
ORACLE®


Java™
ORACLE®

# Lambda Expressions in Java ME Embedded

Sergey Troshin
Principal Member of Technical Staff
Java ME Embedded, Oracle

September 29, 2014

# Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Program Agenda

1. Java ME Embedded Overview
2. Dreaming of Lambdas in ME
3. Dreams come true
4. Alternatives

# Program Agenda

1. Java ME Embedded Overview
2. Dreaming of Lambdas in ME
3. Dreams come true
4. Alternatives

# Java ME Embedded

**Key Principles**

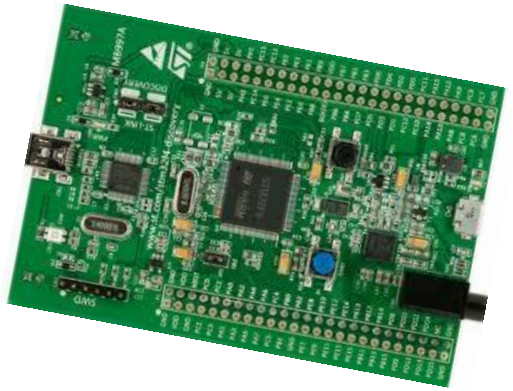
- Java ME 8 is the “little sibling” of Java SE 8
- Portability of applications and libraries across the Java Platform
- Java ME vs. Java SE is a footprint/functionality tradeoff
- Java ME & Java SE release cycles are in sync

**Benefits**

- Modern and flexible platform for delivering embedded software
- Unified development experience & community across Java
- Aligned Java language, core APIs, development, and tools
- Enable 9+ Million Java developers to develop for Java Embedded

# Java ME Embedded Target Devices

- Wide range of embedded hardware from MCU to gateways and wireless modules
- Target CPUs are ARM 9, 11, Cortex-A/M3/M4
- Constrained memory (128K – 32M RAM, from 1M of ROM/Flash)
- Variety of peripherals
- Minimal OS requirement
  - Single process
  - Single native thread
  - “Bare metal” is possible



# Java ME 8 Platform Overview

**Additonal APIs (Examples)**

- Messaging
- Wireless Communication
- Protocols and Data Conversion
- Security and Management
- Additional Optional APIs
- Location
- Web Services
- Sensors
- Additional Optional JSRs
- Vertical Specific APIs

**On-Device I/O Access**

- Device I/O API

**Application Platform**

- **Java ME Embedded Profile** (MEEP 8, JSR 361)

**Security and Trust Services**

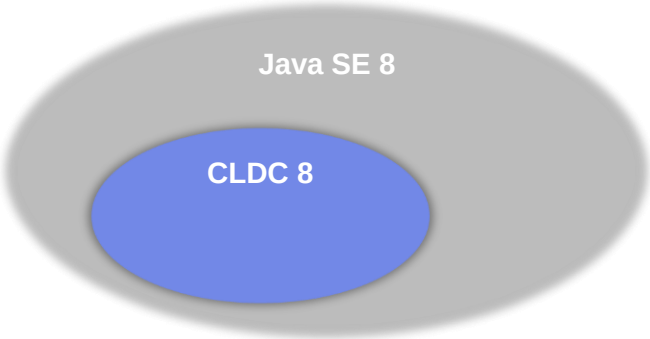
- SATSA

**Java VM**

- **Java ME CLDC 8** (JSR 360, aligned with Java SE 8)

# CLDC 8 High-Level Overview

- Description
  - CLDC 8 is revolutionary update for CLDC 1.1.1 that brings the VM, Java language and core API libraries in alignment with Java SE 8
- Key Features
  - Synchronize Java SE 5/6/7/8 language features into ME
  - Introduce community requested Java SE API library features
  - Virtual Machine update, support latest class file format
  - More footprint optimizations and options
- Specification Requirements
  - CLDC 8 to be an extended strict subset of Java SE 8
  - Consolidated Generic Connection Framework (GCF)
  - Backward binary compatible



