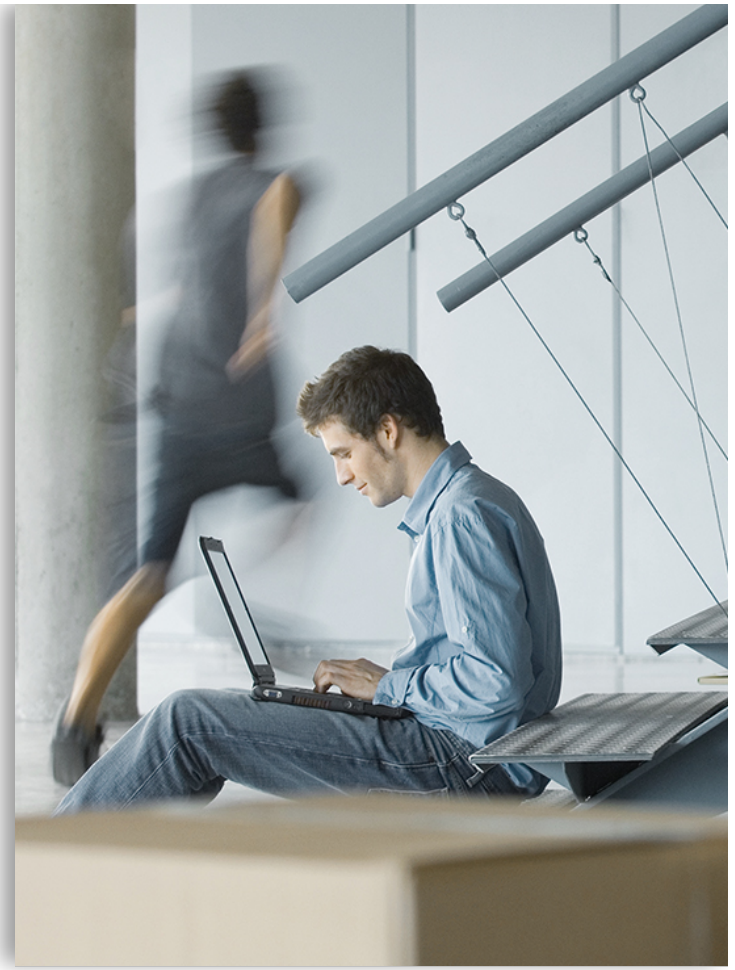

Java™


Java™
Java 8:
Create The Future
Simon Ritter
Head of Java Technology Evangelism
Oracle Corp.
CREATE
THE FUTURE
ORACLE

Program Agenda

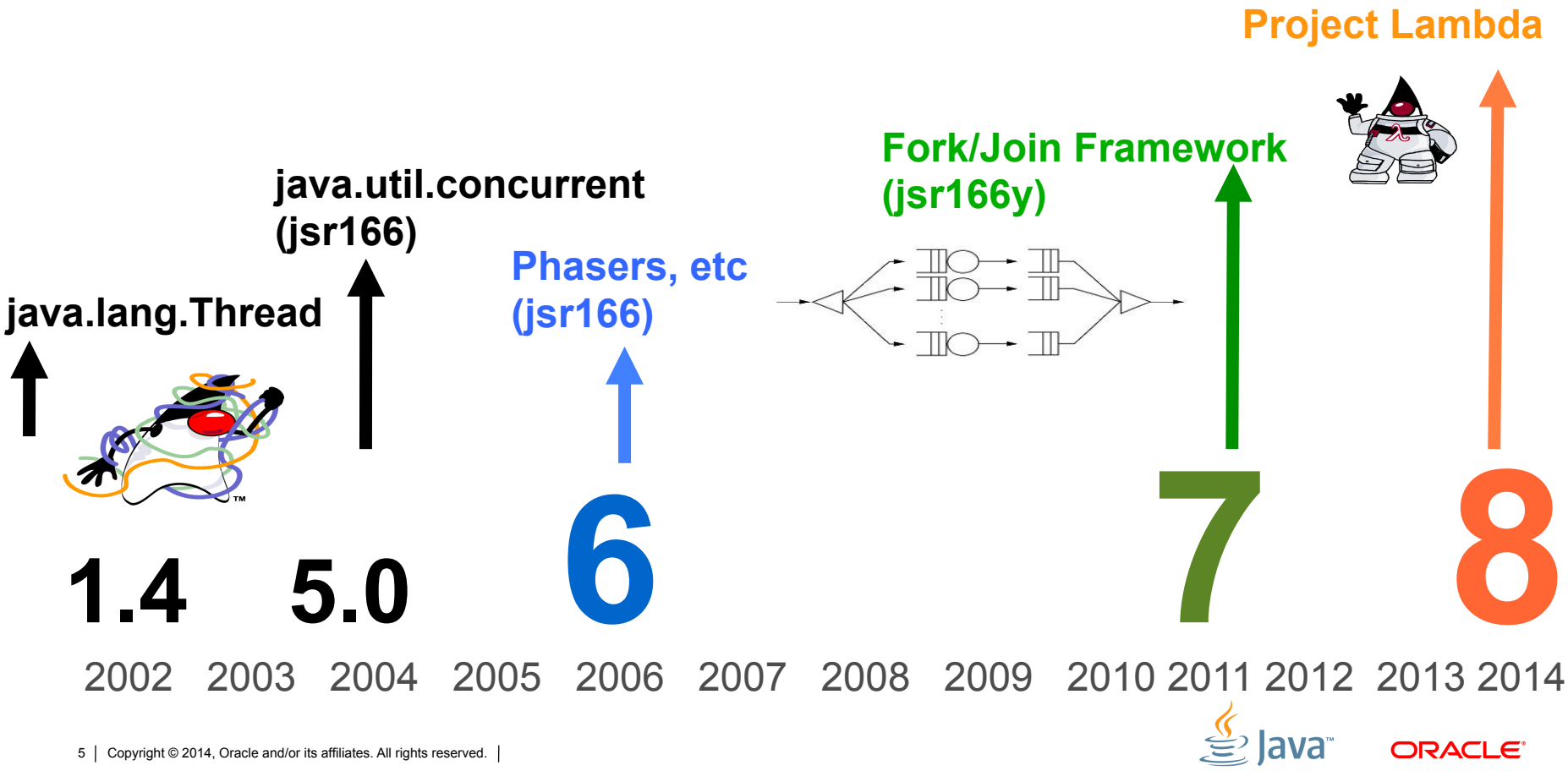
- Java SE 8: Enhancing the Core Java Platform
  - Lambdas and Streams
  - Other new features
- Java ME 8: Building The Internet of Things
- NetBeans 8: The IDE For Java 8
- Where Next?

Java SE 8:
Lambdas & Streams:
Functional Programming In Java



Java™ ORACLE®

Concurrency in Java



Java™ ORACLE®

The Problem: External Iteration

```
List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.gradYear == 2011) {
    if (s.score > highestScore) {
      highestScore = s.score;
    }
  }
}
```

- Client controls iteration
- *Inherently serial:* iterate from beginning to end
- Not thread-safe because business logic is stateful (mutable accumulator variable)

Internal Iteration With Inner Classes

```
List<Student> students = ...
double highestScore = students.
  filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011;
    }
  }).
  map(new Mapper<Student,Double>() {
    public Double extract(Student s) {
      return s.getScore();
    }
  }).
  max();
```

- Iteration handled by the library
- Not inherently serial – traversal *may* be done in parallel
- Traversal *may* be done lazily – so one pass, rather than three
- Thread safe – client logic is stateless
- High barrier to use
  - Syntactically ugly

Internal Iteration With Lambdas

```
SomeList<Student> students = ...
double highestScore = students.
    filter(Student s -> s.getGradYear() == 2011).
    map(Student s -> s.getScore()).
    max();
```

- More readable
- More abstract
- Less error-prone

Lambda Expressions

Some Details

- Lambda expressions represent anonymous functions
  - Same structure as a method
    - typed argument list, return type, set of thrown exceptions, and a body
  - Not associated with a class
- We now have parameterised behaviour, not just values

```
double highestScore = students.
        filter(Student s -> s.getGradYear() == 2011)
        map(Student s -> s.getScore())
        max();
```

