Поддержка Java 8 в Excelsior JET

Никита Липский

Excelsior LLC

EXCELSIOR

про себя

- Более 20 лет профессиональной карьеры
- Инициатор проекта Excelsior JET
  - работал над проектом более 16 лет
  - как идейный вдохновитель
  - компиляторный инженер
  - руководитель
  - и много в каких еще ролях
- Open source проекты WebFX и Java ReStart
  - в свободное от работы время
- twitter: @pjBooms

EXCELSIOR

про Excelsior JET

- Полная реализация Java SE
  - с 2005 года сертифицирована как Java Compatible
- AOT compiler + Java Runtime
  - смешанная компиляция: AOT + JIT
  - поддержка нестандартных загрузчиков классов в AOT режиме (для Eclipse RCP, Tomcat)
- Toolkit
  - Startup Optimizer
  - Deployment

EXCELSIOR

Кто знает про Excelsior JET

EXCELSIOR

Что добавлено в Java 8

- Lambda-expressions
- Default Methods
- Stream API, Time API
- Type annotations, Parameter names
- JavaFX 8
- Nashorn
- Compact Profiles

Default methods

EXCELSIOR

Default Methods

- Методы с телами в интерфейсах
  - Мотивация: расширение Java API, не ломая обратную совместимость

```
List<Person> list = ...
list.forEach(System.out::println)
```

`forEach` нет в Java 7

EXCELSIOR

Default Methods

Повлияли на спецификацию JVM:

- Изменилась семантика инструкций:
  - Invokespecial
  - invokevirtual
  - invokeinterface
  - invokestatic

- Изменилась процедура статической инициализации класса

EXCELSIOR

Resolve метода

- Все invoke инструкции (кроме invokedynamic) ссылаются на вызываемый метод символьно

- Перед тем как invoke инструкция исполнится, должна пройти процедура разрешения символьной ссылки на метод

Resolve метода

Пусть есть ссылка C.foo():

- Ищем foo в C:
- Если не нашли, ищем foo в суперклассах C
- Если не нашли, ищем foo в суперинтерфейсах C
- **New in Java 8:**
  - при поиске метода в суперинтерфейсах, в начале ищется *максимально специфичный default метод*

EXCELSIOR

Resolve метода

Максимально специфичный default метод:

- default метод, который не переопределяется в потомках данного интерфейса, которые имплементируется классом C
- Если есть, то должен быть один
  - Иначе процедура резолва бросает IncompatibleClassChangeError

EXCELSIOR

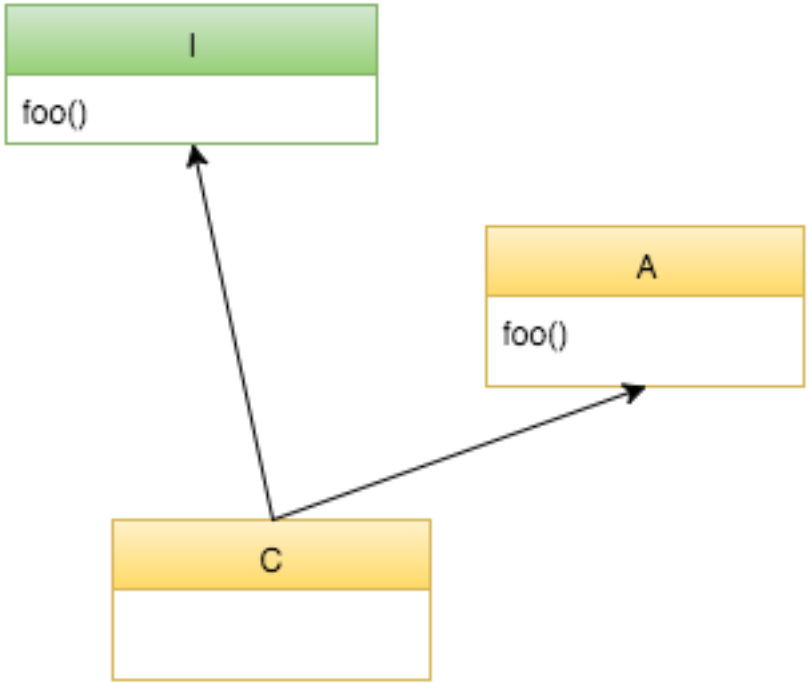
Resolve метода

```
interface I {
  default void foo(){…}
}

class A {
  void foo(){…}
}

class C extends A
        implements I{
}

C.foo() - ?
```



EXCELSIOR

Resolve метода

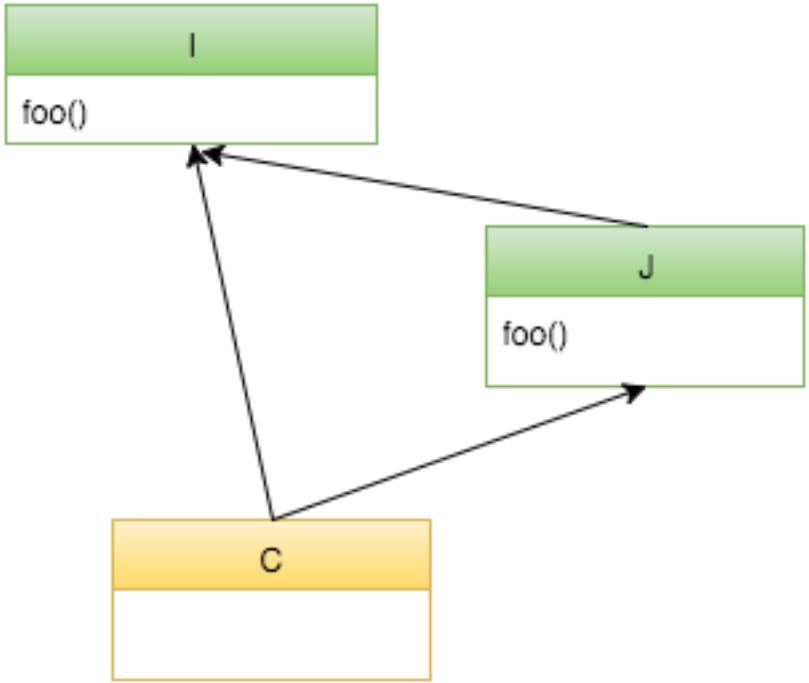
```
interface I {
  default void foo(){…}
}

interface J extends I {
  default void foo(){…}
}

class C implements J,I {

}

C.foo() - ?
```



