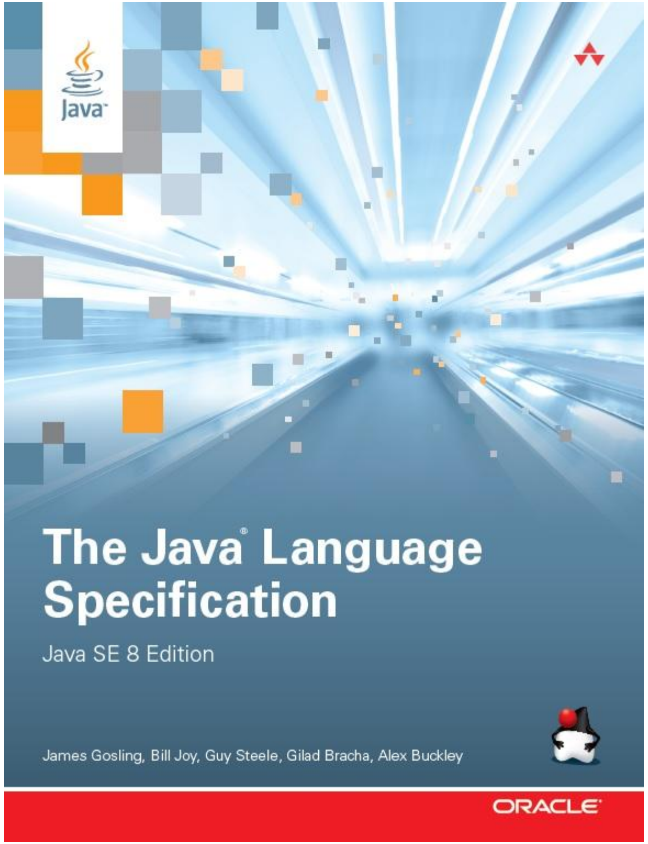
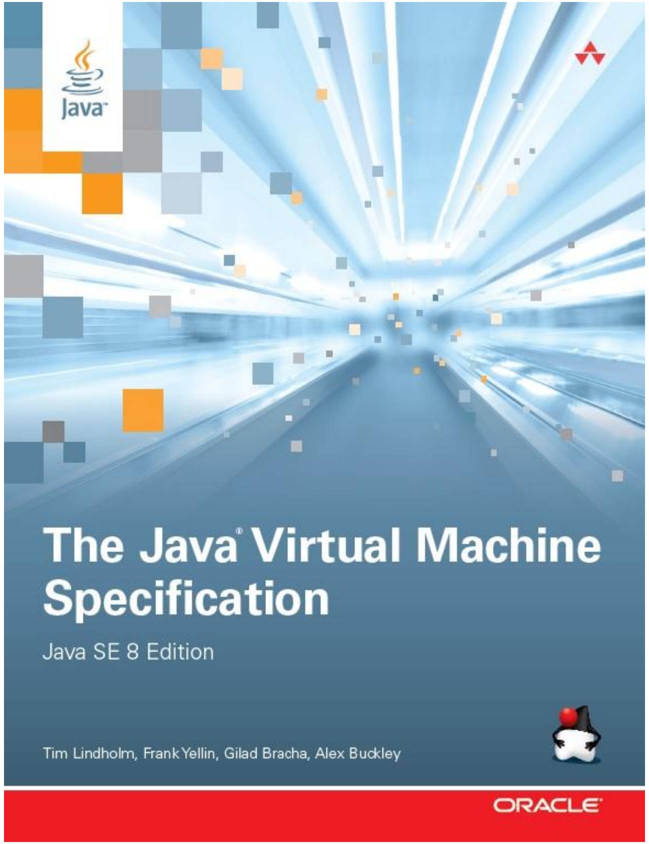
The Road To Lambda