# CLDC HI VM Implementation

- Implements CLDC 8 Specification (JSR360)
- Optimized assembly interpreter
- Optimizing dynamic adaptive and ahead-of-time compilers
- Generational mark’n’compact GC
- Single native thread, multiple Java threads, fair scheduler
- Multitasking (MVM) with isolation and memory quotas
- Minimized Footprint
  - compact data structures
  - aggressive build-time optimizations for footprint and performance

# Program Agenda

1. Java ME Embedded Overview
2. Dreaming of Lambdas in ME
3. Dreams come true
4. Alternatives

# What is Lambda Expression?

- Lambda expression introduced as language feature in Java 8
- Functional interface is an interface with exactly one abstract method
- Lambda expression is a block of code with parameters that can be executed and referenced
- Lambda expression is treated by VM runtime as an instance of a functional interface
- Lambda expression can capture effectively final variables from the enclosing scope

# Sample: Inner Class vs. Lambda Syntax

```
String data[] = {“orange", “apple", “carrot"};
```

```
Arrays.sort(data, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
});
```

```
Arrays.sort(data, (s1, s2) -> s1.compareTo(s2));
```

```
Arrays.sort(data, String::compareTo);
```

# Why Lambdas are Wanted in Java ME

- Common reasoning for Java
  - General purpose languages support functional programming paradigm
  - Anonymous classes can help, but syntax is cumbersome
  - Less text - less errors, neat syntax of Java lambdas
  - More optimal data processing by VM is possible
- Java ME Embedded reasons
  - Inner classes are heavy, lambdas can be more efficient
  - Easier to port any Java code from SE to ME
  - Development experience consistent with Java SE

# Lambda expression friends

- Default interface methods (Defender methods)
  - Allow interface to evolve without breaking the existing code

```
public interface Iterable<T> {
  public default void forEach(Consumer<? super T> consumer) {
    for (T t : this) {
      consumer.accept(t);
}}}
```

- Method reference
  - Allows to use an existing method as lambda expression
  - Static, instance methods and constructors can be referred

```
Comparator<String> comparator = String::compareTo;
```

- Collections updated for lambdas, bulk operations and parallelization

# What is Missing in CLDC 8 Spec for Lambdas?

- CLDC 8 is more close to SE 7 rather than to SE 8
- No invokedynamic support by CLDC VM
- Lambdas are not supported by the CLDC 8 spec
  - Inner classes were generated for lambdas by early javac versions
  - Retrolambda tool to backport Java 8 ‘s lambdas to Java 7, 6 and 5
    https://github.com/orfjackal/retrolambda
- No default interface methods support
- Collections are minimalistic

# Not the Goals for Java ME Embedded

- Parallel data processing makes sense if multiple CPU cores are engaged
- Limited invokedynamic support is enough for Java lambdas
  - Only small subset of java.lang.invoke API is referenced by the Java compiler to implement lambdas in generated class files

    CallSite, MethodHandle, MethodHandles, MethodType, LambdaMetaFactory, LambdaConversionException
  - Non-Java dynamic languages support is not the goal for Java ME Embedded VM
- No goal to provide java.lang.invoke API for developers, for javac only
- Lambdas serialization is out of scope

# Program Agenda

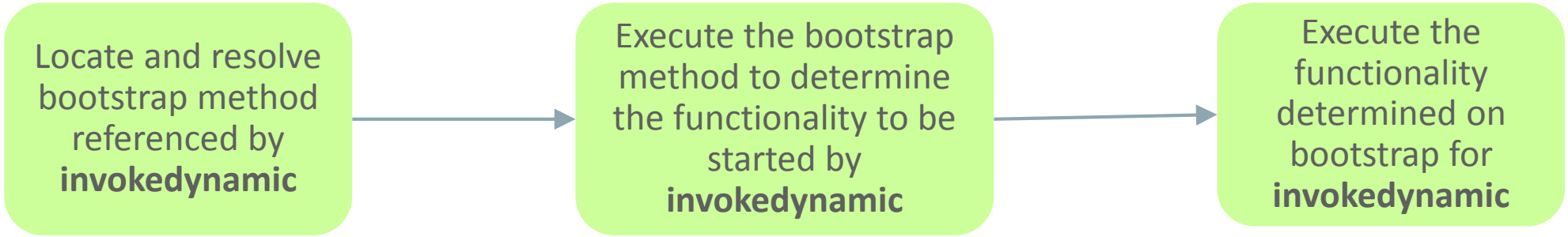
1. Java ME Embedded Overview
2. Dreaming of Lambdas in ME
3. Dreams come true
4. Alternatives

