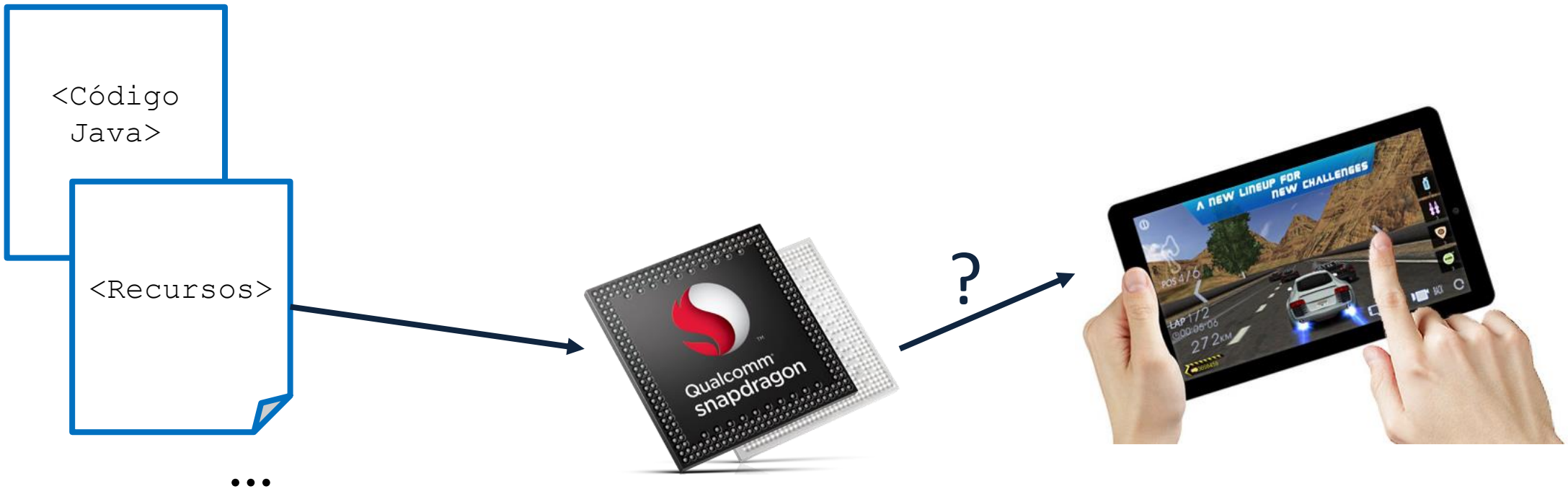
NÍVEIS DE ABSTRAÇÃO NUM COMPUTADOR

TeSP de Aplicações Móveis

André Martins Pereira

IPCA
INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

EXECUÇÃO DE PROGRAMAS



NÍVEIS DE ABSTRAÇÃO

- Permitem isolar o programador da complexidade de funcionamento de um processador
  - Níveis de abstração mais altos oferecem linguagens mais próximas do programador
  - Níveis mais baixos são o mais próximos da máquina possíveis
- Entre uma linguagem de mais alto nível e a linguagem máquina existem vários níveis
- Porquê? Porque não apenas 1 nível?
- Várias razões:
  - Otimizações
  - Compatibilidade
  - Inclusão de programas ou funções pré-complicadas
  - ...

