

COTIL



UNICAMP

Linguagem de Programação III

Profa. Simone Berbert Rodrigues Dapólito
Profa. Tania Basso

Cap. 9 - Herança

Herança

- A herança é um mecanismo que permite que uma classe possa herdar os atributos e comportamentos (métodos) de outra classe, ao mesmo tempo em que novos comportamentos podem ser estabelecidos.

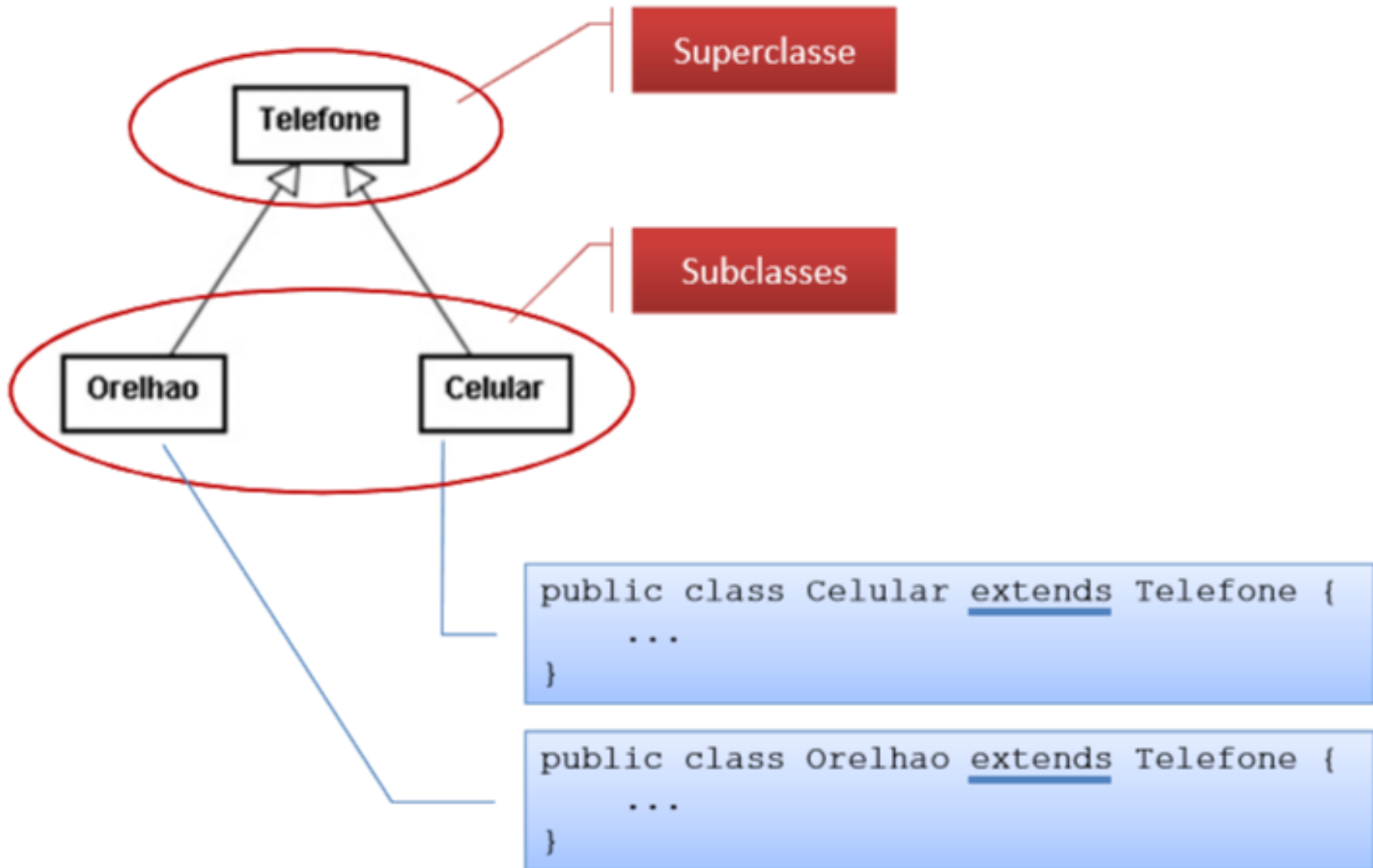
Herança

- A vantagem da herança é agrupar coisas comuns para poder reaproveitar código.