Resolve метода

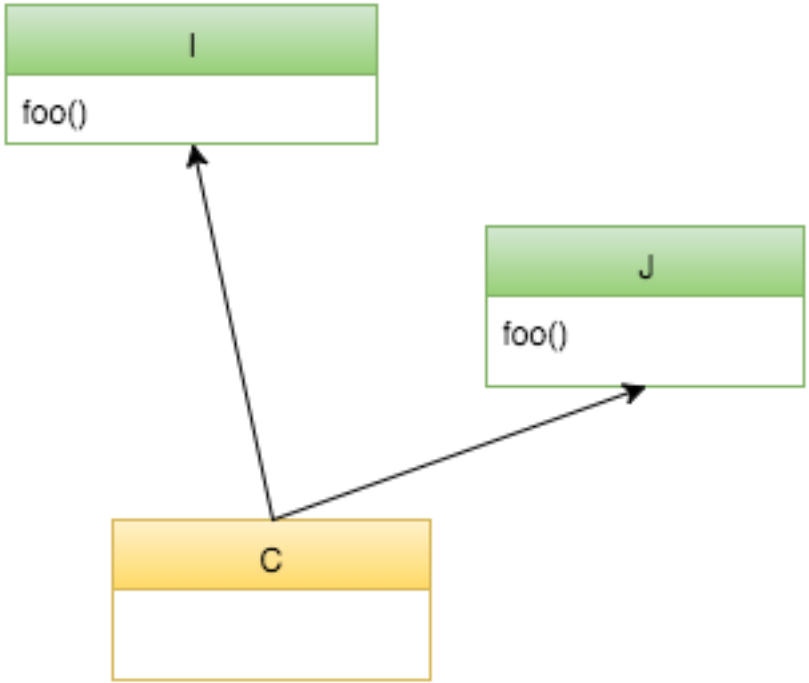
```
interface I {
  default void foo(){…}
}

interface J {
  default void foo(){…}
}

class C implements J,I {

}

C.foo() - ?
```



EXCELSIOR

invokespecial

- Используется для вызова конструкторов и суперметодов
- Целевой метод вызова статически вычислим
  - может вызываться напрямую
  - может инлайниться сразу в вызывающий метод
- Вызывает метод (чаще всего), который возвращается процедурой resolve метода
  - т.о. может вызывать default методы в Java 8

EXCELSIOR

invokespecial

- Поддержка в AOT компиляторе:
  - Resolve методов происходит до исполнения программы (но по JVM спецификации)
  - Каждый метод, каждого компилируемого класса компилируется в машинный код и имеет адрес
  - invokespecial <MethofRef> транслируется в:

    call <MethodAddress>

EXCELSIOR

invokevirtual

- Вызываемый метод зависит от объекта получателя (this)

- Транслируется в косвенный вызов через таблицу виртуальных методов

  invokevirtual <MethofRef>

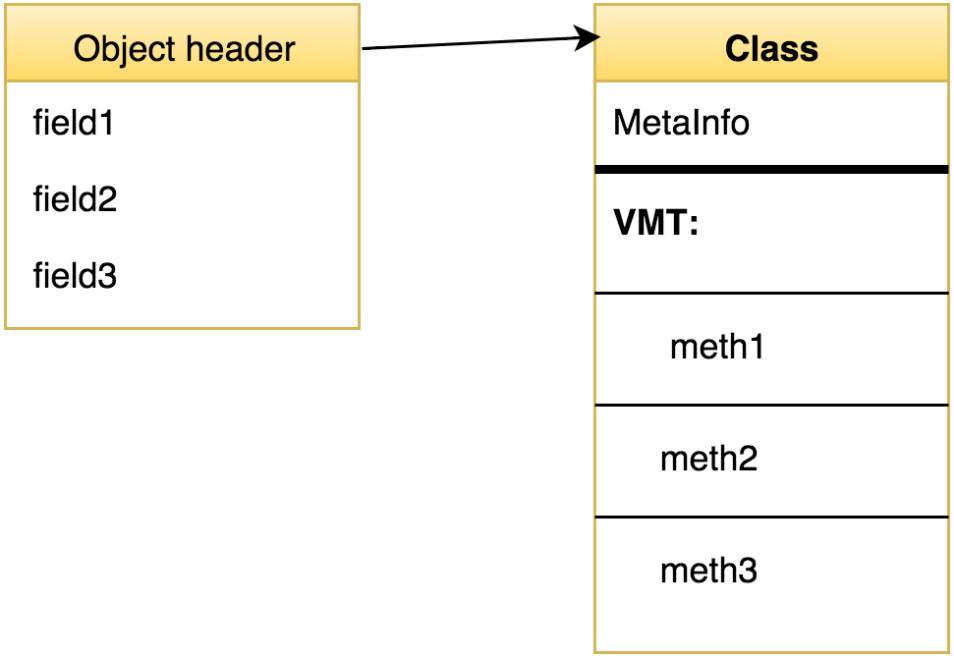
  ->

  call [<VMTAddress> +MethodSlotInVMT*C]

