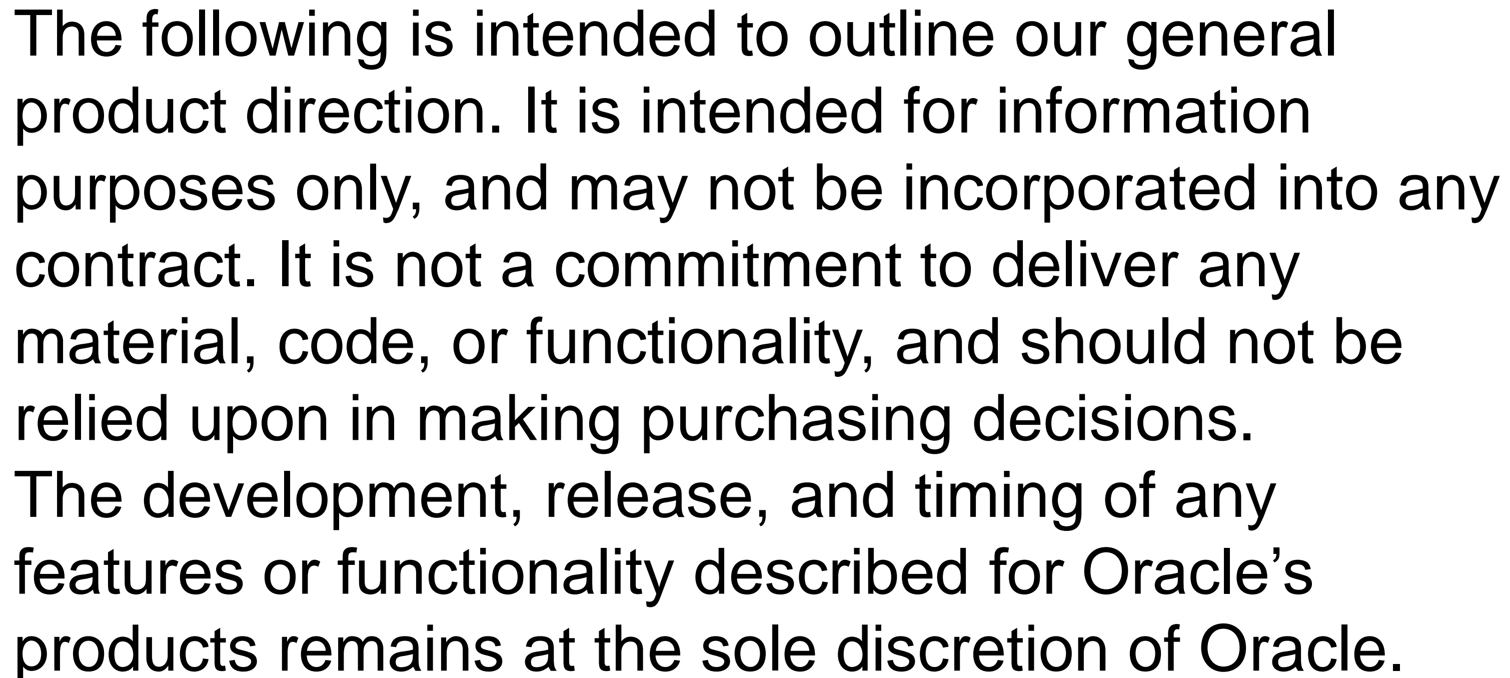
ORACLE®

JSR-335 Update for JCP EC Meeting, January 2012

Alex Buckley
Oracle Corporation



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.

Why closures for Java?

- Help Java programmers easily harness the power of today's multicore processors
- In Java SE 7, the serial code and the parallel code for a given computation look completely dissimilar – a barrier to parallelism
- The idiom of *internal iteration* is key to reducing this barrier
- Closures enable the development of rich, parallel-friendly libraries by supporting internal iteration
- This is not controversial – all other mainstream languages have already embraced closures (C#, VB, JavaScript, Ruby, Obj-C…)

Example: A simple query

"In a music library, get the set of 'favorite' albums where at least one track is highly rated"

```
class Album {
  String      title;
  List<Track> tracks;
}

class Track {
  String title;
  String artist;
  int    rating;
}
```

```
class Library {
  Set<Album> albums;

  Set<Album> favoriteAlbums() {
    // TODO
  }
}
```