@ Java 8 Day, EclipseCon 2014

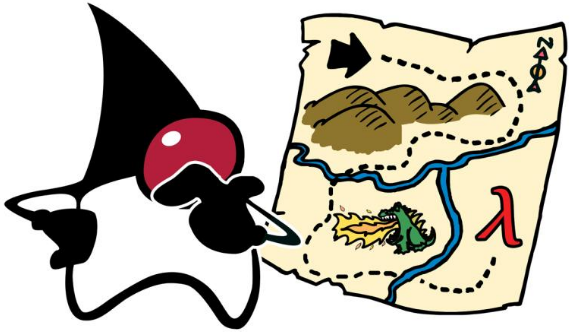
Alex Buckley, Oracle
lambda-dev@openjdk.java.net


Java
The Java® Language Specification
Java SE 8 Edition
James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley
ORACLE


Java
The Java® Virtual Machine Specification
Java SE 8 Edition
Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley
ORACLE

Modernizing Java

- Java SE 8 is a big step forward for the Java Language
  - Lambda Expressions for better abstraction
  - Default Methods for interface evolution
- Java SE 8 is a big step forward for the Java Libraries
  - Bulk data operations on Collections
  - More library support for parallelism
- Together, perhaps the *biggest upgrade ever* to the Java programming model
- Why did we choose the features we did?
- How do we evolve a mature language?



The Language

What is a Lambda Expression?

- A lambda expression is an anonymous method
  - Has a parameter list, a return type, and a body

    `(Object o) -> o.toString()`

  - Body can refer to *effectively final* variables in the enclosing lexical scope

    `(Person p) -> p.getName().equals(name)`

- A method reference is a reference to an existing method

    `Object::toString`

- Allow you to *treat code as data*
  - Behavior can be expressed succinctly, stored in variables, and passed to methods
  - A huge deal because of the impact on library design

What is the Type of a Lambda Expression?

- Many languages have some notion of a *function type*
  - “Function from long to int”
  - Seemed reasonable (at first) to consider adding them to Java
- But…
  - JVM has no native representation of function type in VM type signatures
  - Obvious tool for representing function types is generics
    - But then function types would be erased (and boxed)
  - Is there a simpler alternative?

Functional Interfaces

- Historically we used single-method interfaces to represent functions
  - Runnable, Comparator, ActionListener, FileFilter, …
  - Let’s call them *functional interfaces*
  - And add some new ones like Predicate<T>, Consumer<T>, Supplier<T>
- A lambda expression evaluates to an instance of a functional interface

```
Predicate<String> isEmpty = s -> s.isEmpty();
Predicate<String> isEmpty = String::isEmpty;
Runnable r = () -> { System.out.println("Boo!"); };
```

- We define functional interfaces *structurally*
  - No syntax or opt-in needed
  - Existing libraries are forward-compatible with lambda expressions

Times Change

- In 1995, most popular languages did *not* support lambda expressions
- By 2013, Java was just about the last holdout
  - C# added them in 2007, Objective-C in 2010, C++ in 2011
  - New languages being designed today all do

"In another thirty years people will laugh at anyone who tries to invent a language without closures, just as they'll laugh now at anyone who tries to invent a language without recursion."
- Mark Jason Dominus

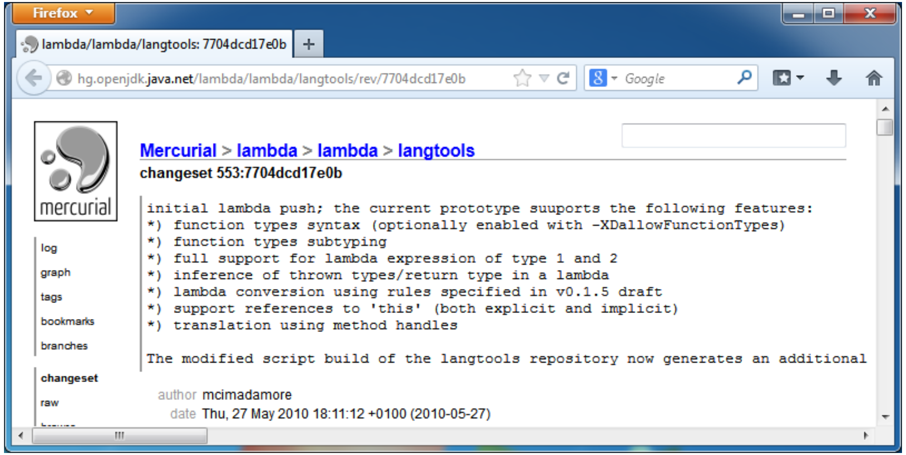
Lambdas for Java – a long and winding road

