription() {
        super.printDescription();
        System.out.println("The RoadBike" + " has " +
                           getTireWidth() + " MM tires.");
    }
}
```

Polimorfismo

```
public class TestBikes {
    public static void main(String[] args) {
        Bicycle bike01, bike02, bike03;

        bike01 = new Bicycle(20, 10, 1);
        bike02 = new MountainBike(20, 10, 5, "Dual");
        bike03 = new RoadBike(40, 20, 8, 23);

        bike01.printDescription();
        bike02.printDescription();
        bike03.printDescription();
    }
}
```

Polimorfismo

- A JVM chama o método correto de cada objeto, mesmo que eles estejam referenciado sob um tipo mais geral

```
Bike is in gear 1 with a cadence of 20 and travelling at a
speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a
speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a
speed of 20.
The RoadBike has 23 MM tires.
```

Polimorfismo

- Anotação @Override
  - Em geral, é uma boa prática anotar os métodos que foram sobrescritos quando herdamos de uma superclasse
  - Se o método não existe em nenhuma superclasse, o compilador acusa erro
  - Também ajuda no entendimento do código

```
public class RoadBike extends Bicycle {
    ...

    @Override
    public void printDescription() {
        // code goes here
    }
}
```

Polimorfismo

- Alguns autores consideram a sobrecarga de métodos como uma forma de polimorfismo
  - Métodos com nomes iguais mas assinaturas diferentes
  - Assinatura: nome do método, número de parâmetros e tipos dos parâmetros
  - Nome dos parâmetros e tipo de retorno NÃO fazem parte da assinatura

```
public int quadrado(int x) {
    return x * x;
}

public double quadrado(double x)
{
    return x * x;
}
```

Palavra-chave *super*

- Vimos anteriormente que a palavra chave this pode ser usada para referenciar o próprio objeto
- Isso permite distinguir variáveis locais e campos do objeto que contém os mesmos nomes
- A palavra-chave super tem uma função parecida em herança: acessar campos e métodos da superclasse
  - Campos ocultos (*hidden fields*)
  - Métodos sobrescritos (polimorfismo)
  - Construtores da superclasse

Palavra-chave *super*

- Campos ocultos
  - Se a subclasse tem um campo com o mesmo nome de um campo na superclasse
  - Para acessar o campo da superclasse, deve-se usar o super
  - Desaconselhável, pois torna o código de difícil interpretação

Palavra-chave *super*

- Métodos sobrescritos

```
public class Superclass {
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

```
public class Subclass extends Superclass {

    @Override
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }

    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Palavra-chave *super*

- Métodos sobrescritos
  - Compilando e executando Subclass

```
Printed in Superclass.
Printed in Subclass
```

Palavra-chave *super*

- Construtores
  - Usamos super e o conjunto de argumentos entre parêntesis
  - Deve ser sempre a primeira instrução do construtor da subclasse

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike( int startCadence, int startSpeed,
                         int startGear, String suspensionType) {
        super(startCadence, startSpeed, startGear);
        this.setSuspension(suspensionType);
    }
}
```

Palavra-chave *super*

- Construtores
  - Quando não há uma chamada explícita a um construtor da superclasse, o compilador Java insere uma chamada ao construtor sem argumentos da superclasse
  - Se a superclasse não tem tal construtor, um erro é gerado

Importante

- Lembre-se que quando não há superclasse declarada, a superclasse direta é Object
- Quando nenhum construtor é definido na classe, o compilador Java cria um construtor padrão

Classes e Métodos *final*

- Métodos final não podem ser sobrescritos
  - Em geral, para comportamentos que não devem ser mudados
  - Quando os métodos são essenciais para manter a consistência dos objetos

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...

    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Classes e Métodos *final*

- Métodos final não podem ser sobrescritos
  - Em geral, para comportamentos que não devem ser mudados
  - Quando os métodos são essenciais para manter a consistência dos objetos
  - É recomendado que métodos chamados dentro de um construtor sejam final
    - Caso contrário, uma subclasse poderia alterar a maneira como o método é construído
    - Pode produzir resultados indesejados

Classes e Métodos *final*

- Classes final não podem ser herdadas
  - Exemplo: classe String
  - Garante que a String não será manipulada

Conversão de Tipos (*Casting*)