What

How

Java™ ORACLE®

Lambda Expressions

Functional Interfaces

- Definition
  - A *functional interface* is an interface with only one abstract method
  - However, the interface may have more than one method
- Identified structurally
  - Type is inferred from the context
  - Works for both assignment and method parameter contexts
- The type of a Lambda expression is a *functional interface*
  - Instances of functional interfaces are created with Lambda expressions
  - `@FunctionalInterface` annotation

Type Inference

- The compiler can often infer parameter types in a lambda expression
- Inferrence based on the target functional interface's method signature

```
static T void sort(List<T> l, Comparator<? super T> c);
```

```
List<String> list = getList();
Collections.sort(list, (String x, String y) -> x.length() - y.length());
```

```
Collections.sort(list, (x, y) -> x.length() - y.length());
```

- Fully statically typed (no dynamic typing sneaking in)
  - More typing with less typing

Lambda Epressions

Local Variable Capture & The Meaning of `'this'`

- Lambda expressions can refer to *effectively final* local variables from the enclosing scope
  - This means a variable behaves as if it is marked final (even if it is not)
  - The variable is assigned once
- Lambda expressions are anonymous functions
  - They are not associated with an object
  - **`this`** will refer to the object in the surrounding scope

Method References

- Method references let us reuse a method as a lambda expression

```
FileFilter x = (File f) -> f.canRead();
```

```
FileFilter x = File::canRead;
```


Constructor References

- Same concept as a method reference
  - For the constructor

```
Factory<List<String>> f = ArrayList<String>::new;
```

Equivalent to

```
Factory<List<String>> f = () -> return new ArrayList<String>();
```


Default Methods

- Provide a mechanism to add new methods to existing interfaces
  - Without breaking backwards compatability
  - Gives Java multiple inheritance of behaviour, as well as types
    - but not state!

```
public interface Set<T> extends Collection<T> {
    ...   // The existing Set methods

  default Spliterator<E> spliterator() {
    return Spliterators.spliterator(this, Spliterator.DISTINCT);
  }
}
```


Static Methods In Interfaces

- Previously it was not possible to include static methods in an interface
- Static methods, by definition, are not abstract
  - `@FunctionalInterface` can have zero or more static methods

```
static <T> Predicate<T> isEqual(Object target) {
        return (null == target)
                ? Objects::isNull
                : object -> target.equals(object);
}
```

Streams API

Aggregate Operations

- Most business logic is about aggregate operations
  - "Most profitable product by region"
  - "Group transactions by currency"
- As we have seen, up to now, Java mostly uses external iteration
  - Inherently serial
  - Frustratingly imperative
- Java SE 8's answer: **`Streams`**
  - With help from Lambdas

Stream Overview

High Level

- Abstraction for specifying aggregate computations
  - Not a data structure
  - Can be infinite
- Simplifies the description of aggregate computations
  - Exposes opportunities for optimisation
  - Fusing, laziness and parrallelism

Stream Overview

Pipeline

- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect

```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals("London")).
  mapToInt(Transaction::getPrice).
  sum();
```

Source

Intermediate operation

Terminal operation

Java™ ORACLE®

Stream Overview

Execution

- The **`filter`** and **`map`** methods don't really do any work
  - They set up a pipeline of operations and return a new **`Stream`**
- All work happens when we get to the **`sum()`** operation
  - **`filter()/map()/sum()`** fused into one pass on the data
    - For both sequential and parallel pipelines

```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals("London")).   // Lazy
  mapToInt(Transaction::getPrice).                        // Lazy
  sum();                                        // Execute the pipeline
```


Stream Sources

Many Ways To Create

- From collections and arrays
  - `Collection.stream()`
  - `Collection.parallelStream()`
  - `Arrays.stream(T array)` or `Stream.of()`
- Static factories
  - `IntStream.range()`
  - `Files.walk()`
- Roll your own
  - `java.util.Spliterator()`

Stream Sources

Manage Three Aspects

- Access to stream elements
- Decomposition (for parallel operations)
  - Fork-join framework
- Stream characteristics
  - **`ORDERED`**
  - **`DISTINCT`**
  - **`SORTED`**
  - **`SIZED`**
  - **`NONNULL`**
  - **`IMMUTABLE`**
  - **`CONCURRENT`**

Stream Intermediate Operations

- Uses lazy evaluation where possible
- Can affect stream characteristics
  - **`map()`** preserves **`SIZED`** but not **`DISTINCT`** or **`SORTED`**
- Some operations fuse/convert to parallel better than others
  - Stateless operations (**`map`**, **`filter`**) fuse/convert perfectly
  - Stateful operations (**`sorted`**, **`distint`**, **`limit`**) fuse/convert to varying degrees

Stream Terminal Operations

- Invoking a terminal operation executes the pipeline
  - All operations can execute sequentially or in parallel
- Terminal operations can take advantage of pipeline characteristics
  - **`toArray()`** can avoid copying for **`SIZED`** pipelines by allocating in advance

Optional<T>

Reducing NullPointerException Occurences

- Indicates that reference may, or may not have a value
  - Makes developer responsible for checking
  - A bit like a stream that can only have zero or one elements

```
Optional<GPSData> maybeGPS = Optional.ofNullable(gpsData);

maybeGPS.ifPresent(GPSData::printPosition);

GPSData gps = maybeGPS.orElse(new GPSData());

maybeGPS.filter(g -> g.lastRead() < 2).ifPresent(GPSData.display());
```

`java.util.function` Package

- **`Predicate<T>`**
  - Determine if the input of type T matches some criteria
- **`Consumer<T>`**
  - Accept a single input argumentof type T, and return no result
- **`Function<T, R>`**
  - Apply a function to the input type T, generating a result of type R
- Plus several more type specific versions

Stream Example 1

Convert words in list to upper case

```
List<String> output = wordList.
  stream().
  map(String::toUpperCase).
  collect(Collectors.toList());
```

Stream Example 1

Convert words in list to upper case (in parallel)

```
List<String> output = wordList.
  parallelStream().
  map(String::toUpperCase).
  collect(Collectors.toList());
```

Stream Example 2

Find words in list with even length

```
List<String> output = wordList.
  stream().
  filter(w -> (w.length() & 1 == 0).
  collect(Collectors.toList());
```

Stream Example 3

Count lines in a file

- BufferedReader has new method
  - `Stream<String> lines()`

```
long count = bufferedReader.
  lines().
  count();
```


Stream Example 4

Find the length of the longest line in a file

```
int longest = reader.
  lines().
  mapToInt(String::length).
  max().
  getAsInt();
```


Stream Example 5

Collect all words in a file into a list

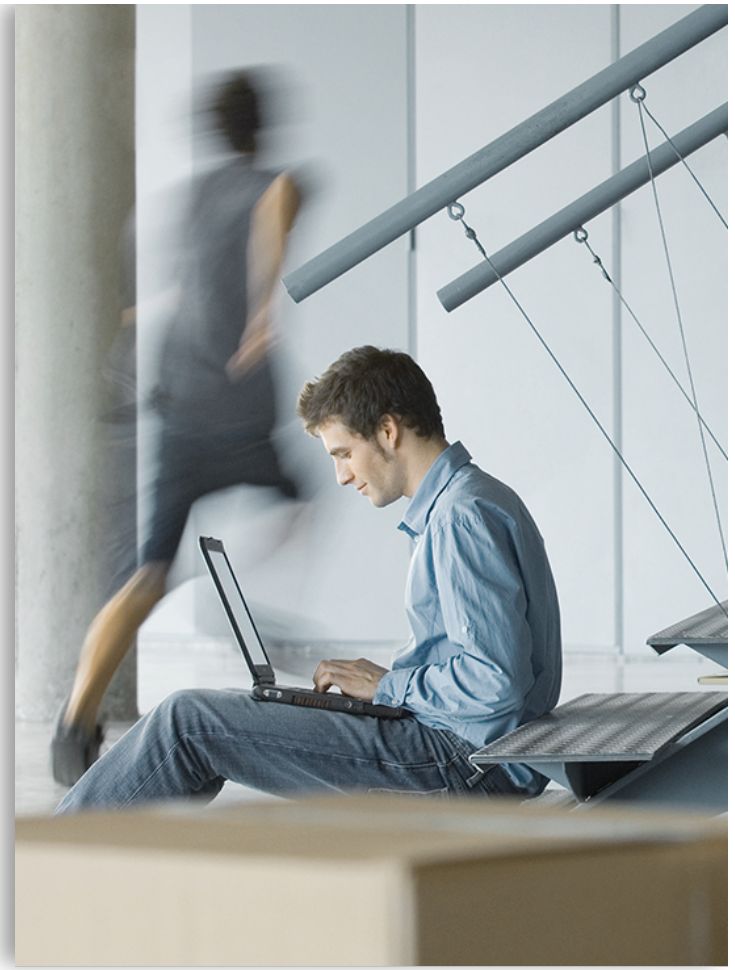
```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  collect(Collectors.toList());
```