Identifying a favorite album

```
// Set hasFavorite to true if some track in album a is rated >= 4

boolean hasFavorite = false;
for (Track t : a.tracks) {
  if (t.rating >= 4) {
    hasFavorite = true;
    break;
  }
}
```

Identifying a favorite album

```
// Set hasFavorite to true if some track in album a is rated >= 4

boolean hasFavorite = false;
for (Track t : a.tracks) {
  if (t.rating >= 4) {
    hasFavorite = true;
    break;
  }
}
```

External iteration

- Client controls iteration
- *Inherently serial:* iterate from beginning to end
- Lots of boilerplate
- Not thread-safe because business logic is stateful

Identifying a favorite album with lambdas

```
// Set hasFavorite to true if some track in album a is rated >= 4

boolean hasFavorite = false;
for (Track t : a.tracks) {
  if (t.rating >= 4) {
    hasFavorite = true;
    break;
  }
}

boolean hasFavorite = a.tracks.anyMatch(t -> t.rating >= 4);
```

Identifying a favorite album with lambdas

```
// Set hasFavorite to true if some track in album a is rated >= 4

boolean hasFavorite = false;
for (Track t : a.tracks) {
  if (t.rating >= 4) {
    hasFavorite = true;
    break;
  }
}
```

Internal iteration

- Iteration / filtering / accumulation controlled by the library
- Not inherently serial
- Thread-safe because business logic is stateless in the client

```
boolean hasFavorite = a.tracks.anyMatch(t -> t.rating >= 4);
```

Making a set of favorite albums

```
// Initialize favs as a set of favorite albums drawn from albums

Set<Album> favs = new HashSet<>();
for (Album a : albums) {
  if (a.tracks.anyMatch(t -> (t.rating >= 4)))
    favs.add(a);
}
```

Making a set of favorite albums

```
// Initialize favs as a set of favorite albums drawn from albums

Set<Album> favs = new HashSet<>();
for (Album a : albums) {
  if (a.tracks.anyMatch(t -> (t.rating >= 4)))
    favs.add(a);
}


Set<Album> favs =
  albums.filter(a -> a.tracks.anyMatch(t -> t.rating >= 4))
        .into(new HashSet<>());
```

Loops v. Lambdas

```
Set<Album> favs = new HashSet<>();
for (Album a : albums) {
  boolean hasFavorite = false;
  for (Track t : a.tracks) {
    if (t.rating >= 4) {
      hasFavorite = true;
      break;
    }
  }
  if (hasFavorite) favs.add(a);
}
```

```
Set<Album> favs =
  albums.filter(a -> a.tracks.anyMatch(t -> t.rating >= 4))
        .into(new HashSet<>());
```

Loops v. Lambdas

Explicit but unobstrusive parallelism

```
Set<Album> favs = new HashSet<>();
for (Album a : albums) {
  boolean hasFavorite = false;
  for (Track t : a.tracks) {
    if (t.rating >= 4) {
      hasFavorite = true;
      break;
    }
  }
  if (hasFavorite) favs.add(a);
}
```

```
Set<Album> favs =
  albums.parallel()
        .filter(a -> a.tracks.anyMatch(t -> (t.rating >= 4)))
        .into(new ConcurrentHashSet<>());
```

The real challenge: Library evolution

- If Java had closures in 1996, APIs would look very different
- Adding closures now, but not evolving core APIs to support them, would be foolish
  - The older APIs get, the more obvious the gaps
  - It is difficult to add entirely new core libraries because the old interfaces (e.g. List) permeate non-core libraries
- Historically, evolving interface-based APIs has been a problem
- Virtual extension methods provide a mechanism for *controlled* evolution of libraries over time
  - Puts burden of evolution on API designers/implementers, not users

JSR-335 features

- Language features
  - Lambda expressions (closures) with “SAM conversion”
  - Method references
  - Virtual extension methods
- Upgraded libraries to use new language features
  - Bulk data operations on Collections
    e.g. filter, map, reduce…
  - “Point lambdafication” of java.util / java.io / java.net
    e.g. “run this closure for every line of a file”
- Synergy with JSR-292 VM enhancements

JSR-335 status

- EDR #1 completed December 2011
  - Specification covers lambda expressions, SAM conversion, method references
  - Prototype of RI compiler available in OpenJDK Project Lambda
- EDR #2 targeted for April 2012
  - Adds type inference and virtual extension methods
- EDR #3 targeted for Summer 2012
  - Adds bulk data operations
  - Initial design is starting now in JSR 166 EG
  - API specification is ultimately expected to go through SE 8 Umbrella JSR