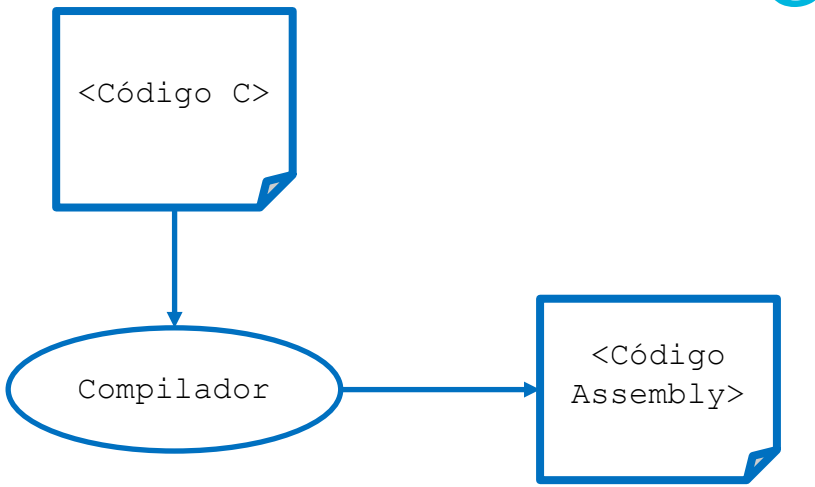
NÍVEIS DE ABSTRAÇÃO

- Do nível mais alto para o mais baixo:
  1. Linguagem de alto nível (*High-Level Language, HLL*) – C, Java, C++, …
  2. Linguagem de representação intermédia (*Intermidiate Representation, IR*) – LLVM, …
  3. Linguagem *bytecode* – JVM bytecode, Dalvik/ART bytecode, …
  4. Linguagem *assembly* – IA-32, x86, x64, … (dependem da arquitetura do CPU)
  5. Linguagem Objeto – (semelhante a *assembly*, mas com as inclusões de outros programas/funções feitas)
  6. Linguagem máquina – binário

NÍVEIS DE ABSTRAÇÃO

- Transformar uma linguagem de um nível ***i*** numa de nível ***i-1*** requer ferramentas específicas
  - Ex.: compliador, linker, assembler, ...
- Transformar "texto" de uma HLL para código máquina implica diferentes tarefas em diferentes fases, dependentes umas das outras
  - Ex.: O assembler precisa do resultado da compilação
- Cada ferramenta é responsável por uma tarefa específica
- Dependendo da linguagem HLL usada podem ser requeridas mais tarefas até chegar ao código máquina

COMPILAÇÃO (C)