# CLDC HI VM Evolution towards Lambdas

- Lambdas are compiled by javac 8 into bytecodes using invokedynamic
- Java compiler does a lot, but not all – VM & runtime support is needed
- Class file parsing is updated to support
  - new constant pool tags and structures (see JVMS 8)
    CONSTANT_MethodHandle, CONSTANT_MethodType, CONSTANT_InvokeDynamic, …
  - invokedynamic bytecode instruction
- Execution of invokedynamic is added to the VM
- Runtime libraries update for lambdas
  - collections, java.util.function API, …

# Intro for invokedynamic

- Introduced in JVMS 7 and designed to determine at runtime what functionality to be executed on method invocation
- References bootstrap method to be called on the first execution
- The bootstrap method resolves what should be executed this and the next times the instruction is executed by the VM



# Lambdas Support in Java ME Embedded

- Java compilation of lambda expressions (by javac)
- Runtime execution of lambda expressions by CLDC HI VM

# Java Compilation of Lambdas (by javac)

- Each lambda expression is matched with proper functional interface
- The body of lambda expression is “desugared” into private static method of container class
- Context captured by the lambda expression is presented as extra parameters of the “desugared” method
- Invocation of the lambda expression is generated using invokedynamic bytecode

# Runtime Execution of Lambdas by CLDC HI VM

- The bootstrap method of invokedynamic generates synthetic Java class at runtime on the first invocation of the lambda expression
  - The synthetic class implements the functional interface of the lambda
  - The context captured by the lambda is presented as private members of the class
  - The class constructor accepts the captured context values
  - The main functional method of the class is implemented as the call of the “desugared” lambda body
  - On the call the parameters adaptation is done, the captured context is passed
- The bootstrap method resolves invokedynamic into the call of constructor of the synthetic class generated for the lambda, captured context passed

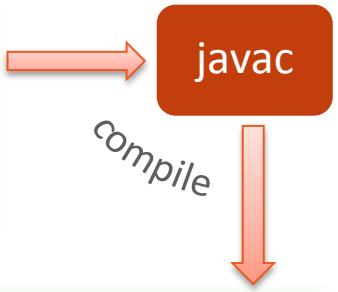
# Runtime Execution of Lambdas by CLDC HI VM (cont.)

- The result of the invokedynamic execution is new instance of the synthetic class pushed onto the Java execution stack
- The synthetic class is reused on each next invocation of the lambda, new instance is created each time
- Unused synthetic classes of lambdas can be collected by GC
- The bootstrap to be started again for lambdas have been executed already, but whose synthetic classes have been collected already

# Sample: Lambda and invokedynamic

Sample.java

```
public class Sample {
  static void main(String[] args) {
    new Thread(
      () -> System.out.println("Hello World!")
    ).start();
  }
}
```

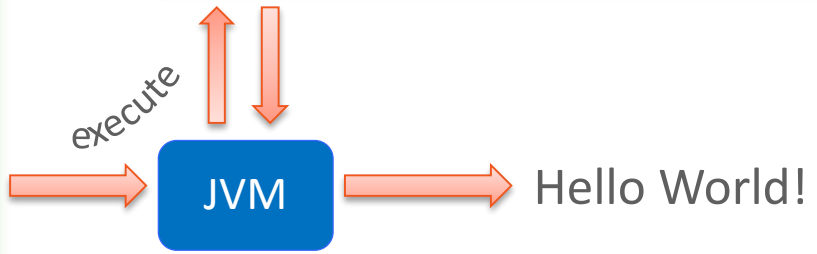



Sample.class

```
public class Sample
  flags: ACC_PUBLIC, ACC_SUPER {

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    Code:
      stack=3, locals=1, args_size=1
         0: new java/lang/Thread
         3: dup
         4: invokedynamic InvokeDynamic #0:run:()Ljava/lang/Runnable;
         9: invokespecial java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
        12: invokevirtual java/lang/Thread.start:()V
        15: return

  private static void lambda$main$0();
    descriptor: ()V
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic java/lang/System.out:Ljava/io/PrintStream;
         3: ldc "Hello World!"
         5: invokevirtual java/io/PrintStream.println:()V
         8: return
}
```

