Computação II – Orientação a Objetos

Fabio Mascarenhas - 2014.1

http://www.dcc.ufrj.br/~fabiom/java

Tipos Parametrizados ou Genéricos

- Os tipos parametrizados ou genéricos são a solução para esse problema

- Podemos *parametrizar* nossa lista pelo *tipo do elemento*, declarando um *parâmetro de tipo* na sua definição:

TIPO ELEMENTO

```
public interface Lista<E> {
    int tamanho();
    E le(int i);
    void escreve(int i, E s);
    void adiciona(E s);
}
```

- `E` é o parâmetro de tipo que indica o tipo dos elementos da lista; por convenção, usamos nomes curtos e em maiúsculas, mas poderia ser qualquer nome

Implementando Lista<E>

- Uma maneira simples de implementa Lista<E> é com a classe a seguir:

```
public class ListaVetor<E> implements Lista<E> {
    private ArrayList<E> al = new ArrayList<E>();
    public int tamanho() {
        return al.size();
    }
    public E le(int i) {
        return al.get(i);
    }
    public void escreve(int i, E s) {
        al.set(i, s);
    }
    public void adiciona(E s) {
        al.add(s);
    }
}
```

Subtipagem

- Duas instâncias do mesmo tipo paramétrico não são subtipo uma da outra se seus parâmetros são diferentes, mesmo que um parâmetro seja subtipo do outro

- `Lista<String>` **não é** subtipo de `Lista<Object>`!

- É fácil ver por quê:

```
Lista<String> ls = new ListaVetor<String>();
Lista<Object> lo = ls; // ok se Lista<String> é
                       // subtipo de Lista<Object>
lo.add(10); // ok pois 10 é um Object
String s = ls.le(0); // oops...
```

Subtipagem e Coringas

- `Lista<`*`Tipo`*`>` é subtipo de `Lista<?>` para qualquer *`Tipo`*
  - Não podemos chamar métodos em `Lista<?>` que recebem `?`, e `?` tem tipo `Object` quando chamamos métodos que retornam `?`
- Lista<*Tipo1*> é subtipo de Lista<? extends *Tipo2*> se *Tipo1* é subtipo de *Tipo2*
  - Como acima, mas `?` tem tipo `Tipo2` quando um método retorna `?`
- Lista<*Tipo1*> é subtipo de Lista<? super *Tipo2*> se *Tipo2* é subtipo de *Tipo1*
  - Podemos chamar métodos que recebem `?` com subtipos de `Tipo2`, e `?` Tem tipo `Object` quando chamamos métodos que retornam `?`

Métodos Genéricos

- Coringas não ser suficientes para escrever métodos genéricos; vamos pensar em um método coleta em `Lista<E>`, que recebe uma `Funcao` e retorna nova lista que é o resultado de passar todos os elementos da lista atual por essa `Funcao`:

```
Lista<?> coleta(Funcao<E,?> f);
```

- No final perdemos o tipo que a função está produzindo! Precisamos de um parâmetro para ele, mas se usar `E` é muito restritivo, então podemos dar um parâmetro pro método:

```
<S> Lista<S> coleta(Funcao<E,S> f);
```

Limites nos Parâmetros

- Não são apenas coringas que podem ter um limite com `extends`, parâmetros de tipo normais também podem ter

- Um limite permite chamar os métodos do tipo que damos como limite, e passar uma referência do tipo do parâmetro para métodos que esperam o tipo do limite:

```
public class ListaFigura<E extends Figura> extends ListaVetor<E>
{
    public void desenhar() {
        for(int i = 0; i < tamanho(); i++) {
            le(i).desenhar();
        }
    }
}
```

Laço `for` de coleções

- Java tem uma versão do laço for que é especializada para percorrer uma coleção (um vetor, ou instâncias de uma das classes de coleção como `HashSet` e `ArrayList`)

- O bloco do laço é executado para cada elemento da coleção, com a variável de controle apontando para o elemento

```
for(int i: vetor) {
    System.out.println(i);
}
```