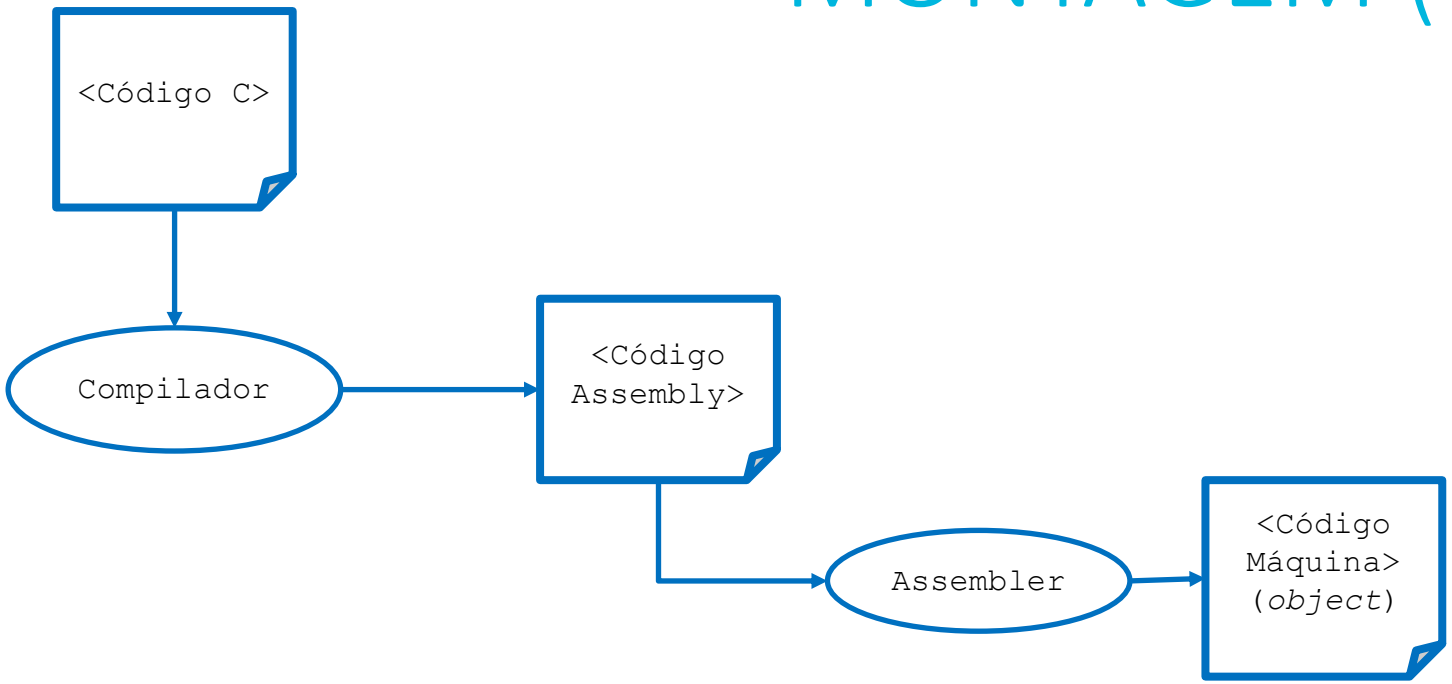


Compliador:

Traduz um programa de um nível superior de abstração (p.e., linguagem C) para outro inferior (assembly independente