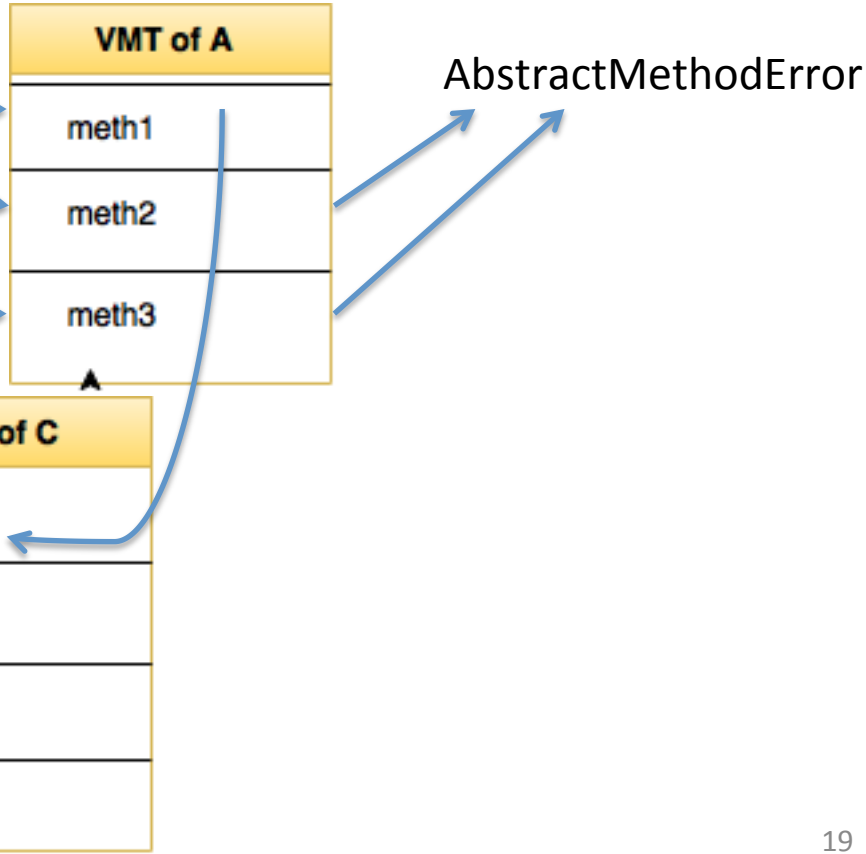
EXCELSIOR

invokevirtual

- Схема представления объекта в памяти



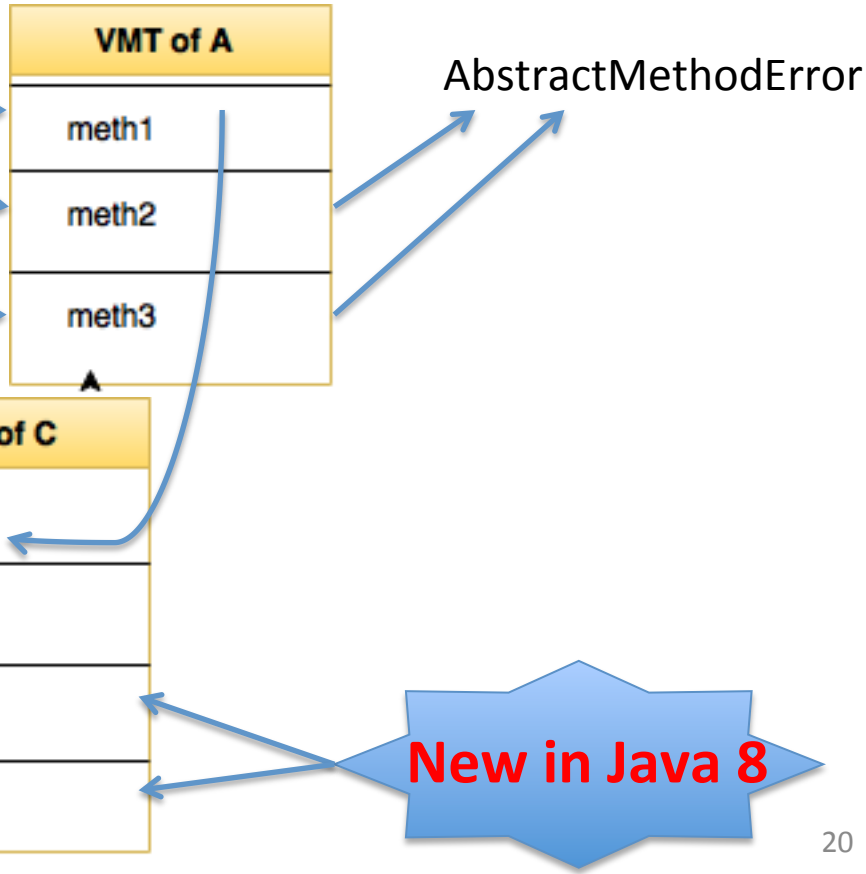
```
abstract class A
          implements I {
{
  void meth1(){…}
  abstract void meth2();
}
interface I {
  void meth3();
}
```




```
class C extends A {
    void meth2(){…}
    void meth3(){…}
    void meth4(){…}
}
```

```
abstract class A
          implements I {
{
  void meth1(){…}
  abstract void meth2();
}
interface I {
  void meth3();
}

class C extends A {
 implements J {
    void meth2(){…}
}
interface J {
   default void meth3(){…}
   default void meth4(){…}
}
```