Stream Example 6

List of unique words in lowercase, sorted by length

```
List<String> output = reader.
  lines().
  flatMap(line -> Stream.of(line.split(REGEXP))).
  filter(word -> word.length() > 0).
  map(String::toLowerCase).
  distinct().
  sorted((x, y) -> x.length() - y.length()).
  collect(Collectors.toList());
```

Java SE 8:
Other New Features



Java™ ORACLE®

Annotations On Java Types

- Annotations can currently only be used on type declarations
  - Classes, methods, variable definitions
- Extension for places where types are used
  - e.g. parameters
- Permits error detection by pluggable type checkers
  - e.g. null pointer errors, race conditions, etc

```
public void process(@notnull List data) {…}
```


Concurrency Updates

- Scalable update variables
  - **`DoubleAccumulator`**, **`DoubleAdder`**, etc
  - Multiple variables avoid update contention
  - Good for frequent updates, infrequent reads
- **`ForkJoinPool`** improvements
  - Completion based design for IO bound applications
  - Thread that is blocked hands work to thread that is running

Parallel Array Sorting

- Additional utility methods in `java.util.Arrays`
  - `parallelSort` (multiple signatures for different primitives)
- Anticipated minimum improvement of 30% over sequential sort
  - For dual core system with appropriate sized data set
- Built on top of the fork-join framework
  - Uses Doug Lea's `ParallelArray` implementation
  - Requires working space the same size as the array being sorted

Date And Time APIs

- A new date, time, and calendar API for the Java SE platform
- Supports standard time concepts
  - Partial, duration, period, intervals
  - date, time, instant, and time-zone
- Provides a limited set of calendar systems and be extensible to others
- Uses relevant standards, including ISO-8601, CLDR, and BCP47
- Based on an explicit time-scale with a connection to UTC

Base64 Encoding and Decoding

- Currently developers are forced to use non-public APIs
  - `sun.misc.BASE64Encoder`
  - `sun.misc.BASE64Decoder`
- Java SE 8 now has a standard way
  - `java.util.Base64.Encoder`
  - `java.util.Base64.Decoder`
  - `encode`, `encodeToString`, `decode`, `wrap` methods

Nashorn JavaScript Engine

- Lightweight, high-performance JavaScript engine
  - Integrated into JRE
- Use existing `javax.script` API
- ECMAScript-262 Edition 5.1 language specification compliance
- New command-line tool, `jjs` to run JavaScript
- Internationalised error messages and documentation

Removal Of The Permanent Generation

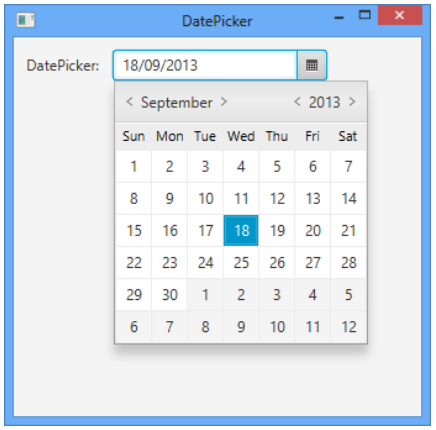
Permanently

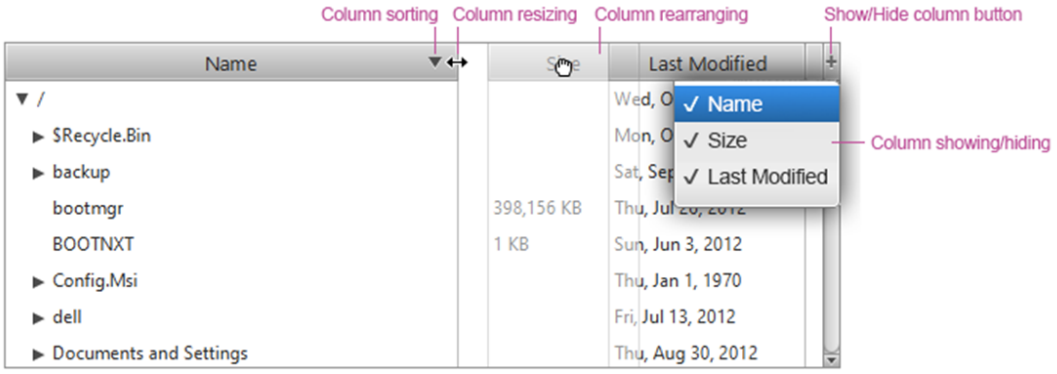
- No more need to tune the size of it
- Current objects moved to Java heap or native memory
  - Interned strings
  - Class metadata
  - Class static variables
- Part of the HotSpot, JRockit convergence

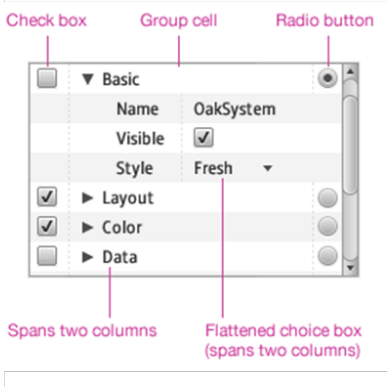
JavaFX 8

New Controls

- **DatePicker**
- **TreeTableView**







 | |

Java™ ORACLE®

JavaFX 8

Touch Support

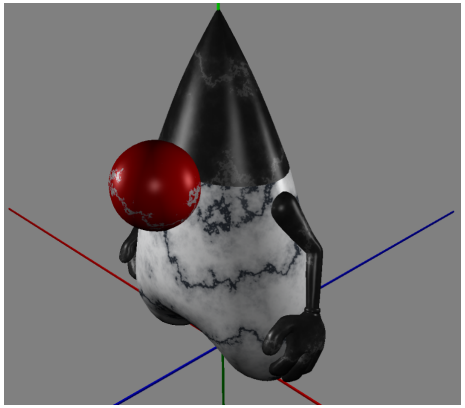
- Gestures
  - Swipe
  - Scroll
  - Rotate
  - Zoom
- Touch events and touch points

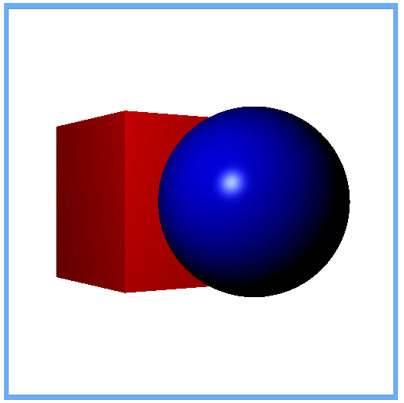
Java™ ORACLE®

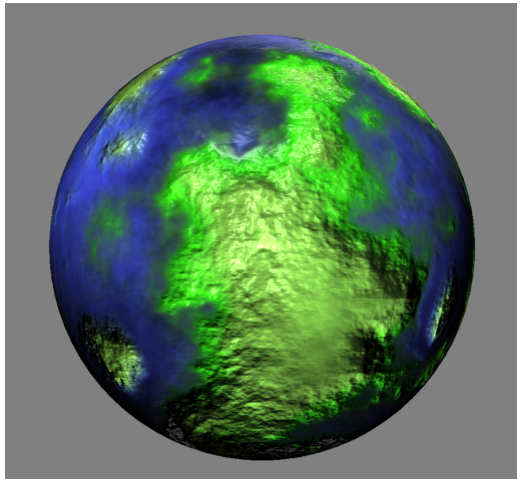
JavaFX 8

3D Support

- Predefined shapes
  - Box
  - Cylinder
  - Sphere
- User-defined shapes
  - `TriangleMesh`, `MeshView`
- `PhongMaterial`
- Lighting
- Cameras