da máquina, tipicamente)

Normalmente inclui mais que um passo de conversão até chegar à linguagem máquina

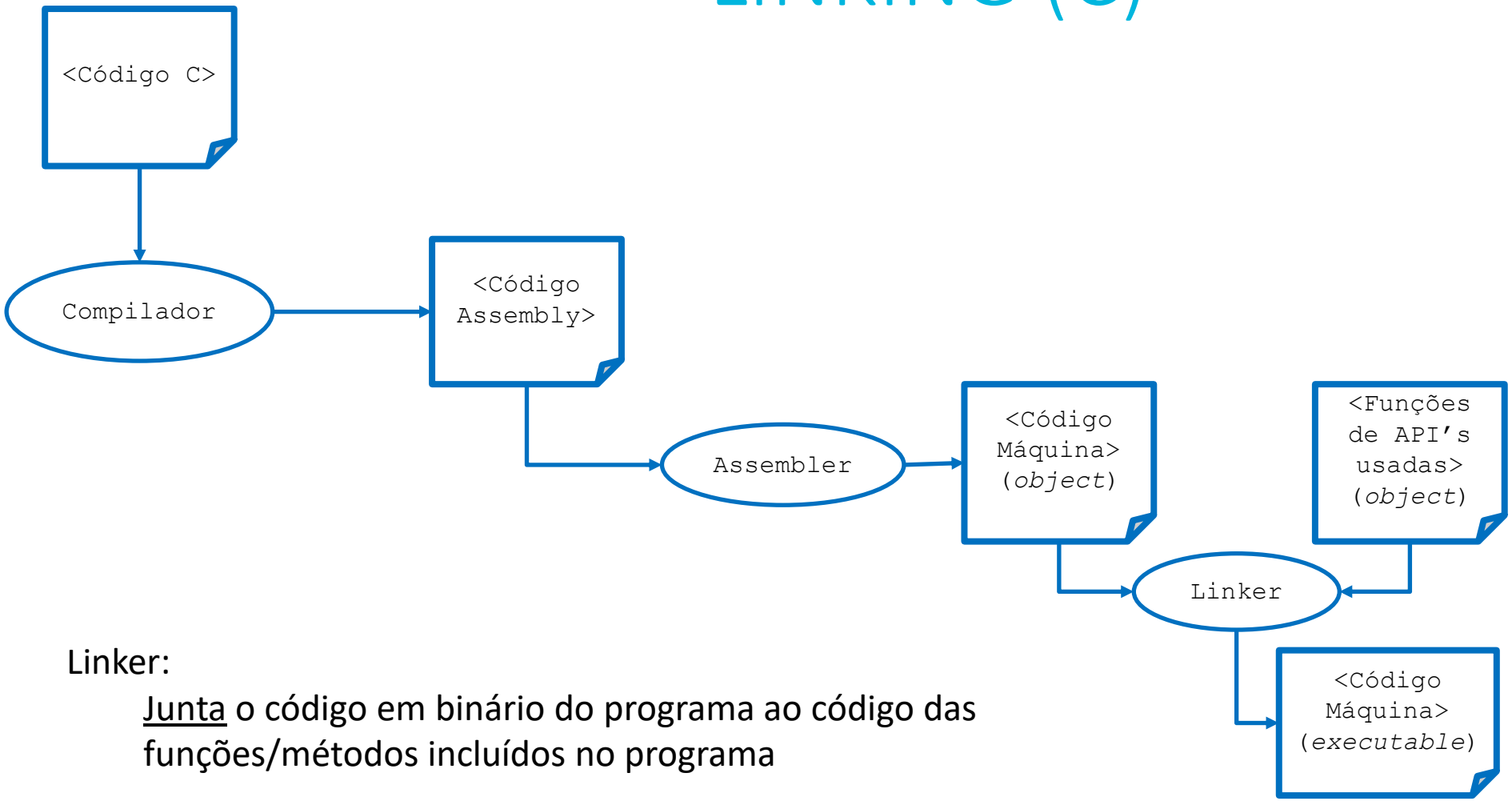
MONTAGEM (C)