Synthetic

```
final class Sample$$Lambda$1 implements java.lang.Runnable
  flags: ACC_FINAL, ACC_SUPER, ACC_SYNTHETIC {

  private Sample$$Lambda$1();
    descriptor: ()V
    flags: ACC_PRIVATE
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial java/lang/Object."<init>":()V
         4: return

  public void run();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=0, locals=1, args_size=1
         0: invokestatic Sample.lambda$main$0:()V
         3: return
}
```



# Even More Support for Lambdas by CLDC HI VM

- Default and static interface methods are supported by CLDC HI VM
- Method references can be used as lambda expressions
- java.lang.invoke API is not provided for developers, for VM only
- Marker interfaces are supported, serialization not

```
interface Marker {}
@FunctionalInterface interface Action { void run(); }
Action a = (Action & Marker)(() -> System.out.println()));
```

- Lambda expressions dynamic compilation at runtime and ahead-of-time
- Optional lambdas preloading for system code (romization)

# What CLDC HI VM can do differently from SE 8

- The synthetic class for lambda is more lightweight then a standard class
  - Loaded and pre-initialized on synthesis
  - Synthetic method calls to be quickened
- Bare minimum of java.lang.invoke.LambdaMetaFactory functionality is supported by CLDC HI VM
- java.lang.invoke API types CallSite, MethodType, MethodHandle, etc. are used as internal marker types, not instantiated indeed
  - Instead of CallSite creation on bootstrap constant pool entry of the invokedynamic is updated with synthetic class ID

# Proposed CLDC 8+ Spec Changes for Lambdas

- Define strict subset of java.lang.invoke API sufficient for Java lambdas
- Define limited invokedynamic support by VM
- Remove limitations on class file data structures used by invokedynamic
- Remove BootstrapMethods attribute from the unsupported attributes list
- Introduce API changes implied by lambda: collections, java.util.function API subset, …
- Support default and static methods of interfaces

# Program Agenda

1. Java ME Embedded Overview
2. Dreaming of Lambdas in ME
3. Dreams come true
4. Alternatives

# Classless Alternative for Lambda Implementation

- No class generation at runtime on invokedynamic bootstrap
- The predefined VM internal polymorphic class that can mimic any functional interface is used instead, not Java class indeed
- Instances of the polymorphic class are created for lambda invocations
- Adapter methods for lambda body calls have to be generated still
- Efficiency for expected popular use cases of lambda expressions
- With bridge methods and marker interfaces it gets closer to regular classes
- Not entirely spec compliant

# Conclusions

- CLDC 8 Spec to be updated for lambda expressions support
- CLDC HI VM team researched different ways to implement lambdas expressions efficiently
- CLDC HI VM development builds implement lambdas in the way aligned with standard Java tools and concepts
- The future Java ME Embedded platforms will provide lambda expressions support

# References

- Oracle Java ME Embedded Platform (OTN)
  http://www.oracle.com/technetwork/java/embedded/javame/embed-me/overview/index.html
- Java ME Embedded 8
  http://docs.oracle.com/javame/8.0/index.html
- CLDC 8 Specification (JSR 360)
  https://jcp.org/en/jsr/detail?id=360
- Java ME blogs:
  - Alexander Belokrylov - https://blogs.oracle.com/javame/
  - Terrence Barr - http://terrencebarr.wordpress.com/

# Q & A

# Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Hardware and Software
## Engineered to Work Together


Java™
ORACLE®


ORACLE®