Exemplo:

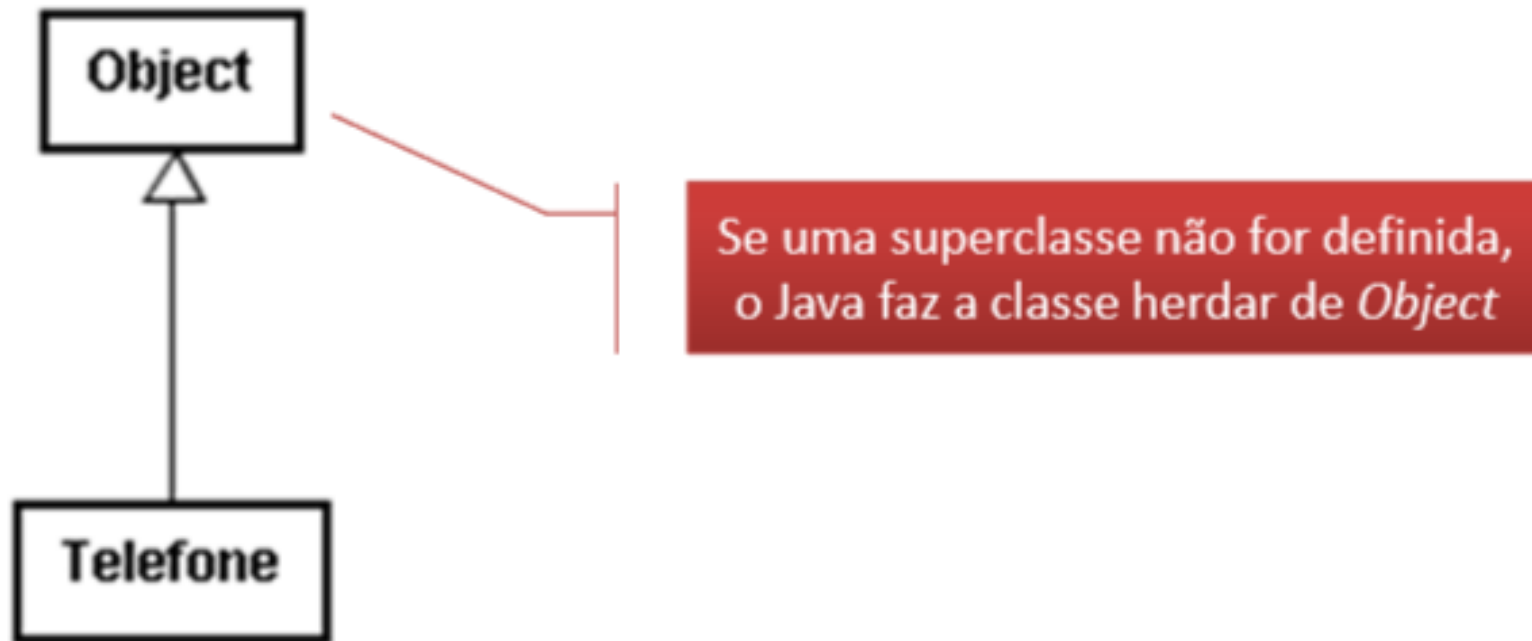
- Classe: Veículo Terrestre
- Subclasse: Moto, Caminhão, Passeio

Programando a Herança



Herança da Classe Object

- Toda classe em Java herda de apenas uma superclasse



Herança - Exemplo

```
class CarroNormal {  
    boolean ligado;  
    public void liga() {  
        ligado = true;  
    }  
    public void desliga() {  
        ligado = false;  
    }  
}
```