Iterable<E> e Iterator<E>

- Qualquer classe pode ser usada com um for de coleções, contanto que implemente a interface parametrizada Iterable<E>:

```
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

- A interface `Iterator<E>` é uma versão genérica da nossa `Iterador`:

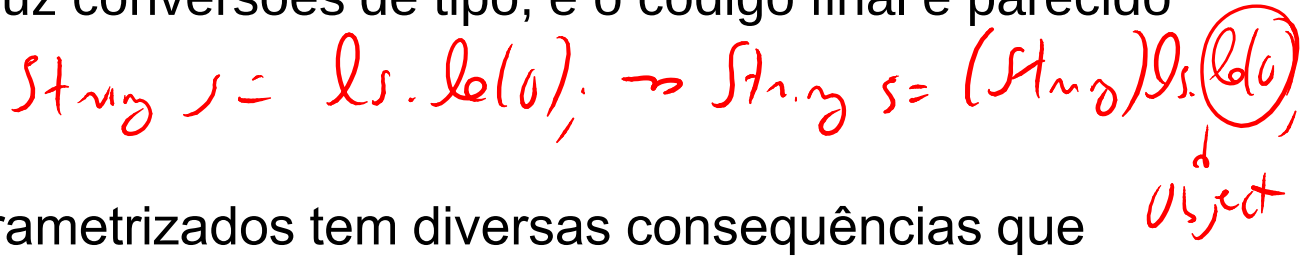
```
public interface Iterator<E> {
  boolean hasNext();
  E next(); // lança NoSuchElementException
    void remove();
}
```

Sutilezas dos Tipos Genéricos

- Tipos genéricos só existem em tempo de compilação, não fazem parte da implementação de Java

- Em geral, parâmetros são convertidos para `Object` pelo compilador, a não ser que ele tenha um limite com `extends`

- O compilador também introduz conversões de tipo, e o código final é parecido com o da “lista de objetos”

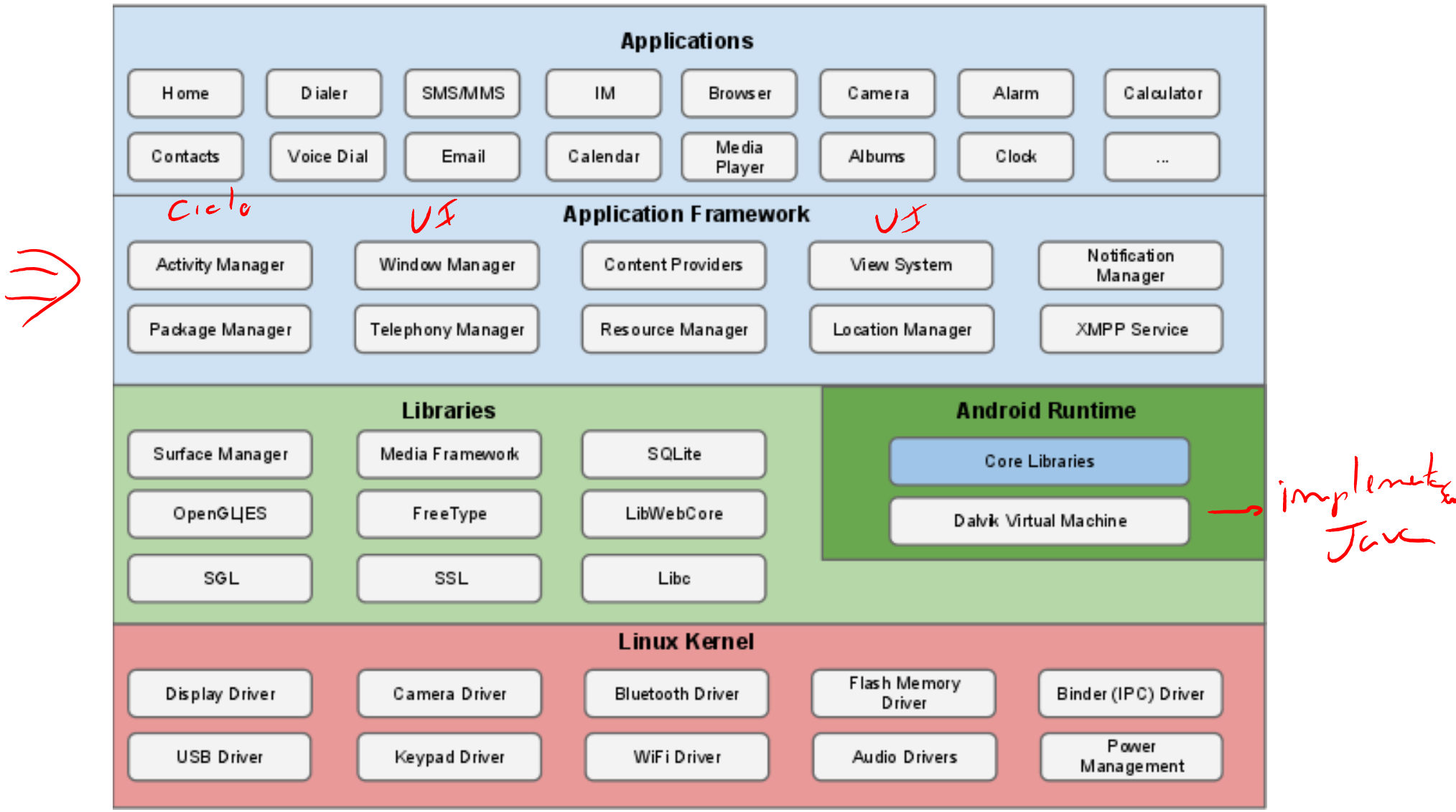


- Essa “deleção” dos tipos parametrizados tem diversas consequências que estão descritas na documentação

Android

- Android é um sistema operacional para dispositivos móveis
  - *Kernel* Linux, drivers e bibliotecas do sistema, *frameworks* de aplicação (Android Software Development Kit, Android Native Development Kit) e aplicações embutidas
- Aqui estaremos interessados no Android SDK, um framework para desenvolvimento de aplicações para Android na linguagem Java
- O sistema Android possui farta documentação

Arquitetura do SO Android



Java no Android

- O sistema Android não usa a implementação oficial de Java, mas a sua própria, equivalente à versão 1.6 da implementação oficial