Assembler:

"Monta" os comandos/instruções em binário (*object*), de acordo com as regras do CPU

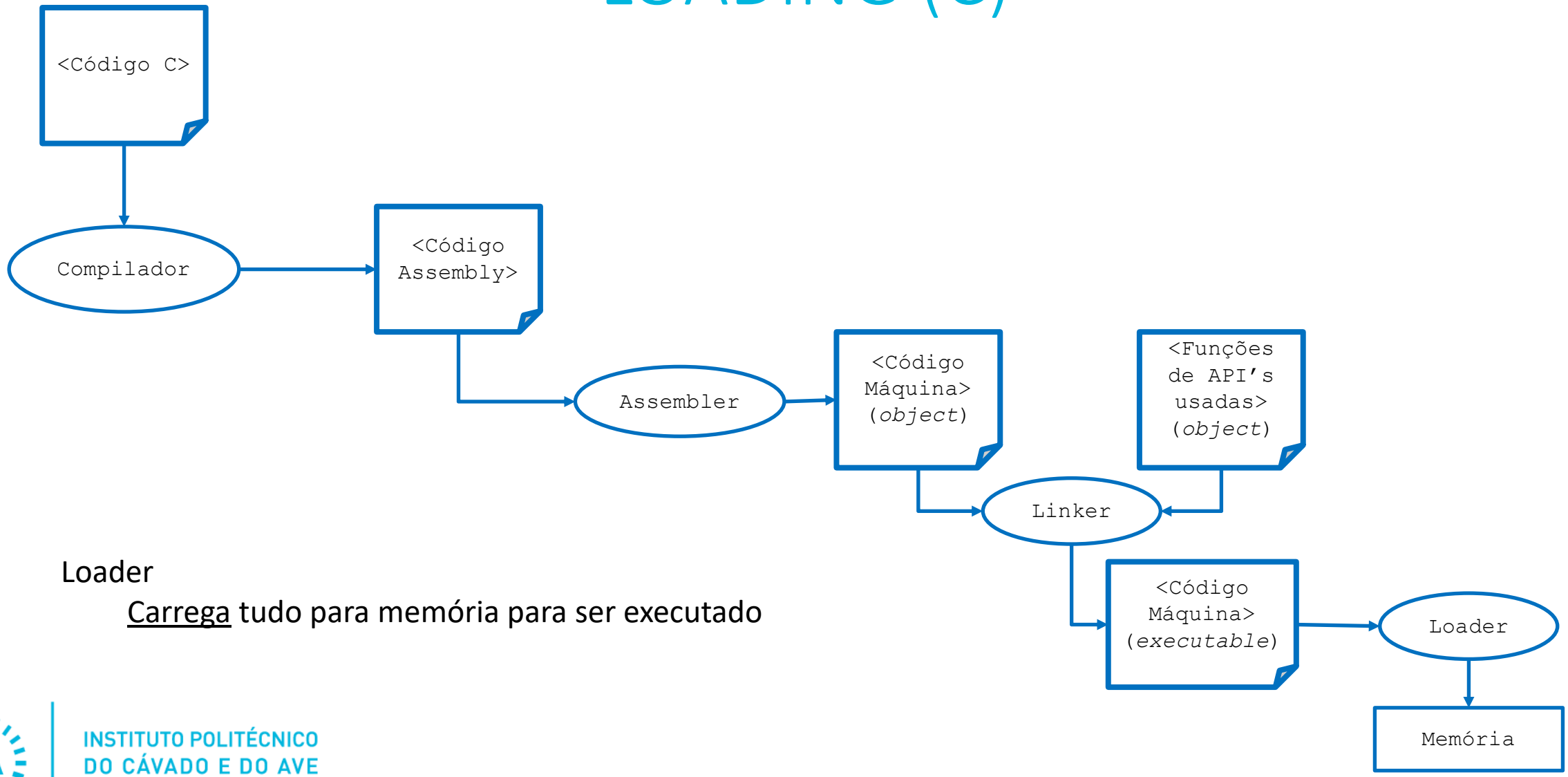
LINKING (C)