Поддержка default методов в Excelsior JET

- Компиляция методов интерфейсов
- Изменение процедуры резолва методов
- Изменение процедуры заполнения VMT

EXCELSIOR

Найденные баги при поддержки Java 8 в Excelsior JET

- В JVM спецификации
  - override метода
    (https://bugs.openjdk.java.net/browse/JDK-8098577)
- В HotSpot
  - статическая инициализация классов в присутствии default методов инициализирует больше классов чем нужно
    (https://bugs.openjdk.java.net/browse/JDK-8098557)
  - в resolve методов: “InterfaceMethod CP entry pointing to a class should cause ICCE”
    (https://bugs.openjdk.java.net/browse/JDK-8087223)

EXCELSIOR

Lambda выражения

EXCELSIOR

Лямбда выражения

Было

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hi,I'm Thread");
    }
}).start();
```

EXCELSIOR

Лямбда выражения

Стало …

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hi,I'm Thread");
    }
}).start();
```

EXCELSIOR

Лямбда выражения

Стало

```
new Thread(()->
        System.out.println(“Hi,I’m Thread”)
).start();
```

Лямбда выражения

Как лямбда-выражения повлияли на спецификацию JVM?

EXCELSIOR

Лямбда выражения

Как лямбда-выражения повлияли на спецификацию JVM?

Никак!

EXCELSIOR

Трансляция лямбда выражений в Java байткод

```
class B {
    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        list.removeIf( p ->
            (p.size >= bottom && p.size <= top)
        );
    }
}
```

EXCELSIOR

Трансляция лямбда выражений в Java байткод

```
class B {
    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        list.removeIf(new Predicate() {
            public boolean apply(Person p) {
               (p.size >= bottom && p.size <= top);
            }
        );
    }
}
```

EXCELSIOR

Трансляция лямбда выражений в Java байткод

```
class B {
    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        list.removeIf(new Predicate() {
            public boolean apply(Object o) {
                (p.size >= bottom && p.size <= top);
            }
        );
    }
}
```

EXCELSIOR

Трансляция лямбда выражений в Java байткод

```
class B {
    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        list.removeIf(
          [lambda for lambda$1 as Predicate
              capturing (bottom, top) ]
    }

 static boolean lambda$1(int bottom, int top, Person p)
    {
        return p.size >= bottom && p.size <= top;
    }
}
```

EXCELSIOR

Трансляция лямбда выражений в Java байткод

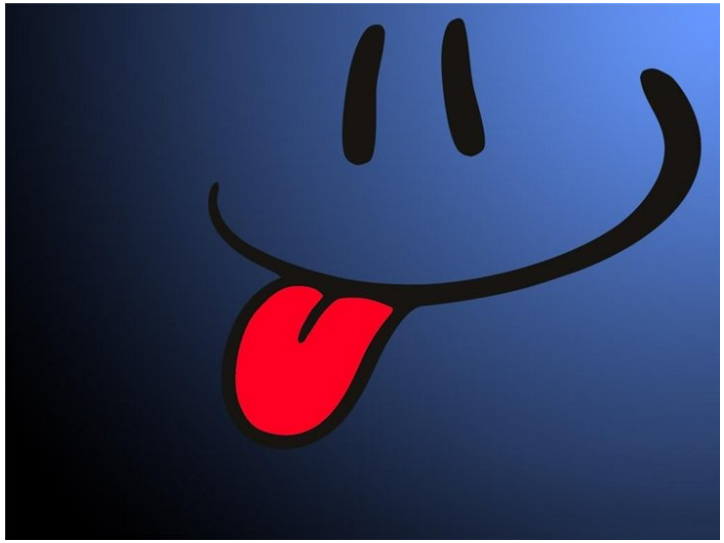
```
class B {
    public void foo() {
        List<Person> list = ...
        final int bottom = ..., top = ...;
        list.removeIf(
           indy((MH(metaFactory), //bootstrap
                 MH(invokeVirtual Predicate.apply),
           MH(invokeStatic B.lambda$1))(bottom, top))
    }

 static boolean lambda$1(int bottom, int top, Person p)
    {
         return p.size >= bottom && p.size <= top;
     }
}
```

EXCELSIOR

Лямбда-выражения

A Indy – это фича Java 7!



Indy

Indy – сокращение от invokedynamic

- В первый раз indy зовет bootstrap метод, который возвращает объект CallSite
- Далее и при последующих вызовах от объекта callsite берется MethodHandle и зовется invokeExact

MethodHandle

MethodHandle – целевой объект indy:

- может быть доступом к полю, методу
- адаптером других MethodHandle
  - адаптеры аргументов
  - частичное применение аргументов (binding)
  - условный переход (guardWithTest)

Должны работать быстро

- Все что можно выразить через MethodHandle, можно выразить через Reflection, но это неэффективно

EXCELSIOR

MethodHandle

MethodHandles с Java 7 Update 40 реализованы через lambda-forms:

- Внутреннее представление методхэндлов
- По этому внутреннему представлению динамически порождается Java байткод (Runtime анонимные классы)
- Реализация состоит на 90% из pure Java кода и переиспользуется в Excelsior JET