- 1997 – Odersky / Wadler experimental “Pizza” work
  - “IMHO Pizza makes finally a usefull language out of java” comp.lang.java 3/6/97
- 1997 – Java 1.1 added inner classes
  - Too bulky, complex name resolution rules, many limitations
- 2006-2008 – vigorous community debate
  - Multiple proposals, including BGGA and CICE
  - Each had a different orientation
    - BGGA – facilitating control abstraction in libraries
    - CICE – reducing syntactic overhead of inner classes
  - Things ran aground at this point…

ORACLE

Lambdas for Java – a long and winding road

- 2009-Dec – OpenJDK Project Lambda formed
- 2010-May – First prototype
- 2010-Nov – JSR-335 filed
- 2011-Nov – Early Draft Review #1
- 2011-Nov – Developer Preview binaries on java.net
- 2012-Jun – Early Draft Review #2
- 2013-Feb – Early Draft Review #3
- 2014-Jan – Proposed Final Draft



ORACLE

Evolving a Mature Language – Key Forces

- Encouraging change
  - Adapting to change
    - Everything changes: hardware, attitudes, fashions, problems, demographics
  - Righting what's wrong
    - Inconsistencies, holes, poor user experience
- Discouraging change
  - Maintaining compatibility
    - Low tolerance for change that will break anything
  - Preserving the "feel of Java"
    - Can't alienate user base in quest for "something better"
    - Easy to focus on cool new stuff, but there's lots of cool old stuff too

ORACLE

Lambdas: Adapting to Change

- In 1995, pervasive sequentiality infected programming language design
  - for-loops are sequential and impose a specific order
    - Why wouldn't they be? Why invite nondeterminism?
    - Determinism is convenient – when free
    - This sequentiality assumption propagated into libraries (e.g., Iterator)
  - Pervasive mutability
    - Mutability is convenient – when free
    - Object creation was expensive and mutation was cheap
- In today's multicore world, these are the wrong defaults!
  - Can't just outlaw for-loops and mutability
  - Instead, gently *encourage* something better
- Lambda expressions are that gentle push

ORACLE

Problem: External Iteration

- Snippet takes the red blocks and colors them blue
- Uses for-each loop
  - Loop is *inherently sequential*
  - Client has to manage iteration
  - This is called *external iteration*

```
for (Shape s : shapes) {
    if (s.getColor() == RED)
        s.setColor(BLUE);
}
```

- for-each loop hides complex interaction between library and client
  - Iterable, iterator(), Iterator.next(), Iterator.hasNext()
- What's the problem? Conflates the *what* with the *how*.
- A language construct that is inherently sequential is a significant problem.

Solution: Internal Iteration

- Re-written to use lambda and Collection.forEach
  - Not just a syntactic change!
  - Now the library is in control
  - This is *internal iteration*
  - More *what*, less *how*

```
shapes.forEach(s -> {
    if (s.getColor() == RED)
        s.setColor(BLUE);
})
```

- Library is free to use parallelism / out-of-order execution / laziness
- Client passes behavior (lambda) into the API as data
- Enables API designers to build more powerful, expressive APIs
  - Greater power to abstract over behavior

ORACLE

Lambdas & Libraries

Lambdas Enable Better APIs

- Lambda expressions *enable delivery of more powerful APIs*
- The client-library boundary is more permeable
  - Client can provide bits of functionality to be mixed into execution
  - Client determines the *what*
  - Library remains in control of the *how*
- Safer: less state management in the client
- Faster: exposes more opportunities for optimization