Linker:

Junta o código em binário do programa ao código das funções/métodos incluídos no programa

LOADING (C)



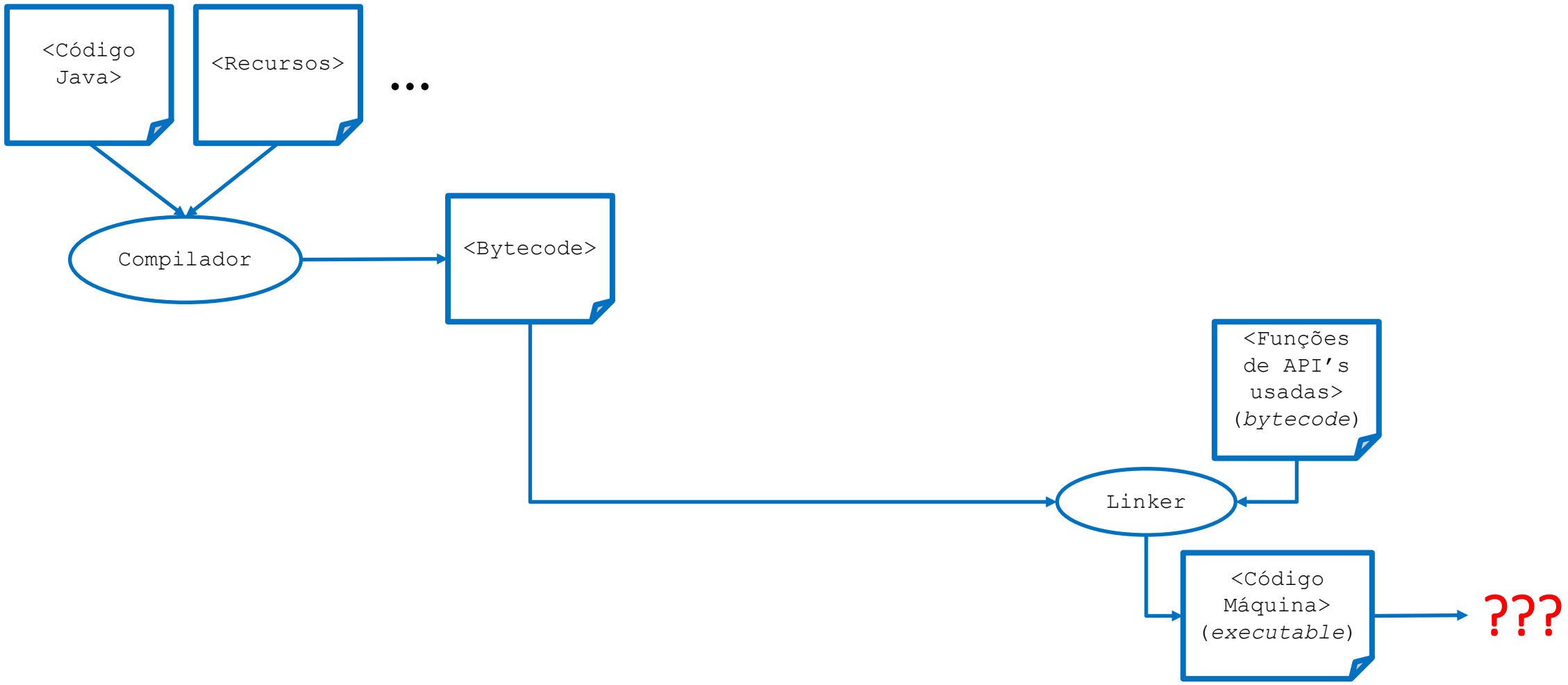
Loader

Carrega tudo para memória para ser executado

DA COMPILAÇÃO À EXECUÇÃO

- No exemplo anterior, o processo de compilação e execução de um programa em **C** passou por 4 fases
- É sempre assim com esta linguagem? **Não!**
- E com outras linguagens? O que muda? **Depende…**

JAVA PARA ANDROID



JAVA PARA ANDROID

- A linguagem *Java* funciona ligeiramente diferente
- O resultado final (ficheiro executável) não é carregado para memória
  - Não tem código máquina/objeto
- Existe uma **Máquina Virtual** (essa sim em memória) que executa as instruções desse ficheiro
  - Lê uma a uma
  - Interpreta o que ela faz
  - Executa o que é suposto
- Não é a mesma coisa? **Não!**
- Qual a diferença?
- Porque funciona assim?

MÁQUINA VIRTUAL VS EXECUÇÃO NATIVA

- Com máquina virtual, o mesmo código compilado dá para qualquer máquina
  - Quem vai interpretar e executar o código é uma "aplicação" (Dalvik, ART, JavaVM, …)
  - Essa máquina virtual transforma o *bytecode* num executável (aplicação) tendo em conta as características da máquina física onde vai correr
  - Por outras palavras, a máquina virtual transforma cada instrução *bytecode* na correspondente instrução em código máquina no processador em que executa
- Funciona como um intermediário entre código e processador
  - O processador não executa diretamente uma versão em *bytecode* do programa
- Desvantagem: máquina virtual gasta mais recursos (memória e cpu)
  - Mais lento a compilar
  - Dificulta a utilização de otimizações do processador

MÁQUINA VIRTUAL VS EXECUÇÃO NATIVA

- Mas, no fundo, mesmo com máquina virtual, o processador acaba por executar o mesmo tipo de instruções, apenas não o faz diretamente
  - Operações com números (soma, divisão, subtração, comparação, …)
  - Instruções de salto (*Jump*)
  - Acessos a memória
  - Chamadas e retornos de funções (*call*, *ret*, …)
  - …

ANDROID RUNTIME

- Dalvik VM
  - Máquina Virtual responsável por executar as aplicações
    - Depois de escritas, compiladas e instaladas
  - Otimizada para minimizar consumo de memória
    - Máquinas virtuais Java normalmente usam *stacks* de memória
    - Dalvik é uma máquina virtual baseada em registos do processador
    - Dispositivos normalmente têm falta de RAM, quando comparados a processadores
    - Dalvik usa um conjunto de instruções mais pequeno (para usar menos memória); Porém, mais complexo
  - Segundo a Google, este design permite um dispositivo correr várias instâncias da máquina virtual eficientemente