EXCELSIOR

Runtime анонимные классы

- Настолько анонимные, что их даже нет в JVM спецификации
  - полную семантику возможно вычислить только из исходных текстов HotSpot

Runtime анонимные классы

- Из Java создаются через

  `Unsafe.defineAnonymousClass`
- Не принадлежат какому-либо классслоадеру, их нельзя найти, нет reflection
- В Java 8 их семантика поменялась
  - они могут доступаться до private членов классов их породивших (до тел лямбд)

Вернемся к лямбдам

Задача: Пусть есть Java код

```
Runnable r =
    ()->System.out.println(“Hi”);
r.run();
```

Требуется: вместо этого кода породить:

```
System.out.println(“Hi”);
```

EXCELSIOR

Решение HotSpot

- Исполняем код метода, пока он не станет горячим
  - порождая и загружая анонимный класс-завертку для лямбда выражения
- Инлайним в код целевого метода сначала код единственного метода анонимного класса, а затем и само тело лямбды
- Анонимный класс собираем GC

EXCELSIOR

Решение JET*

- Восстанавливаем из байткода в статическом компиляторе лямбда-выражение в исходном виде
- Вычисляем, что единственное использование лямбды – это ее непосредственный вызов
- Инлайним тело лямбда выражения

* Пока не реализовано

EXCELSIOR

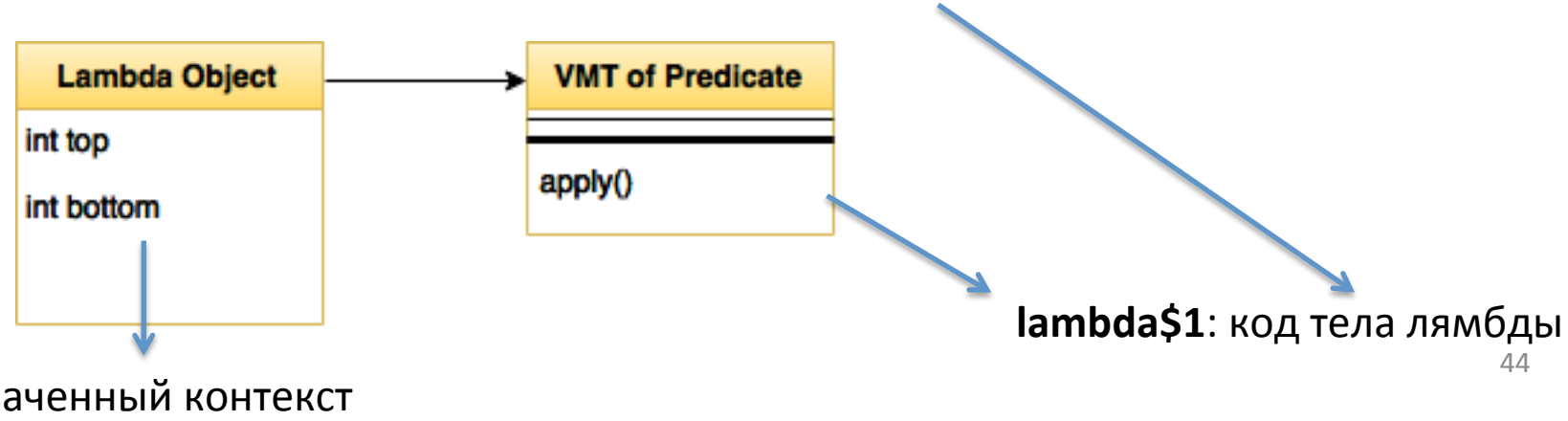
Решение JET*

А если нам не удалось заинлайнить метод куда передается лямбда-выражение?

EXCELSIOR

Решение JET*

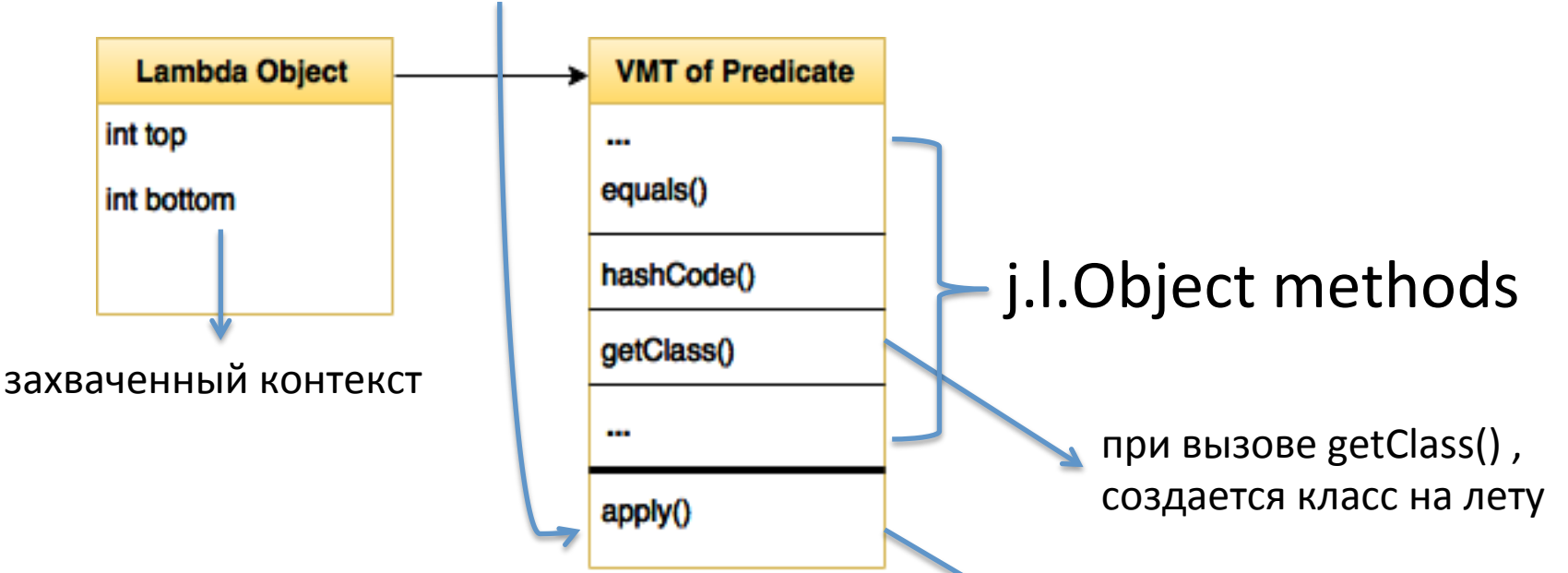
```
(p -> (p.size >= bottom && p.size <= top))
```