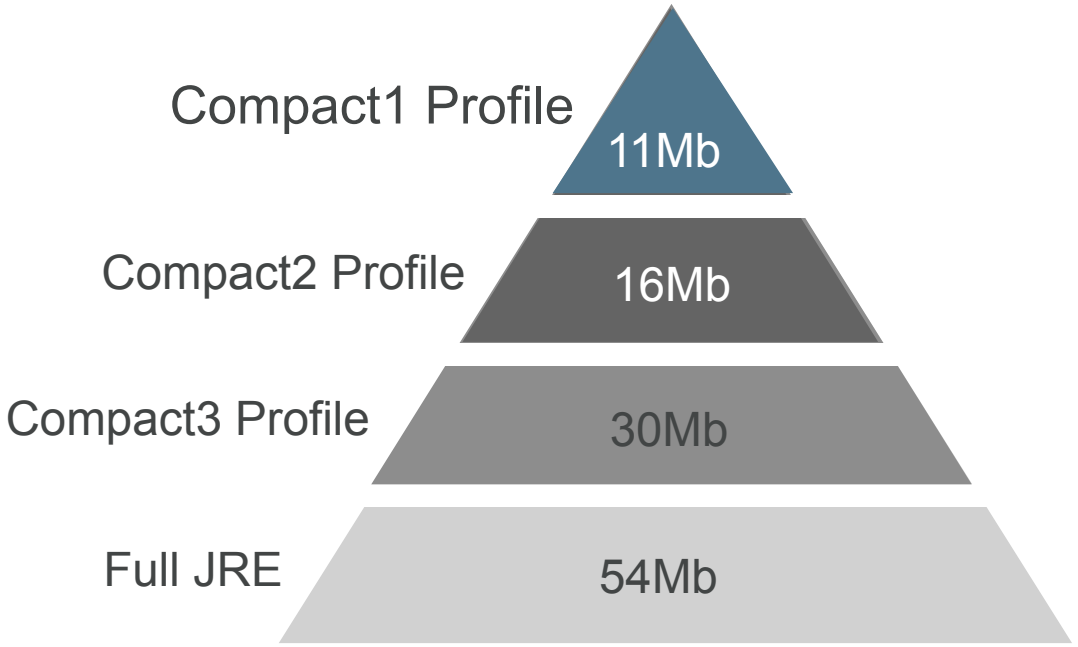






Compact Profiles

Approximate static footprint goals

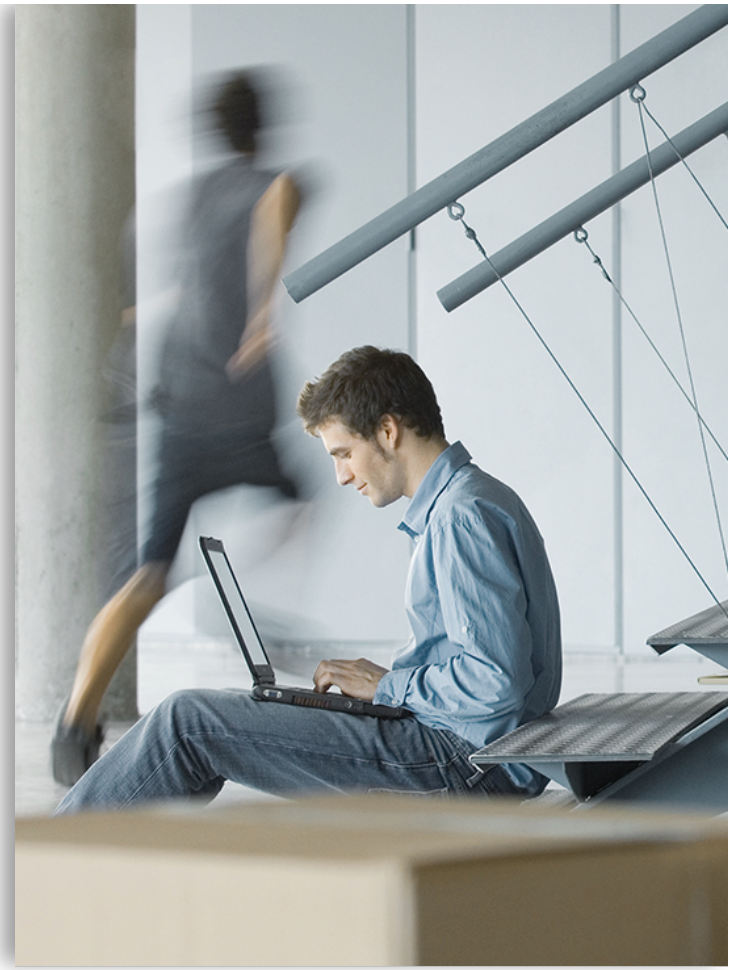


Java SE Embedded

- Optimised binary for embedded use from devices to gateways
- Latest Java innovations in the smallest footprint
- Use of compact profiles
  - Approximate JRE size 20.4Mb (ARM v7 VFS, Hard Float)
    - Compact profile 1: 10.4Mb
    - JavaFX: 10Mb
- Production ready binary for popular platforms that run Linux
  - ARM v6/7 with Hard floating point
  - Raspberry Pi is one of the reference platforms