Example: Sorting

- If we want to sort a List today, we'd write a Comparator

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

- Could replace Comparator with a lambda, but only gets us so far
- Comparator conflates *extraction of sort key* with *ordering* of that key
  - Better to separate the two aspects

ORACLE

Example: Sorting

- Added static method Comparator.comparing(f)
  - Takes a “key extractor” function from T to some Comparable key
  - Returns a Comparator<T>
  - This is a *higher-order function* – functions in, functions out

```
interface Comparator<T> {
    public static <T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> f) {
        return (x, y) -> f.apply(x).compareTo(f.apply(y));
    }
}
Comparator<Person> byLastName
      = Comparator.comparing(Person::getLastName);
```


ORACLE

Lambdas Enable Better APIs

- The comparing() method is one built for lambdas
  - Consumes an “extractor” function and produces a “comparator” function
  - Factors key extraction (client concern) from comparison (library concern)
  - Eliminates redundancy, boilerplate
- Key effect on APIs is: *more composability*
  - Centralize generation of Comparators in one place
  - Leads to better factoring, more regular client code, more reuse
- Lambdas in the language

  → can write better libraries

  → more readable, less error-prone user code

ORACLE

Lambdas Enable Better APIs

- Generally, we prefer to evolve the programming model through libraries
  - Time to market – can evolve libraries faster than language
  - Decentralized – more library designers than language designers
  - Risk – easier to change libraries, more practical to experiment
  - Impact – language changes require coordinated changes to multiple compilers, IDEs, and other tools
- But sometimes we reach the limits of what is practical to express in libraries, and need a little help from the language
  - But a little help, in the right places, can go a long way!

Problem: Interface Evolution

- The example used a new Collection method – forEach()
  - If Java had lambdas in 1997, our Collections would surely look different
- Interfaces are a double-edged sword
  - Cannot compatibly evolve them unless you control all implementations
  - Reality: APIs age
    - As we add cool new language features, existing APIs look even older!
  - Lots of bad options for dealing with aging APIs
    - Let the API stagnate
    - Replace it in entirety (every few years!)
    - Nail bags on the side (e.g., Collections.sort())

Default Methods

- Need a proper mechanism for compatibly evolving APIs
- New feature: *default methods*
  - Virtual interface method with default implementation
  - “default” is the dual of “abstract”

```
interface Collection<T> {
  default void forEach(Consumer<T> action) {
    for (T t : this) { action.apply(t); }
  }
}
```

- Lets us compatibly evolve libraries over time
  - Default implementation provided in the interface
  - Subclasses can override with better implementations
  - Adding a default method is binary-compatible *and source-compatible*

ORACLE

Default Methods

- Is this multiple inheritance in Java?
  - Java always had multiple inheritance of *abstract methods*
  - This adds multiple inheritance of *behavior*
  - But not of *state*, where most of the trouble comes from
- Compared to C# extension methods:

  Java's default methods are *virtual* and *declaration-site,* not *static* and *use-site*
- Compared to Scala's Traits:

  Java interfaces are stateless (more like Fortress' Traits)
- How do we resolve conflicts between declarations in multiple supertypes?
  - Three simple rules

ORACLE

Rule #1: Class Wins

- If a class can inherit a method from a superclass and a superinterface, prefer the superclass method
  - Defaults *only* considered if no method declared in superclass chain
  - True for both concrete and abstract superclass methods
- Ensures compatibility with pre-SE 8 inheritance
  - Any call site that linked under previous rules links to the same target
- Otherwise…

Rule #2: Subtypes Win

- If a class can inherit a method from two interfaces, and one is more specific than (a subtype of) the other, prefer the more specific
  - An implementation in List would take precedence over one in Collection
- The shape of the inheritance tree doesn't matter
  - Only consider the set of supertypes, not order in which they are inherited
- Otherwise…