EXCELSIOR

Решение JET*

```
(p -> (p.size >= bottom && p.size <= top))
```



Лямбды в JET: текущий статус

- пока работают через indy, но тела лямбд статически компилируются
- динамическая компиляция классов-заверток для лямбд очень быстрая (<1ms)
- на стандартных бенчмарках, и реальных приложениях пока не было замечено уменьшения производительности из-за лямбд

Типовые аннотации

EXCELSIOR

Типовые аннотации

Могут быть везде где есть тип:

```
class TypeAnnotationsExample {

    @Encrypted String data;
    List<@NonNull String> strings;
    HashSet names =
        (@Immutable HashSet) set;

}
```

EXCELSIOR

Типовые аннотации

- Кодируются в Java байткоде атрибутом типа Runtime(In)VisibleTypeAnnotation – новые атрибуты Java 8 байткода
- Доступны в рантайм через Reflection API:
  - getTypeAnnotations

Как представляются в Excelsior JET?

EXCELSIOR

Reflection в HotSpot

- HotSpot JVM хранит в MetaSpace рантайм представление Java класс-файла
- В класс-файле есть вся нужная информация для reflection
- Реализация Reflection API в HotSpot работает непосредственно с класс-файлами

EXCELSIOR

Reflection в Excelsior JET

- После статической компиляции от классов не остается класс-файлов
  - После резолва ссылок в AOT компиляторе и трансляции байткода в машинный код, большая часть класс-файла становится не нужна в своем первоначальном виде.
- Мета-информация о классах пишется в исполняемый файл в секцию данных
- Пишется только то, что необходимо Reflection
  - имена полей, методов, **аннотации**

EXCELSIOR

Reflection в Excelsior JET

- Аннотации пишутся в формате очень похожем на формат аннотаций в Java байткоде
  - можно переиспользовать Java код, который их разбирает

Для **Java 8** AOT компилятор пришлось научить сохранять типовые аннотации

EXCELSIOR

Имена параметров

- Если `javac` задать ключ `-parameters`, то
  - в класс-файл будут писаться имена параметров
  - имена параметров будут доступны через reflection

Как представляются в Excelsior JET?

Легко!

EXCELSIOR

New Java 8 APIs

New Java 8 APIs

- Stream API
- Time API
- JavaFX 8
- Nashorn

Как поддерживаются в JET?

EXCELSIOR

New Java 8 APIs

- Stream API
- Time API
- JavaFX 8
- Nashorn

Как поддерживаются в JET?

Также как и все остальные Java API !

EXCELSIOR

New Java 8 APIs

- Excelsior – официальный Java SE licensee
  - лицензируем все исходные тексты Java SE у Oracle
  - можем поступать с ними как хотим, например компилировать Java байт-код платформы в машинный код и использовать свою реализацию JVM
  - обязаны проходить тесты на совместимость (JCK)

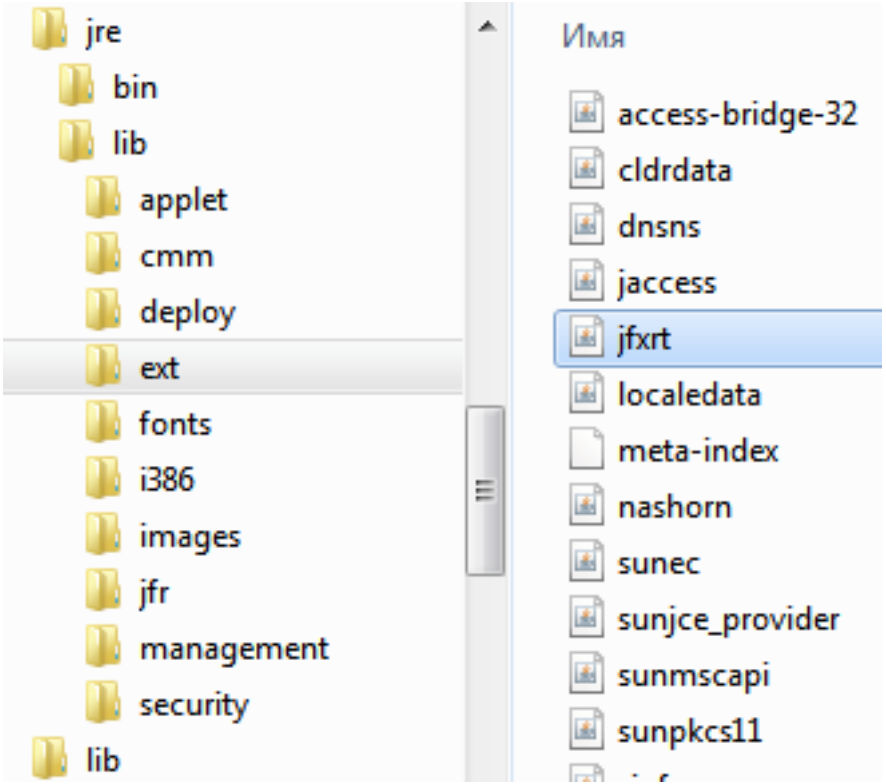
New Java 8 APIs

Не все Java 8 APIs одинаково стандартны:

- Stream API, Time API входят в стандарт Java 8
- Nashorn, JavaFX **НЕ** входят в стандарт Java SE
  - в частности нет тестов JCK

EXCELSIOR

Структура Oracle JRE



EXCELSIOR

Структура Oracle JRE

- Классы Nashorn и JavaFX находятся в папке ext
  - грузятся Extension классслоадером
- Все что находится в папке ext можно официально не включать в Private JRE дистрибутива своего приложения

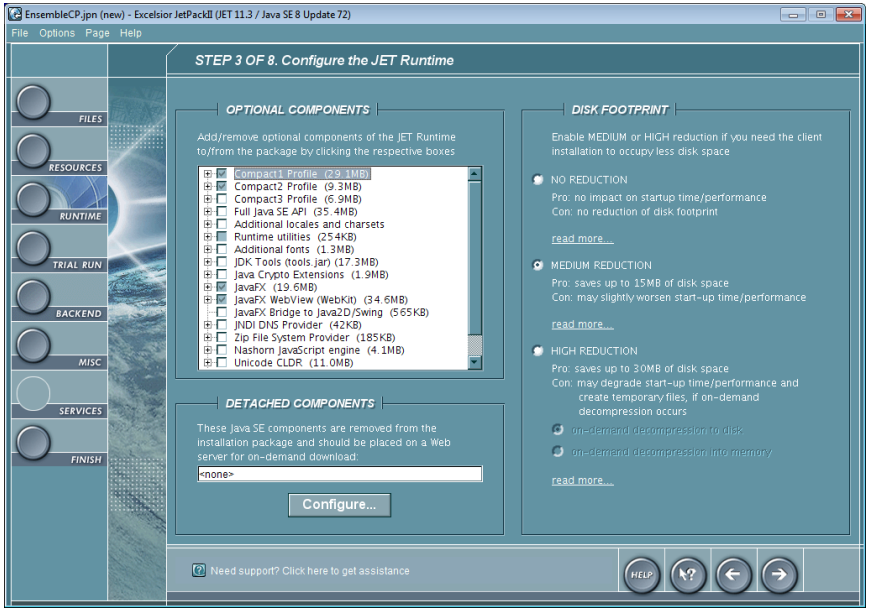
Распространение приложений скомпилированных Excelsior JET

- Скомпилированные с помощью Excelsior JET приложения всегда распространяются с приватной копией JET Runtime:
  - ваши пользователи не должны ставить “Excelsior JRE”

Распространение приложений скомпилированных Excelsior JET

- Структура JET Runtime на диске очень похожа на Oracle JRE:
  - ресурсы (шрифты, картинки) и библиотеки с native методами находятся на тех же местах
  - классы из rt.jar и других .jar компилируются в набор DLL

JetPackII



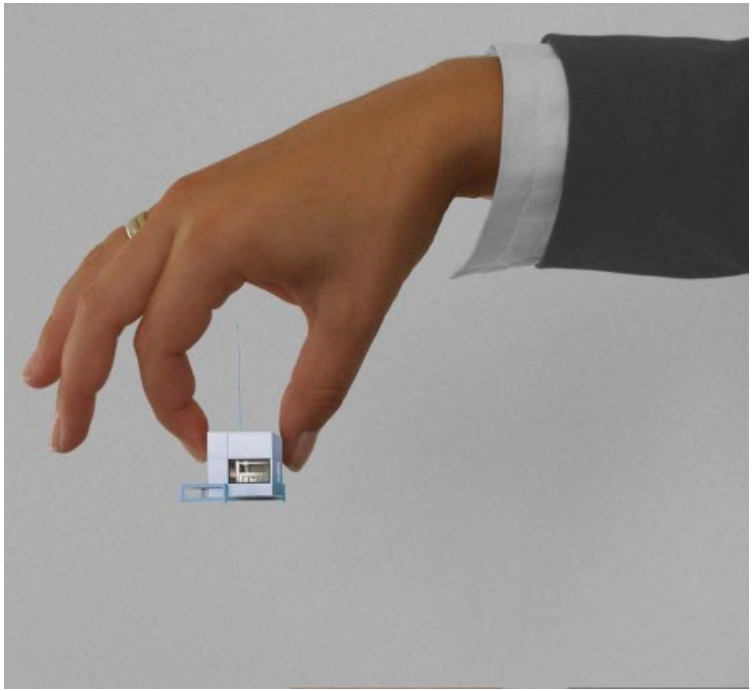
Подготавливает ваше приложение для дальнейшего распространения

EXCELSIOR

JetPackII

- JET-скомпилированные исполняемые файлы связываются с приватной копией JET Runtime
- Позволяет выбирать нужные части JET Runtime для вашего приложения
- **New in Java 8:** позволяет **НЕ** включать JavaFX и Nashorn, если они не используются вашим приложением