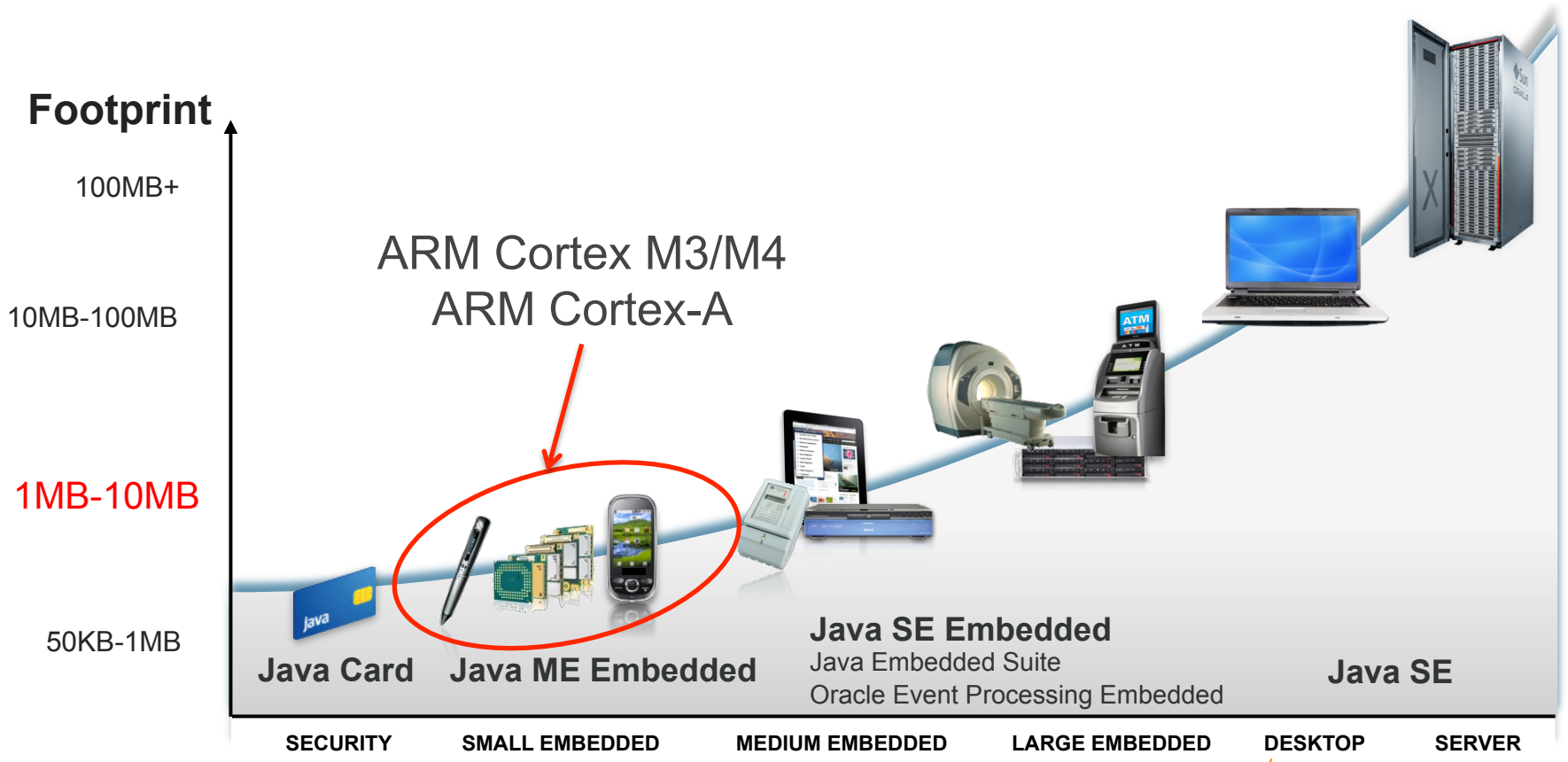
Java™ ORACLE®

Java ME 8: Building The Internet of Things

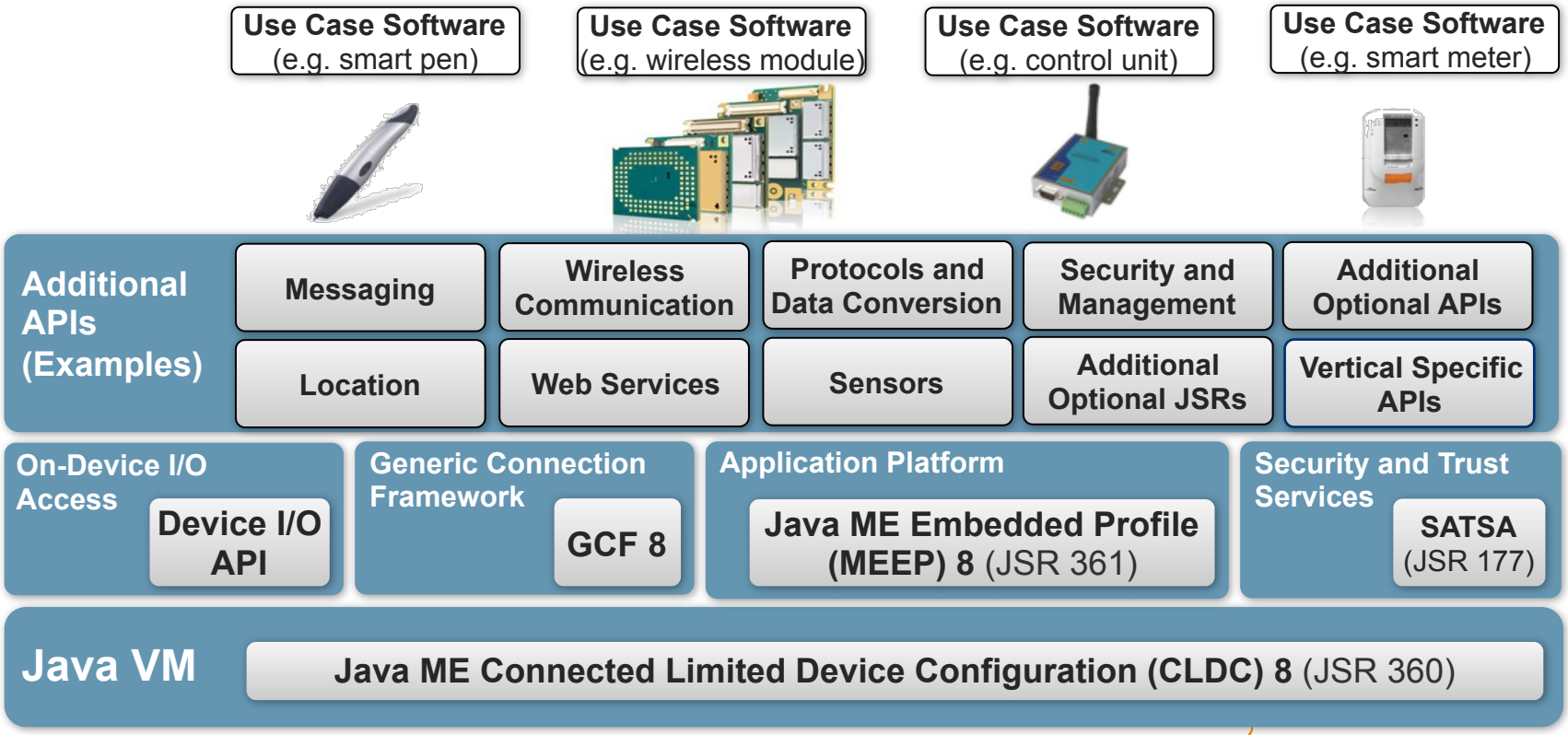


Java™ ORACLE®

Java ME 8 Focus



Java ME 8 Platform Overview



 | |

Java™ ORACLE®

Java ME 8 Key Features

- Aligned with Java SE 8
  - Language, libraries, VM
- Designed for embedded
  - Fully headless operation
  - Remote software provisioning and management
- Highly portable and scalable
  - Minimum RAM footprint of 1Mb
- Consistent across devices

Java ME 8 Key Features

- Advanced application platform
  - Multi-application model
- Modularised software services
  - Faster and more flexible software development and deployment
- Multi-client domains (“partitioning”)
  - Different clients can have different security domains
- Access to peripheral devices (Device I/O API)
- Compatible with JCP/JSR standard APIs