Herança - Exemplo

```
class CarroTurbo extends CarroNormal {  
    boolean turbo;  
  
    public void ligaTurbo() {  
        turbo = true;  
    }  
  
    public void desligaTurbo() {  
        turbo = false;  
    }  
}
```

Herança - Exemplo

```
class appCarros {  
    public static void main(String args[]) {  
        CarroNormal carro1;  
        CarroTurbo carro2;  
        carro1 = new CarroNormal();  
        carro2 = new CarroTurbo();  
        //carro1 não tem os atributos e métodos de carro turbo  
        carro1.liga();  
        //carro2 tem atributos e métodos de carro normal e de  
        carro turbo  
        carro2.liga();  
        carro2.ligaTurbo();  
    }  
}
```


Herança - Construtores

- Nós já vimos que uma classe herda de sua classe pai as suas características porém a subclasse não herda o construtor, isto é, o construtor é individual para cada classe.
- Sempre que o construtor de uma classe é invocado ele invoca, implicitamente o construtor da superclasse, isto é, de forma automática. Não precisamos nos preocupar com isso.

Herança - Construtores

- Se nós criarmos um construtor com parâmetros e estendermos(herdarmos) a classe, obrigatoriamente teremos que, explicitamente, invocar o construtor da superclasse passando os parâmetros.

Herança – Ex. Construtores

```
public class CarroNormal {  
    boolean ligado;  
    String marca;  
    public CarroNormal(String marca) {  
        this.marca = marca;  
    }  
    public void liga() {  
        ligado = true;  
    }  
    public void desliga() {  
        ligado = false;  
    }  
}
```

Herança – Ex. Construtores

```
class CarroTurbo extends CarroNormal {  
    public CarroTurbo(String marca) {  
        super(marca) ;  
    }  
    boolean turbo;  
    public void ligaTurbo() {  
        turbo = true;  
    }  
    public void desligaTurbo() {  
        turbo = false;  
    }  
}
```

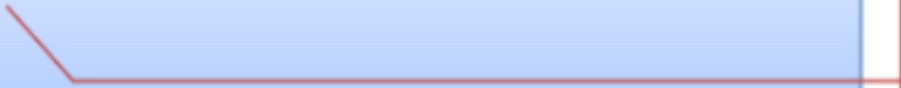
Herança – Ex. Construtores

```
class appCarros {  
    public static void main(String args[]) {  
        CarroNormal carro1;  
        CarroTurbo carro2;  
        carro1 = new CarroNormal("Marca X");  
        carro2 = new CarroTurbo("Marca Y");  
        //carro1 não tem os atributos e métodos de carro turbo  
        carro1.liga();  
        //carro2 tem atributos e métodos de carro normal e de  
        carro turbo  
        carro2.liga();  
        carro2.ligaTurbo();  
    }  
}
```

O Modificador protected


- Atributos e métodos declarados com o modificador protected podem ser acessados pelas suas subclasses

```
class Telefone {  
    protected String numero;  
    ...  
}
```



O atributo é
declarado como
protected na
superclasse

```
class Celular extends Telefone {  
    public void adicionarDDD(String ddd) {  
        String n = ddd + this.numero;  
    }  
}
```



Métodos da
subclasse possuem
acesso ao atributo
declarado na
superclasse

Sobrescrita de Métodos

- Técnica também conhecida como overriding
- Quando uma classe herda de outra, ela pode redefinir métodos da superclasse, isto é, sobrescrever métodos
 - Os métodos sobrescritos substituem os métodos da superclasse
 - A assinatura do método sobrescrito deve ser a mesma do método original

Sobrescrita de Métodos

```
class Telefone {  
    public void telefonar() {  
        //código para telefonar  
    }  
}
```

```
class Orelhao extends Telefone {  
    public void telefonar() {  
        //código para telefonar do orelhão  
    }  
}
```

```
Orelhao o = new Orelhao();  
o.telefonar();
```