Rule #3: There is No Rule 3

- If rule #1 does not apply, and rule #2 does not yield a *unique, most specific default-providing interface…*
  - Explicitly reabstract it
  - Implement the method yourself
  - Implementation can delegate to non-inherited implementation with new construct ***X*.super.m()**

```
interface A {
    default void m() { ... }
}
interface B {
    default void m() { ... }
}
class C implements A, B {
    // Must implement/reabstract m()
    void m() { A.super.m(); }
}
```


Diamonds – No Problem

- Diamonds do not pose a problem for behavior inheritance
  - More problematic for state inheritance
- For D, there is a unique most-specific default-providing interface – A
  - D inherits m() from A, via two paths
  - “Redundant” inheritance does not affect the resolution

```
interface A {
    default void m() { ... }
}
interface B extends A { }
interface C extends A { }
class D implements B, C { }
```


ORACLE

Example – Evolving Interfaces

- Adding a new method with default is source- and binary-compatible
- Default methods are instance methods
  – Type of ‘this’ is the declaring interface
  – Default implementation can invoke methods from enclosing interface, e.g. iterator()

```
interface Collection<E> {
    default boolean removeIf(Predicate<? super E> filter) {
        boolean removed = false;
        Iterator<E> it = iterator();
        while (it.hasNext()) {
          …
        }
        return removed;
    }
}
```


ORACLE

Example – "Optional" Methods

- Adding a default *to an existing method* is source- and binary-compatible
- Default methods can reduce implementation burden
  - Most implementations of Iterator don't provide a useful remove()
  - So why make implementers write one that just throws?

```
interface Iterator<T> {
    boolean hasNext();

    T next();

    default void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Example – Combinators

- Comparator.reversed() – reverses sort order of a Comparator
  - Default method on Comparator, just invokes compare()

```
interface Comparator<T> {
    default Comparator<T> reversed() {
        return (o1, o2) -> compare(o2, o1);
    }
}
Comparator<Person> byLastNameDescending
      = Comparator.comparing(Person::getLastName).reversed();
```

The Java Libraries

Bulk operations on Collections

- “Color the red blocks blue” can be decomposed into filter+forEach

```
shapes.forEach(s -> {
    if (s.getColor() == RED)
        s.setColor(BLUE);
})
```

⬇

```
shapes.stream()
      .filter(s -> s.getColor() == RED)
      .forEach(s -> { s.setColor(BLUE); });
```

- Why is this code better? Each part does one thing, and is clearly labeled.

 | |

ORACLE

Bulk operations on Collections

- Collect the blue Shapes into a List

```
List<Shape> blueBlocks
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .collect(Collectors.toList());
```

- If each Shape lives in a Box, find Boxes containing a blue shape

```
Set<Box> hasBlueBlock
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .map(Shape::getContainingBox)
            .collect(Collectors.toSet());
```

Bulk operations on Collections

- The new bulk operations are expressive and composable
  - Compose compound operations from basic building blocks (lambdas)
  - Each stage does one thing
  - Client code reads more like the problem statement
  - Structure of client code is less brittle (anti-fragile)
    - Small changes to the problem -> small changes to the code
  - Less extraneous “noise” from intermediate results
    - No ‘accumulator’ or state variables

Which do you prefer?

```
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Group>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

- Or…

```
txns.stream()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sort(comparing(Seller::getName))
    .forEach(s -> System.out.println(s.getName());
```


From Collections to Streams

- To add bulk operations, we create a new abstraction: Stream
  - Represents a stream of values
  - Not a data structure – doesn't store the values
  - Source can be a Collection, array, generating function, I/O…
- Operations that produce new streams are lazy
  - Encourages a "fluent" usage style
  - Efficient – does a single pass on the data

```
collection.stream()
          .filter(f::isBlue)
          .map(f::getBar)
          .forEach(System.out::println);
```