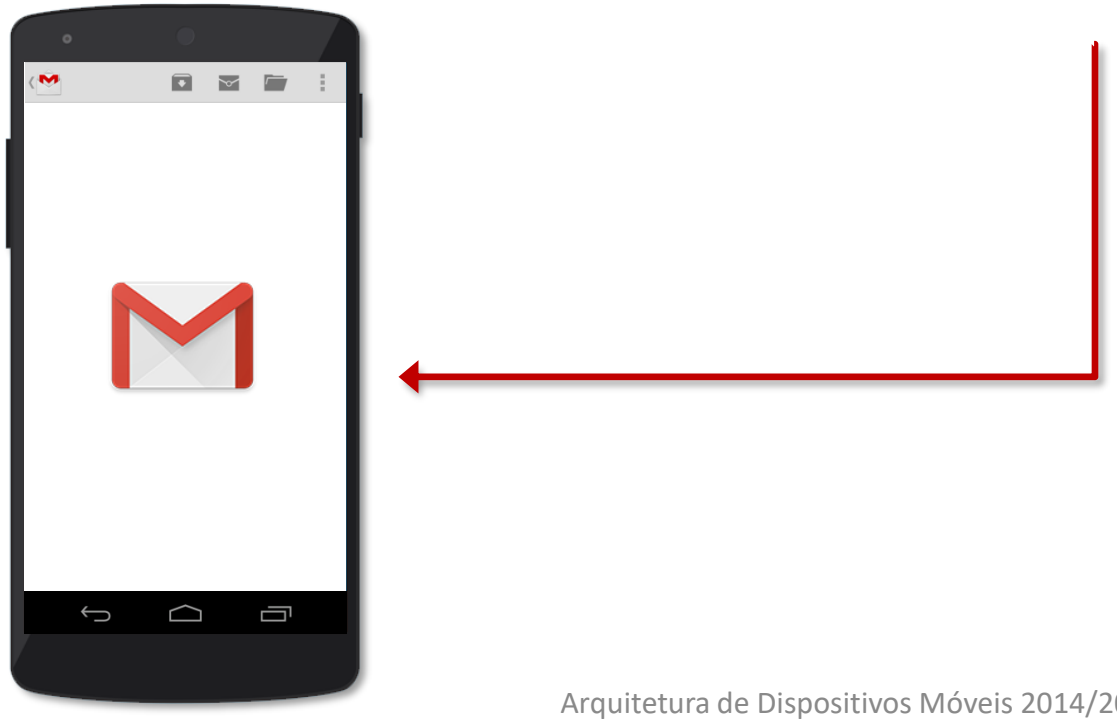
ANDROID RUNTIME

- Como é que funciona a Dalvik VM?
  1. Programadores escrevem código Java;
  2. Compilador transforma código Java em bytecode
     - Do tipo JVM (Java Virtual Machine)
  3. Ferramenta **dx** traduz este bytecode JVM para bytecode Dalvik
     - Guardado em ficheiro **.dex**
  4. Todos os ficheiros são guardados num ficheiro compacto (**.apk**)
     - Seguindo uma organização própria
  5. Depois de instalar o ficheiro **.apk**, a Dalvik VM executa as instruções dos ficheiros **.dex**

ANDROID RUNTIME

</> —javac→ .class —dx→ .dex —packager→ .apk —installer→ APP

Programadores | Bytecode JVM | Bytecode Dalvik | Android package | Aplicação)