Como o método foi sobrescrito, é chamado o método da subclasse

Sobrescrita de Métodos

```
class Telefone {  
    public void telefonar() {  
        //código para telefonar  
    }  
}
```

```
class Orelhao extends Telefone {  
    public void telefonar(int numero) {  
        //código para telefonar do orelhão  
    }  
}
```

```
Orelhao o = new Orelhao();  
o.telefonar();
```


Não há sobrescrita de método. Métodos sobrescritos devem ter a mesma assinatura (tipo de retorno, nome do método e parâmetros)



Usando o super

- O método que foi sobrescrito pode ser acessado pelo método que o sobrescreveu através da palavra-chave super

```
class Orelhao extends Telefone {  
    public void telefonar() {  
        super.telefonar()  
    }  
}
```

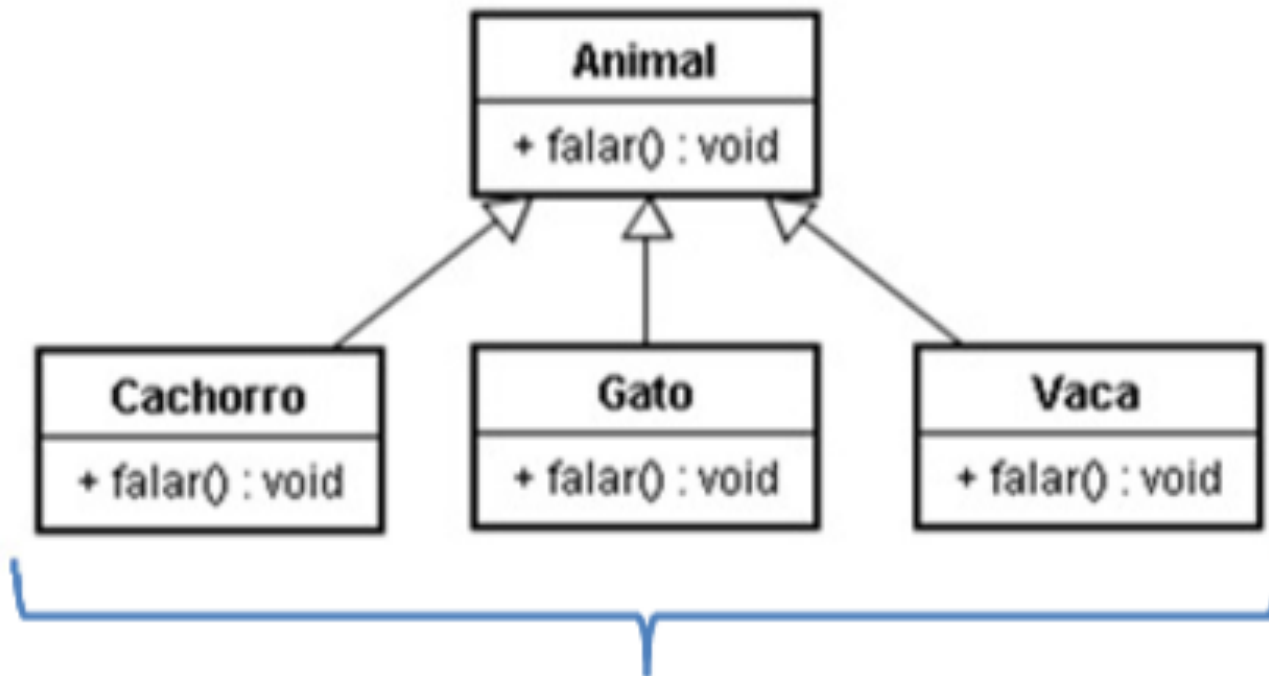


Chama o método
da superclasse

Polimorfismo

- É a capacidade que um método tem de agir de diferentes formas, dependendo do objeto sobre o qual está sendo chamado
- Quando ocorre a chamada de um método, a JVM decide qual método invocar dependendo do objeto instanciado na memória