- A maior parte das classes nos pacotes `java.*` e `javax.*` estão presentes

- Classes específicas do sistema Android estão nos pacotes `android.*`, e classes úteis para aplicações Android estão nos pacotes `org.*`

- Outras bibliotecas Java geralmente funcionam sem modificações, mas às vezes dependem de partes que não estão no Android, mesmo que possuam equivalentes

Android Development Tools

- O ADT é um pacote que junta todo o necessário para desenvolver em Android:
  - O Android SDK
  - O *emulador* Android e uma *imagem* para o emulador
  - Uma versão do Eclipse específica para programar para Android
- Baixe o .zip do ADT nesse link (cerca de 500Mb!)
- Deixe ele descompactado em um pen drive, para podermos usar no laboratório

Hello, Android

- Vamos criar e rodar uma aplicação
- Uma aplicação Android já vem com muita estrutura, pois ela está inserida em um framework bastante complexo
- Ela já está separada em um *controlador* (a classe principal) e uma *visão* (descrita em arquivos XML), e mesmo algumas partes que formam um *modelo* bem simples (no arquivo `strings.xml`)
- Para rodar essa aplicação, primeiro precisamos criar um *dispositivo virtual* no emulador (ou conectar um dispositivo real via USB)

O Emulador

- Se usarmos a imagem de sistema padrão o emulador pode ficar lento demais
- Então vamos baixar uma imagem de sistema Intel x86 no Android SDK Manager, e criar um dispositivo virtual que use essa imagem
- Ainda assim é possível que o emulador fique lento, especialmente se a resolução do dispositivo virtual for alta
- O melhor mesmo é plugar um dispositivo via USB!