Java™ ORACLE®

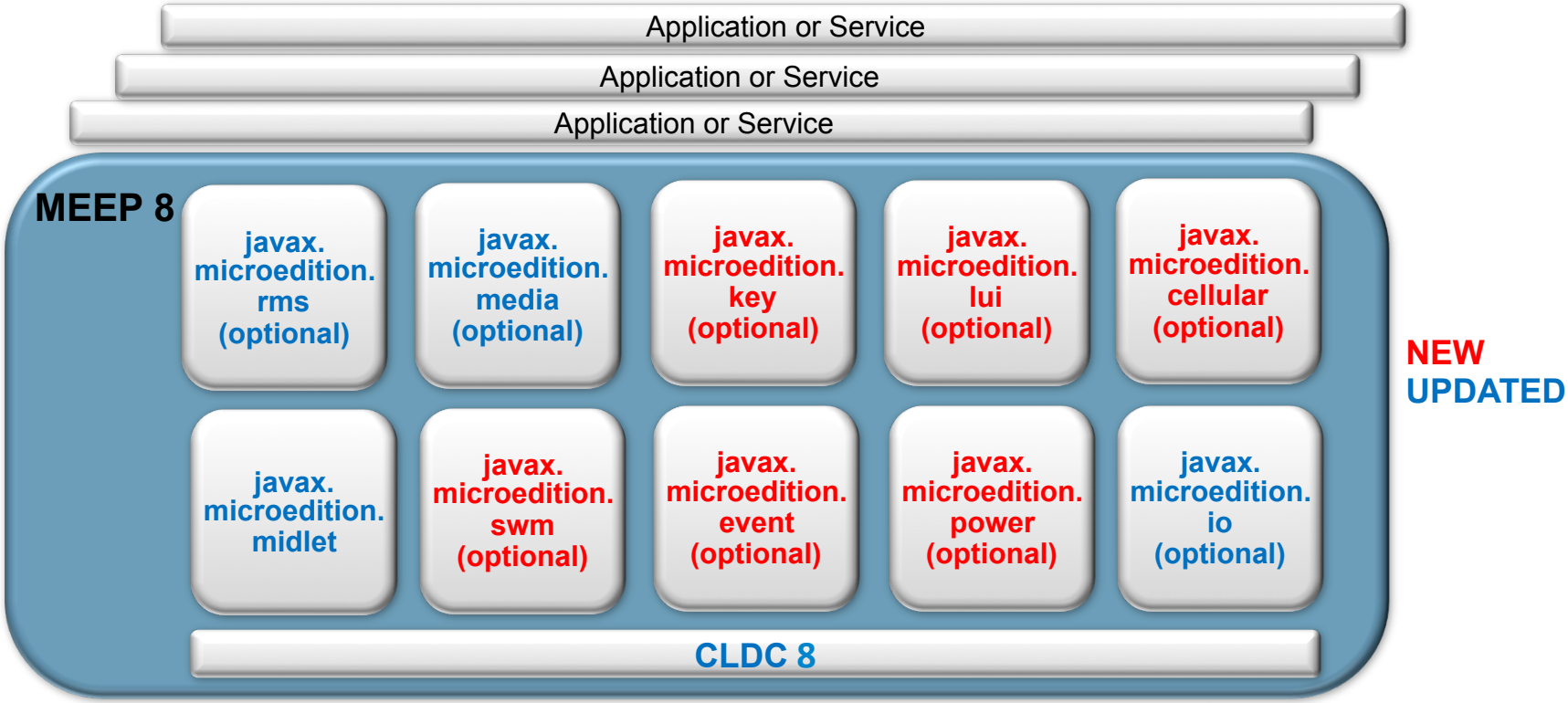
Generic Connection Framework 8

Substantially Increased Range of Connection Types

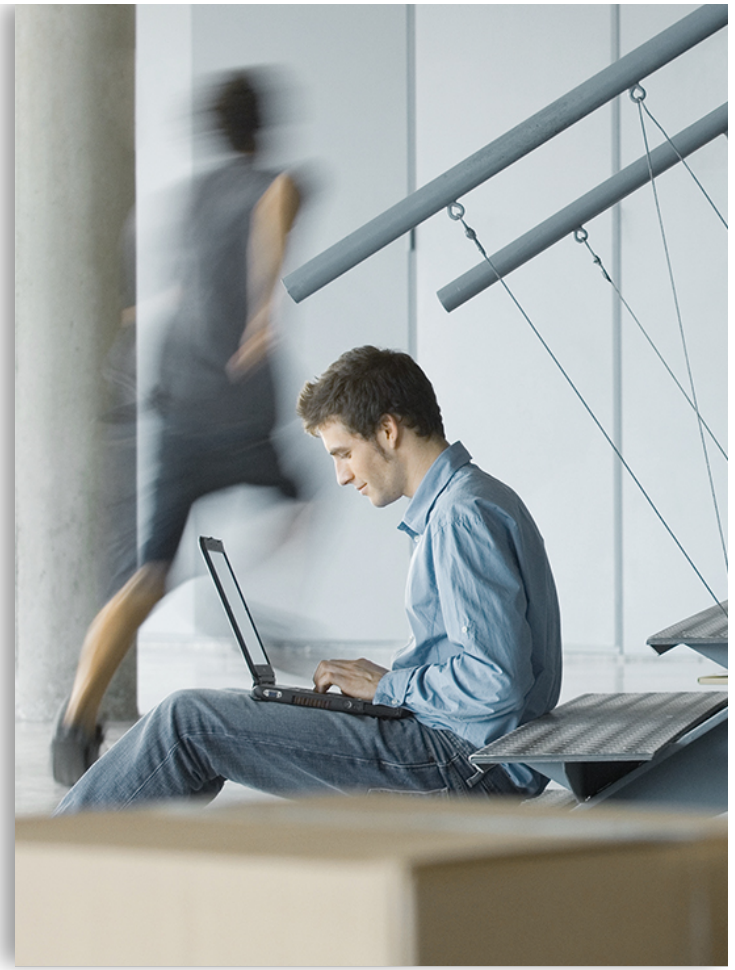
- `SecureServerConnection`
- `SecureDatagramConnection`
- `ModemConnection`
- `UDPMulticastConnection`
- `CommConnection`
- `HttpConnection`
- `HttpsConnection`
- `SecureConnection`
- `ServerSocketConnection`
- `SocketConnection`
- `UDPDatagramConnection`

Java ME Embedded Profile (MEEP) 8

Architecture



NetBeans 8:
The IDE For Java 8

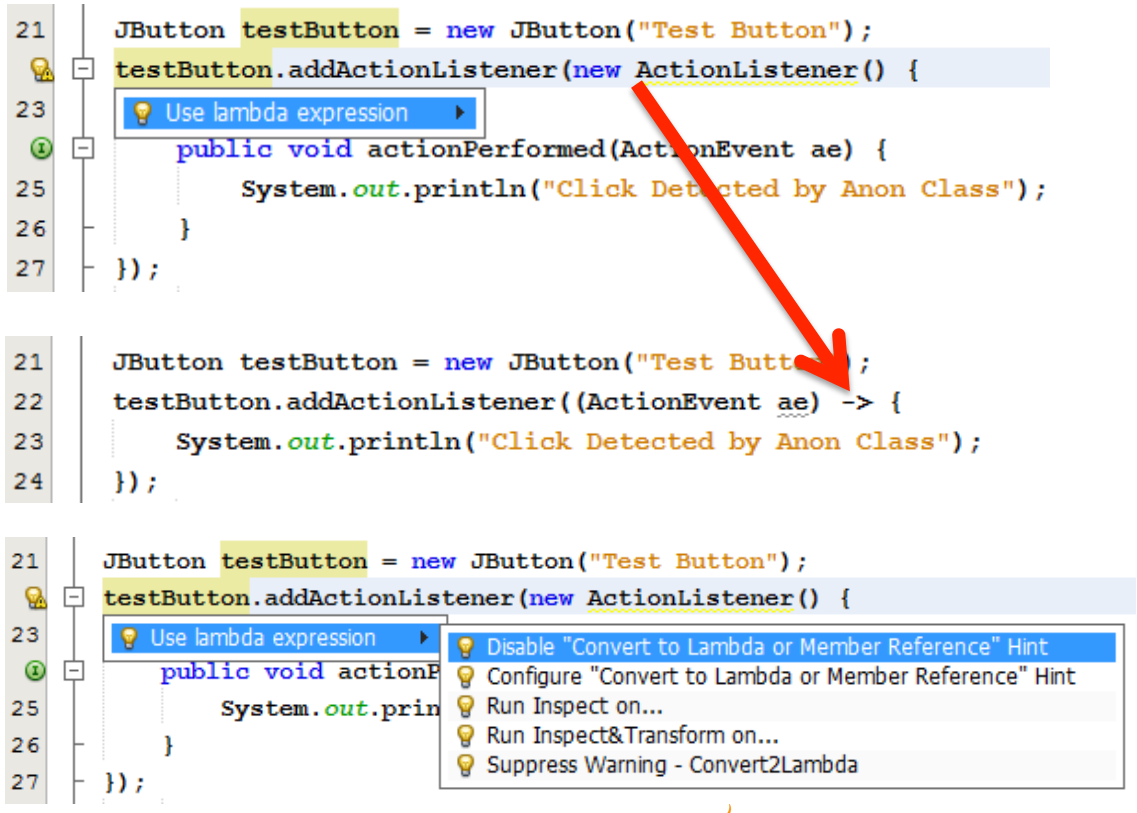


Java™ ORACLE®

Tools for Java SE 8

Lambda Expressions

- Quickly convert anonymous inner classes to lambdas



Java™ ORACLE®

Tools for Java SE 8

Lambda Expressions

- Static analysis to ensure safe transformation, automatically add cast for correct type

```
27    try {
28        conv = AccessController.doPrivileged(
                  new PrivilegedExceptionAction<B2CConverter>() {
30                    @Override
                      public B2CConverter run() throws Exception {
32                        return new B2CConverter(enc);
33                    }
34                });
35    } catch (PrivilegedActionException ex) {
36        Exception e = ex.getException();
37        if (e instanceof IOException) {
38            throw (IOException) e;
39        }
40    }
```

```
27  try {
28      conv = AccessController.doPrivileged(
29              (PrivilegedExceptionAction<B2CConverter>) () -> new B2CConverter(enc));
30  } catch (PrivilegedActionException ex) {
31      Exception e = ex.getException();
32      if (e instanceof IOException) {
33          throw (IOException) e;
34      }
35  }
```