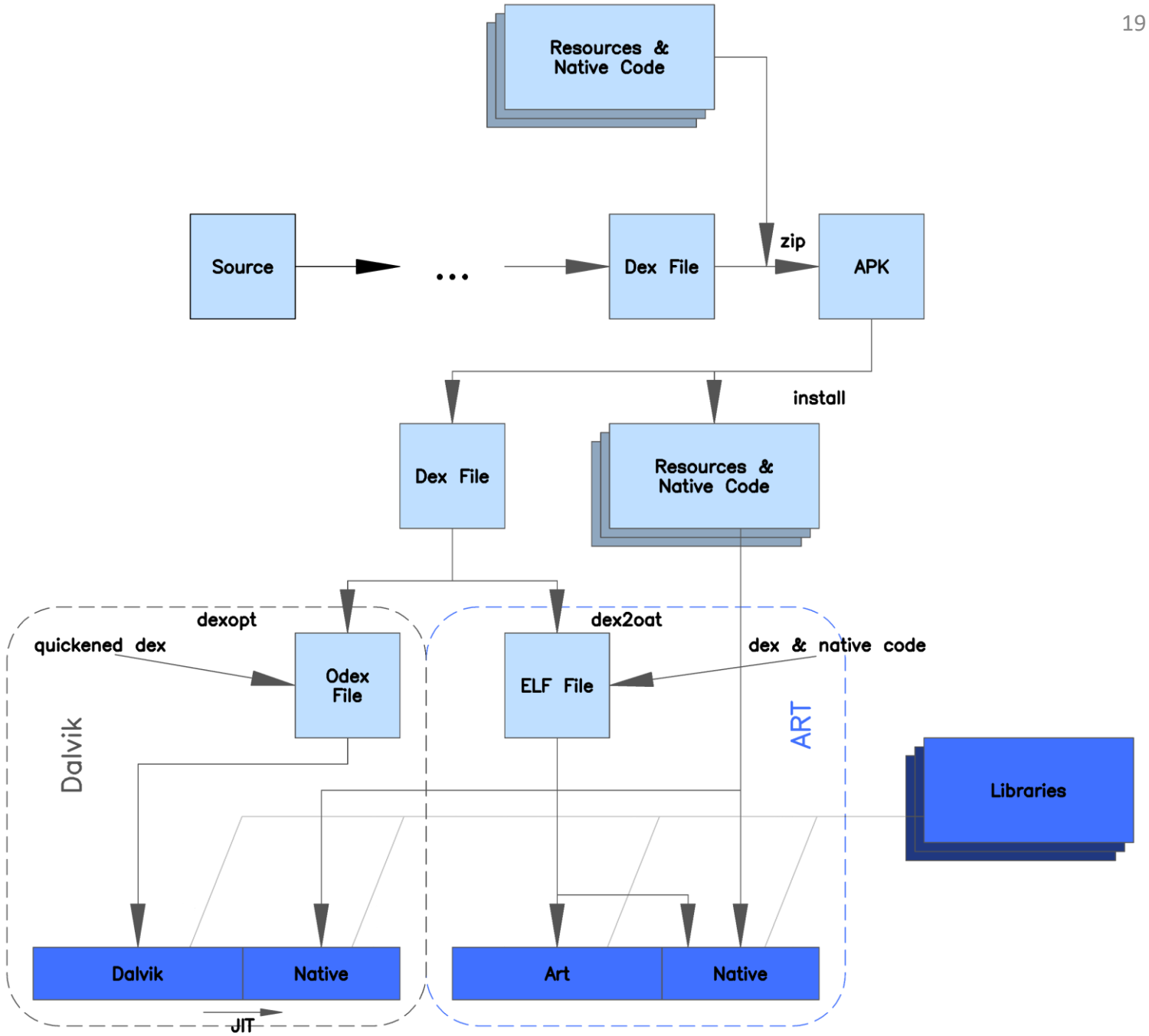


ANDROID RUNTIME

- Na versão 4.4 (KitKat) foi incluída uma alternativa: *Android Runtime* (ART)
- Na versão 5.0 (Lollipop), ART substitui inteiramente Dalvik
- Motivo:
  - Davlik: just-in-time (JIT) compilation;
    - Cada vez que uma aplicação é lançada, compila-se o bytecode
  - ART: ahead-of-time (AOT) compilation
    - Bytecode compilado apenas durante a instalação
  - Uso de processador reduzido; Duração da bateria aumenta
  - "Contratempo": aplicações demoram mais tempo a instalar

ANDROID RUNTIME

ART vs Dalvik



NÍVEIS DE ABSTRAÇÃO NUM COMPUTADOR