Exemplo de Polimorfismo



As subclasses sobreescrevem o método *falar()*

Exemplo de Polimorfismo

```
class Animal {  
    public void falar() {  
    }  
}
```

```
class Cachorro extends Animal {  
    public void falar() {  
        System.out.println("Au");  
    }  
}
```

```
class Gato extends Animal {  
    public void falar() {  
        System.out.println("Miau");  
    }  
}
```

```
class Vaca extends Animal {  
    public void falar() {  
        System.out.println("Mu");  
    }  
}
```

Cada animal implementa o método *falar()* do seu modo

Exemplo de Polimorfismo

```
Animal a = new Cachorro();  
a.falar();
```

Resultado: "Au"

```
Animal a = new Gato();  
a.falar();
```

Resultado: "Miau"

```
Animal a = new Vaca();  
a.falar();
```

Resultado: "Mu"

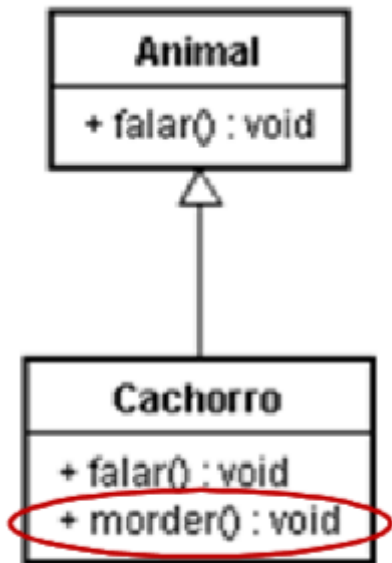
O método invocado é determinado pelo tipo do objeto que está armazenado na memória

```
Cachorro c = new Cachorro();  
Animal a = (Animal) c;  
a.falar();
```

Resultado: "Au"

A forma como objeto é referenciado não influencia na decisão sobre qual método será invocado

Exemplo de Polimorfismo



```
Animal a = new Cachorro();
a.falar();
```

Resultado: "Au"

```
Animal a = new Cachorro();
a.morder();
```

Método
inexistente

```
Animal a = new Cachorro();
Cachorro c = (Cachorro) a;
c.morder();
```

OK

O tipo pelo qual o objeto é referenciado determina
quais métodos e/ou atributos podem ser invocados

O Operador instanceof

- Utilizado para verificar se um objeto pertence à determinada classe

```
Animal a = new Cachorro();
```

```
a instanceof Cachorro
```

```
true
```

```
a instanceof Animal
```

```
true
```

```
a instanceof Gato
```

```
false
```

```
a instanceof Object
```

```
true
```

Normalmente é utilizado antes de realizar um cast, para garantir que a operação é válida

O Operador final

- O modificador final quando utilizado em classes ou métodos bloqueia, respectivamente, a extensão e a sobreescrita, isto é, utilizamos este modificador quando não desejamos que nossa classe tenha subclasses ou o método seja sobreescrito.
- Devido às suas características este modificador deve ser utilizado com parcimônia e sabedoria pois ele vai contra um dos pilares da orientação a objetos: Reuso de código através da herança e/ou sobreescrita.

O Operador final

```
public final class Jipe extends
    Veiculo{

    public void ligarTracao4x4() {
        //Código aqui
    }

    public void buzinar() {
        System.out.println("Buzina jipe");
    }
}
```

O Operador final

```
public class Jipe extends Veiculo{  
    public void ligarTracao4x4() {  
        //Código aqui  
    }  
  
    public final void buzinar() {  
        System.out.println("Buzina  
jipe");  
    }  
}
```