Compact Profiles



Compact Profiles

Compact1	Compact2	Compact3
java.lang	java.rmi	java.lang.instrument
java.io	java.sql	java.lang.management
java.math	javax.transaction	javax.management
java.nio	javax.xml	javax.naming
java.util	org.w3c.dom	javax.script
java.net	org.xml.sax	javax.security
java.security		javax.sql
javax.crypto		javax.xml.crypto
java.text		org.ietf.jgss

Compact Profiles

- Есть в Java SE Embedded (Linux builds)
- В OpenJDK (не собираются на Windows и Mac OS X)

История компонентизации Java SE от Excelsior

- Разбиение Java SE на APIs (2000 год)
  - компиляция их в DLL
- JetPerfect (2001 год)
  - компиляция приложений с нужными частями Java SE в один EXE файл
  - SWT Example: 1.4 MB
- Java Runtime Slim-Down (2007 год)
  - совместимое с Java SE спецификацией решение
- Compact Profiles (2016 год)

EXCELSIOR

История компонентизации Java SE от Sun/Oracle

- Kernel JRE (2006 год)
- Project Jigsaw 2007 – 2017(?)
- Compact Profiles (2014 год)

Compact Profiles

- Первый рабочий и легальный способ уменьшения Private JRE
- Есть JCK (тесты совместимости)

Compact Profiles

- Первый рабочий и легальный способ уменьшения Private JRE
- Есть JCK (тесты совместимости)

- Есть в Excelsior JET для всех десктопных платформ (Windows, Linux, Mac OS X)

EXCELSIOR

Compact Profiles

- Поддержка в Excelsior JET:
  - Переразбивка JET RT DLLs по границам Compact Profiles
  - Поддержка в JetPackII: паковка приложения с компактным профилем, на котором может быть запущено приложение
  - Проходит JCK для соответствующих компактных профилей

JavaFX on Compact Profiles

JavaFX 8 on Compact Profiles

- JavaFX условно состоит из трех логических частей:
  - Core JavaFX
  - JavaFX Web (WebKit based)
  - JavaFX bridges to Swing и SWT

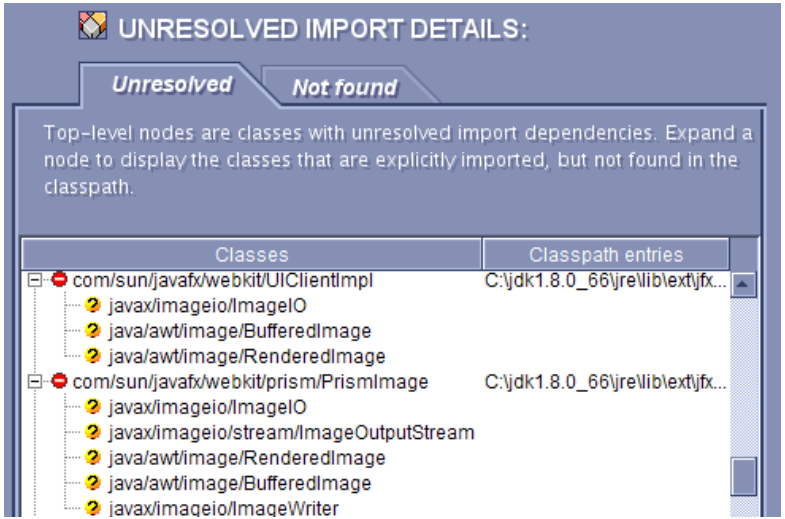
JavaFX 8 on Compact Profiles

- Core JavaFX может работать на Compact2
  - FXML зависит от XML

- JavaFX Swing Bridge требует Full Java SE
  - Так как очевидно зависит от Swing

JavaFX 8 on Compact Profiles

- JavaFX Web тоже имеет зависимости от AWT …



EXCELSIOR

JavaFX 8 on Compact Profiles

- JavaFX Web тоже имеет зависимости от AWT, которые не разрешаются во время исполнения ☺!

JavaFX 8 on Compact Profiles

- JavaFX Web тоже имеет зависимости от AWT, которые не разрешаются во время исполнения, кроме одной ☹ …

JavaFX 8 on Compact Profiles

- JavaFX Web тоже имеет зависимости от AWT, которые не разрешаются во время исполнения, кроме одной, которая тоже не разрешается в режиме -Xverify:none

JavaFX 8 on Compact Profiles

Поддержка JavaFX Web в Excelsior JET:

- зависимости от AWT откладываются до разрешения во время исполнения
  - не разрешаются в статическом компиляторе
- Все классы JavaFX верифицируются до исполнения
  - не требуется верификация во время исполнения и загрузка “лишних” классов
- В результате JavaFX Web работает на **Compact2**!

EXCELSIOR

Java Packager (Oracle JDK) vs. JetPackII (Excelsior JET)

	Java Packager	JetPackII with Compact Profiles
Ensemble Demo (with WebView sample)	46 MB	21 MB
BrickBreaker	42 MB	10 MB

Примечание: замеры производились на Windows 32-bit для Java 8 Update 72, представлены размеры дистрибутивов без зависимостей

EXCELSIOR

Заключение

- Java 8 – лучший релиз Java со времен Java 5
  - лямбды повышают производительность труда
- Лямбды не повлияли на спецификацию JVM
  - но желательна специальная поддержка в JVM
- Compact Profiles уменьшают размер Private JRE
  - важно для магазинов приложений и в embedded
- Java != HotSpot

EXCELSIOR

Вопросы и ответы

Никита Липский,
Excelsior

nlipsky@excelsior-usa.com
twitter: @pjBooms

EXCELSIOR