Tools for Java SE 8

Internal Iterators via Java 8 Streams

- Smoothly convert to internal iterators via hints and tips

```
7  public int getNumberOfErrors(String grammarName){
8      int count = 0;
9      for (ElementRule rule : getRules()) {
10         if(rule.hasErrors()){
11             count += rule.getErrorCount();
12         }
13     }
14     return count;
15 }
```

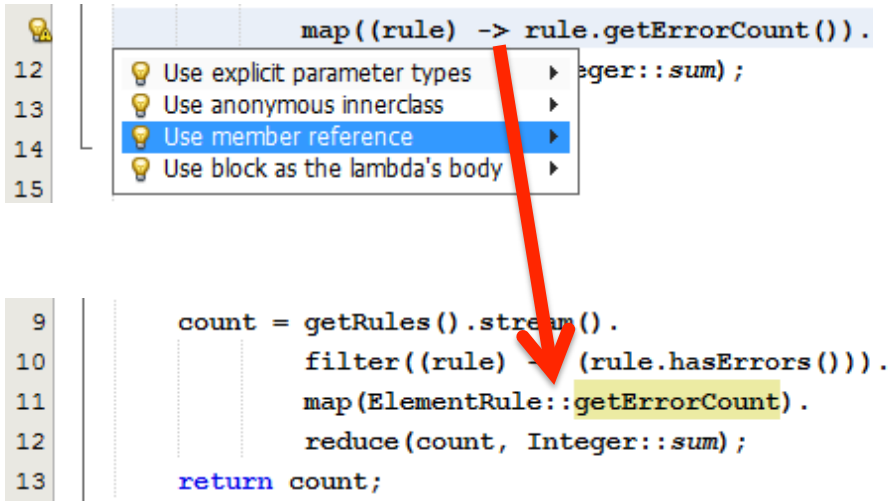
```
7  public int getNumberOfErrors(String grammarName) {
8      int count = 0;
9      count = getRules().stream().
10             filter((rule) -> (rule.hasErrors())).
11             map((rule) -> rule.getErrorCount()).
12             reduce(count, Integer::sum);
13     return count;
14 }
```


Java™ ORACLE®

Tools for Java SE 8

Method References

- Easily convert from lambdas to method references

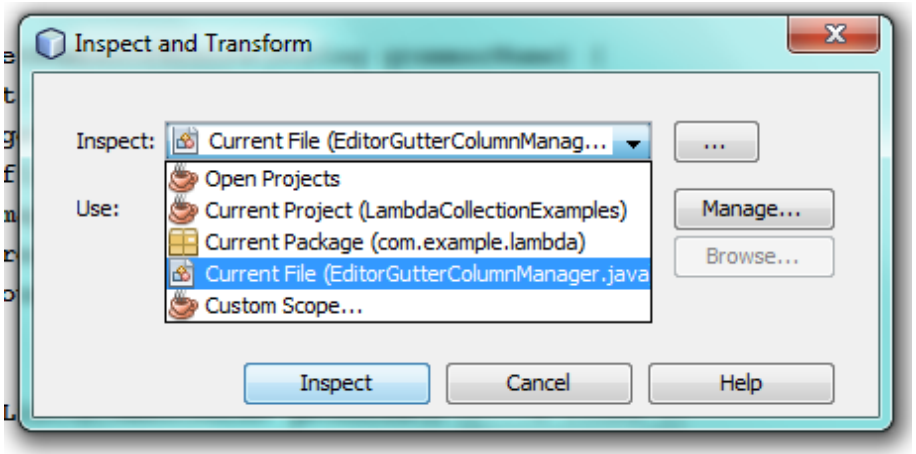


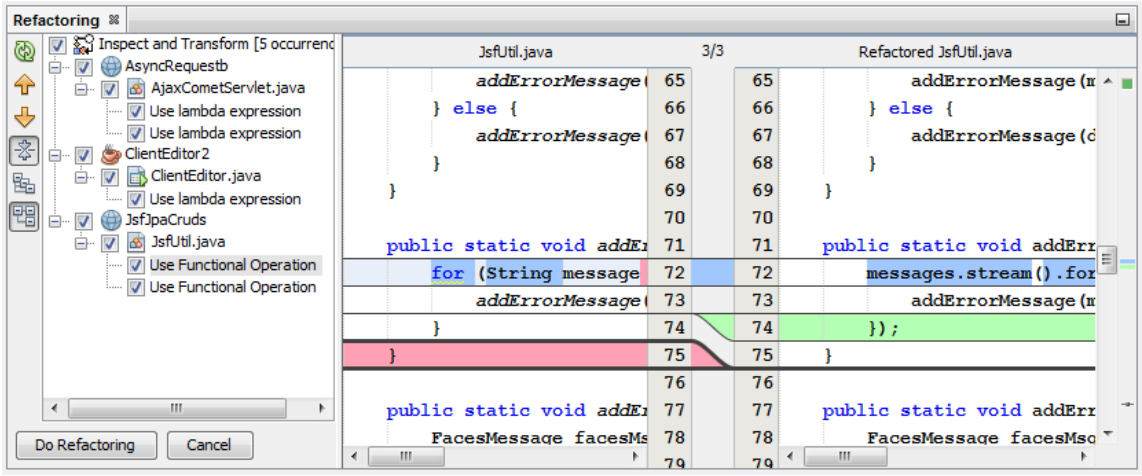
Java™ ORACLE®

Tools for Java SE 8

Refactoring in Batch Mode

- Specify a scope for upgrading to Java 8
  - All/current projects
  - Specific package
  - Specific class
- Run converters
- Visually preview proposals for refactoring





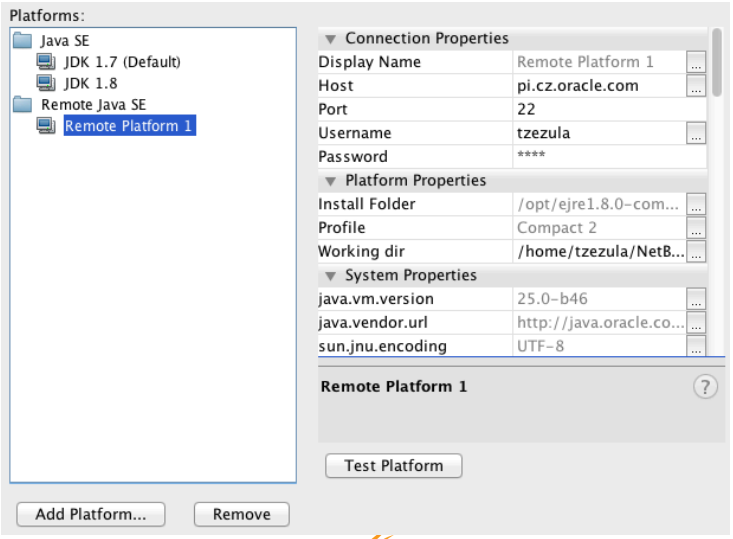
 | |

Java™ ORACLE

Tools for Java SE Embedded 8

Seamless Integration

- Full Development Cycle support against a remote platform
  - Intuitive development
  - One-click deploy
  - Remote debugging
  - Comprehensive remote profiling
- Complete end to end integration for Raspberry Pi and other embedded devices,
  - e.g.Web Services.