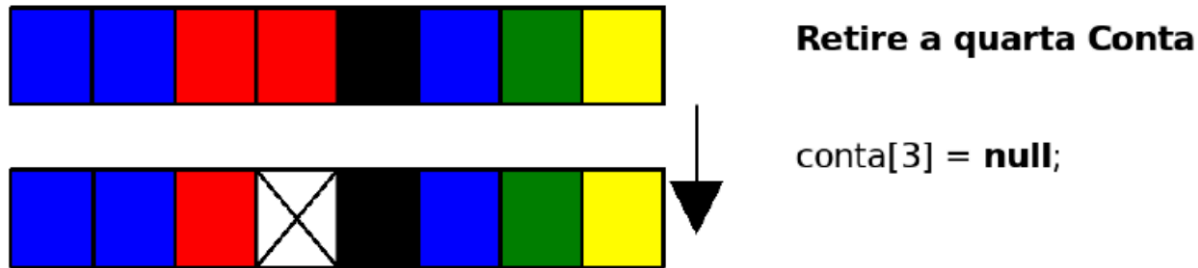
Polimorfismo

- Exemplo do Livro Java – Como programar
- H.M Deitel, P. J. Deitel
- 4ª. Edição – página 456
- Exemplo vídeo game

ArrayList

- Manipular arrays (vetores estáticos) é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:
 - Não podemos redimensionar um array em Java;
 - É impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
 - Não conseguimos saber quantas posições do array já foram “populadas”

ArrayList



- Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco?
 - Precisaremos procurar por um espaço vazio?
 - Guardaremos em alguma estrutura de dados externa as posições vazias?
 - E se não houver espaço vazio?
 - Teríamos de criar um array maior e copiar os dados do antigo para ele?

ArrayList

- Inserção e remoção de elementos dinamicamente
- ArrayList não é um array: internamente, ela usa um array como estrutura para armazenar os dados
 - é uma lista de objetos
- Para criar um ArrayList, basta chamar o construtor:

```
import java.util.ArrayList;
```

```
ArrayList lista = new ArrayList();
```

ArrayList

- **Método add**

- recebe o objeto a ser inserido e o coloca no final da lista

lista.add("Manoel");

- 2. permite adicionar o elemento em qualquer posição da lista

lista.add(2,"Manoel");

ArrayList

- Todos os métodos trabalham com Object.

```
ContaCorrente c1 = new ContaCorrente();  
c1.deposita(100);
```

```
ContaCorrente c2 = new ContaCorrente();  
c2.deposita(200);
```

```
ArrayList contas = new ArrayList();  
contas.add(c1);  
contas.add(c2);
```

ArrayList

- Para saber quantos elementos há na lista
`System.out.println(contas.size());`
- Para remover itens da lista, informar a posição
`contas.remove(3)//começa do zero`
- Para ler dados do ArrayList
`Conta c1 = contas.get(2) //posição`

ArrayList

- Em qualquer lista, é possível colocar qualquer Object. Com isso, é possível misturar objetos

```
ContaCorrente cc = new ContaCorrente();  
ArrayList lista = new ArrayList();  
lista.add(cc); //objeto ContaCorrente  
lista.add("Uma String"); //objeto String
```

ArrayList

- Mas e depois, na hora de recuperar esses objetos? Como o método get devolve um Object, precisamos fazer o cast.

Conta c1 = (**Conta**) contas.get(2)

- Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples...

ArrayList

```
ArrayList<ContaCorrente> lista = new  
ArrayList<ContaCorrente>();
```

```
lista.add (c1);
```

```
lista.add (c2);
```

```
lista.add (c3);
```

- Indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo ContaCorrente.
- Elimina a necessidade de casting

ArrayList

Usando **Iterator** para percorrer uma lista

```
import java.util.Iterator;

Iterator <ContaCorrente> it = lista.iterator();

while (it.hasNext())
{
    ContaCorrente cc = it.next();
    cc.deposita(500);
}
```

Bibliografia

- Site Java Starter - www.t2ti.com – acessado em 24/04/2015
- Curso Fundamentos de Java oferecido pela Softblue: www.softblue.com.br – acessado em 20/04/2015