- Como vimos, a herança permite dizer que um objeto mais especializado também é um tipo mais geral
  - Herança é uma relação “é um”
  - Exemplo: MountainBike é uma Bicycle e um Object
- Isso permite declarar tipos especializados como tipos mais gerais
  - **Casting implícito**

```
MountainBike mountainBike = new MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();
```

Conversão de Tipos (*Casting*)

- Se tentarmos associar um tipo mais geral a um tipo mais especializado, teremos um erro de compilação
  - Compilador não sabe que o tipo mais geral é também um tipo especializado

```
MountainBike mountainBike = new MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();

mountainBike = obj;  // compile-time error
```

Conversão de Tipos (*Casting*)

- Para corrigir isso, devemo fazer um **casting explícito**, informando ao compilador que o objeto é de fato um tipo mais especializado
  - Casting explícito só vale dentro da hierarquia de herança

```
MountainBike mountainBike = new MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();

mountainBike = (MountainBike)obj; // Ok
```

Conversão de Tipos (*Casting*)

- Se durante a execução o objeto não for do tipo assinalado, uma exceção é lançada
- Para evitar erro de execução, podemos testar o tipo da classe com instanceof

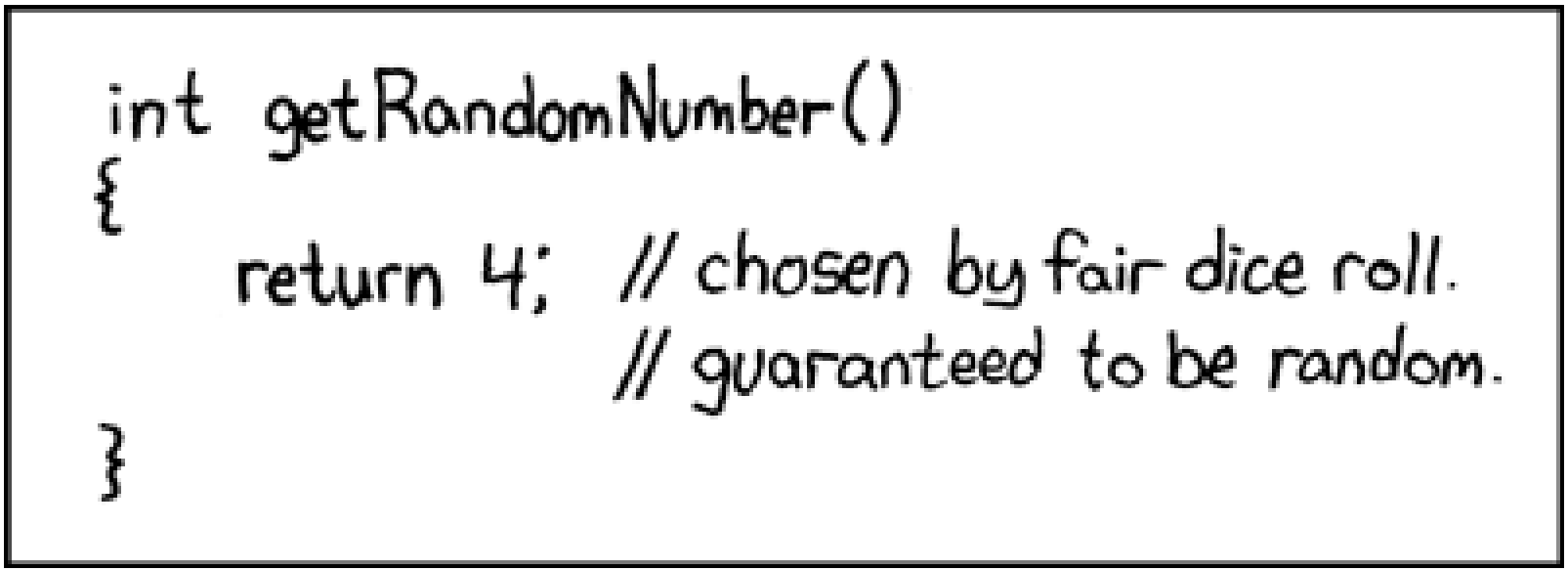
```
MountainBike mountainBike = new MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();

if (obj instanceof MountainBike)
    mountainBike = (MountainBike)obj;
```

Resumo

- Revisão de conceitos
- Herança de membros da classe
- Encapsulamento
- Polimorfismo
- Palavra-chave *super*
- Conversão de tipos (*casting*)

Dúvidas


int getRandomNumber()
{
return 4; // chosen by fair dice roll.
// guaranteed to be random.
}