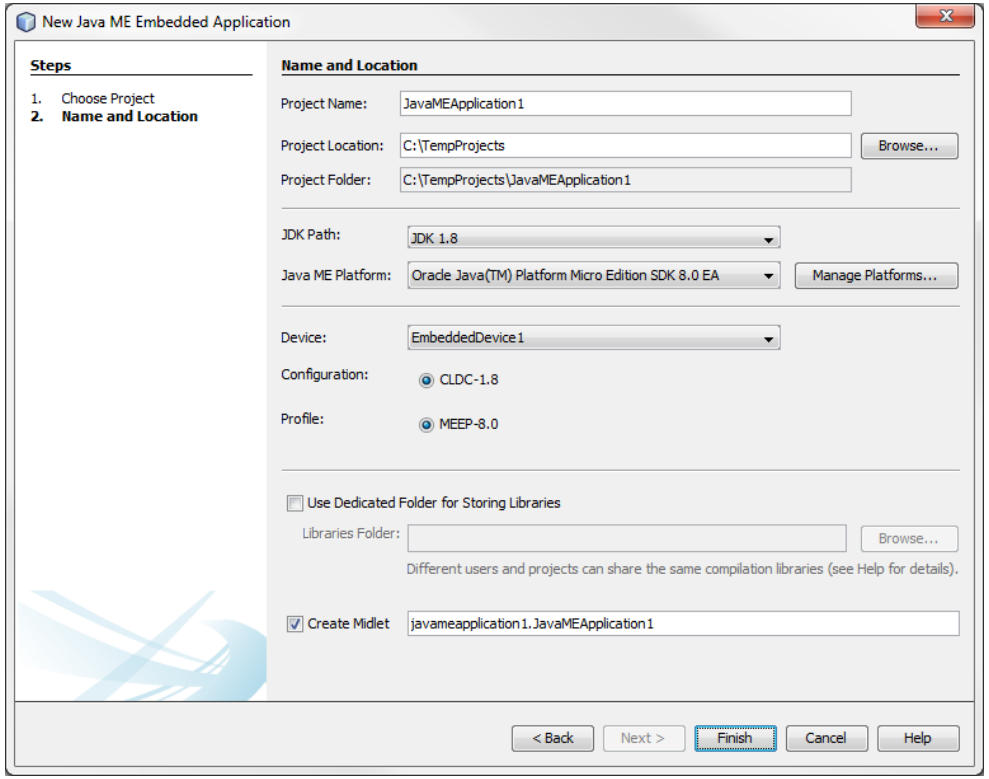
 | |

Java™ ORACLE®

Tools for Java ME Embedded 8

Seamless Integration

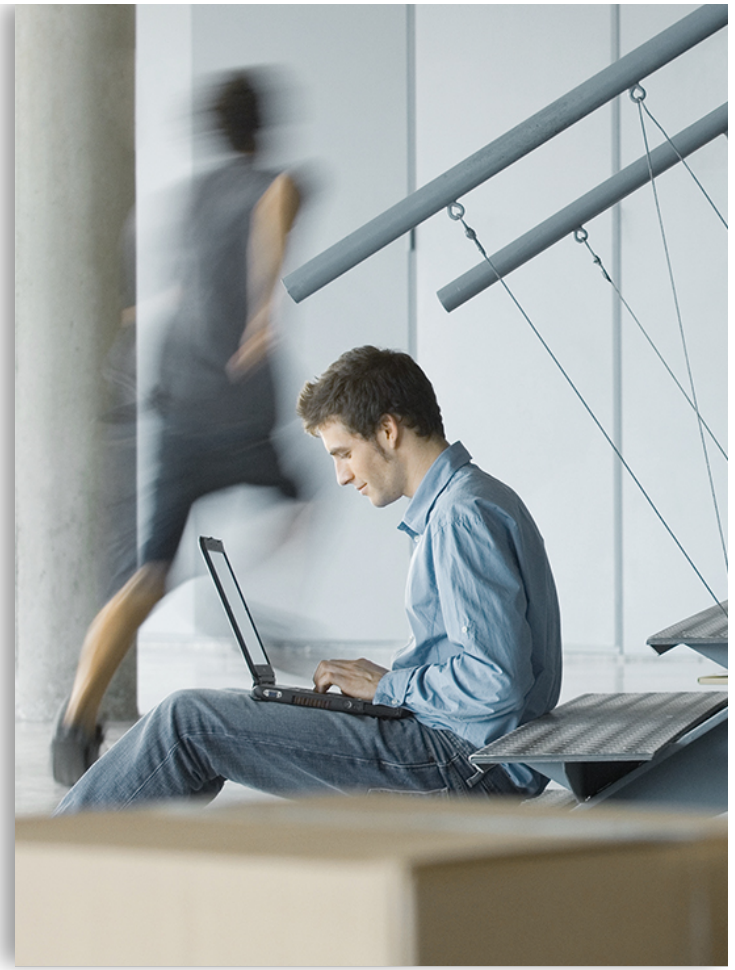
- Java ME 8 CLDC Platform Emulator
- Intuitive tools and editors for JDK 8 on Java ME
- Simple customization of optional packages



 | |

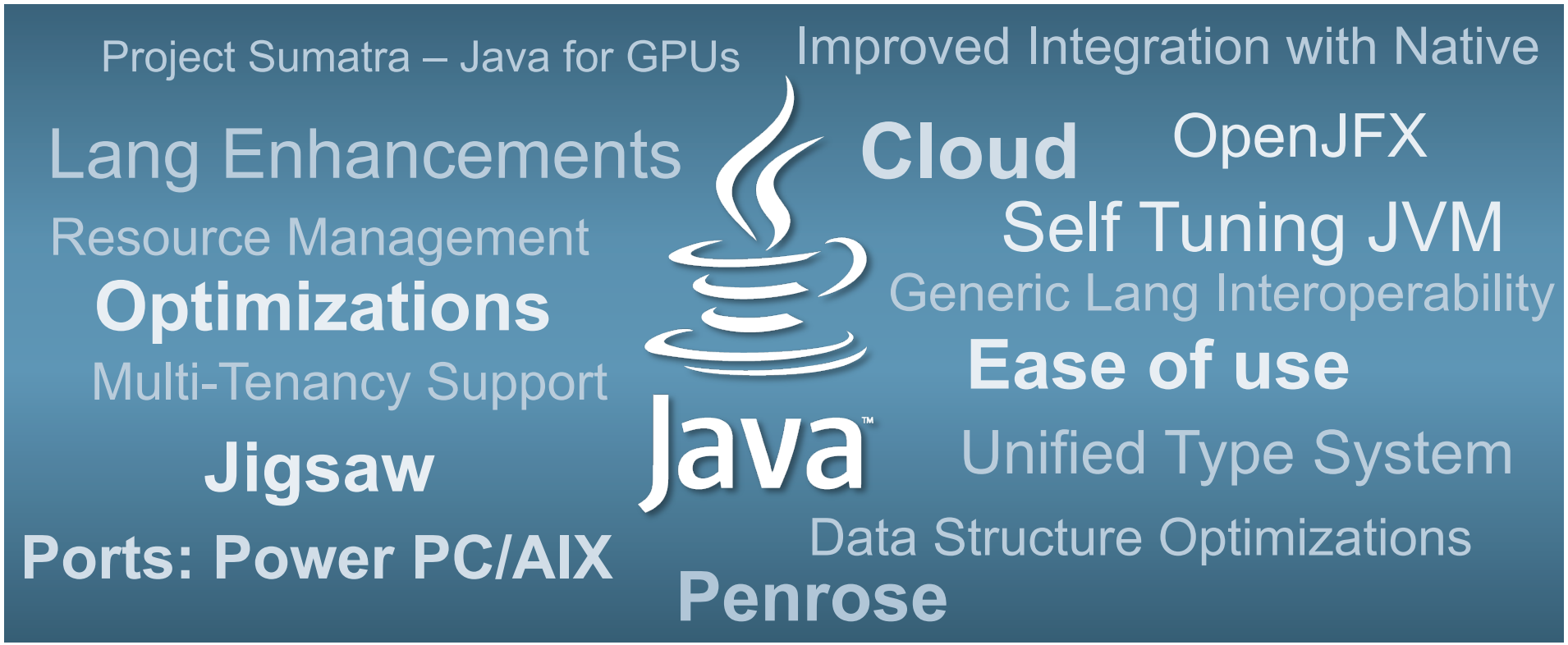
Java™ ORACLE®

Where Next?



Java™ ORACLE®

To Java SE 9 and Beyond!



Java ORACLE

Conclusions

- Java SE 8 adds powerful new features to the core
  - Lambda expressions, streams and functions
  - New controls, stylesheet and 3D support in JavaFX
- Java ME 8 focused on embedded development
  - Device I/O APIs
- NetBeans 8, ready for Java 8 development
  - Lambda support, embedded support
- Java continues to evolve


ORACLE
Java™
CREATE
THE FUTURE