ORACLE

Comparing Approaches

Collections	Streams
Code deals with individual data items	Code deals with data set
Focused on *how*	Focused on *what*
Code doesn’t read like the problem statement	Code reads like the problem statement
Steps mashed together	Well-factored
Leaks extraneous details	No “garbage variables”
Inherently sequential	Same code can be sequential or parallel

Parallelism

Parallelism

- Goal: Handle parallelism in the libraries, not the language
  - Libraries can hide a host of complex concerns (task scheduling, thread management, load balancing)
- Goal: Reduce conceptual and syntactic gap between sequential and parallel forms of the same computation
  - Historically, sequential and parallel code for a given computation don't look anything like each other

ORACLE

Obtrusive Parallelism

- Java SE 7 has a general-purpose Fork/Join framework
  - Powerful and efficient, but not so easy to program to
  - Based on recursive decomposition
    - Divide problem into subproblems, solve in parallel, combine results
    - Keep dividing until small enough to solve sequentially
  - Tends to be efficient across a wide range of processor counts
  - Generates reasonable load balancing with no central coordination

ORACLE

Parallel Sum with Fork/Join (SE 7)

```
ForkJoinPool pool = new ForkJoinPool(nThreads);
SumFinder finder = new SumFinder(new SumProblem());
pool.invoke(finder);

class SumFinder extends RecursiveAction {
  private final SumProblem problem;
  int sum;

  protected void compute() {
    if (problem.size < THRESHOLD)
      sum = problem.solveSequentially();
    else {
      int m = problem.size / 2;
      SumFinder left, right;
      left = new SumFinder(problem.subproblem(0, m))
      right = new SumFinder(problem.subproblem(m, problem.size));
      forkJoin(left, right);
      sum = left.sum + right.sum;
    }
  }
}
```

```
class SumProblem {
  final List<Shape> shapes;
  final int size;

  SumProblem(List<Shape> ls) {
    this.shapes = ls;
    size = ls.size();
  }

  public int solveSequentially() {
    int sum = 0;
    for (Shape s : shapes) {
      if (s.getColor() == BLUE)
        sum += s.getWeight();
    }
    return sum;
  }
  public SumProblem subproblem(int start, int end) {
    return new SumProblem(shapes.subList(start, end));
  }
}
```


Parallel Sum with Streams (SE 8)

- Sequential sum-of-weights:

```
int sumOfWeight
    = shapes.stream()
            .filter(s -> s.getColor() == BLUE)
            .mapToInt(Shape::getWeight)
            .sum();
```

- Parallel sum-of-weights:

```
int sumOfWeight
    = shapes.parallelStream()
            .filter(s -> s.getColor() == BLUE)
            .mapToInt(Shape::getWeight)
            .sum();
```

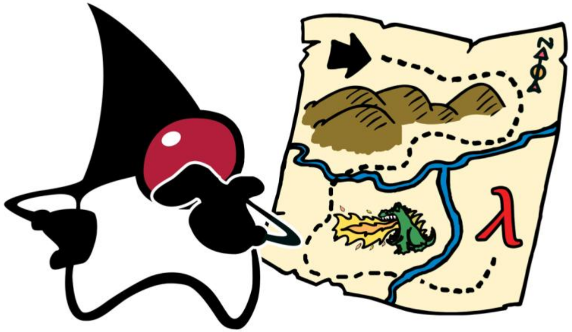
- Explicit but unobtrusive parallelism
- All three operations fused into a single parallel pass

So … Why Lambda?

- It's about time!
  - Java was the lone holdout among mainstream OO languages over lambdas
  - Adding them to Java is no longer a radical idea
- Lambdas provide libraries with a path to multicore
  - Parallel-friendly APIs need internal iteration
  - Internal iteration needs a concise code-as-data mechanism
- Lambdas empower library developers
  - More powerful, flexible libraries
  - Higher degree of cooperation between libraries and client code
  - Better libraries means more expressive, less error-prone code for users!

Modernizing Java

- Java SE 8 is a big step forward for the Java Language
  - Lambda Expressions for better abstraction
  - Default Methods for interface evolution
- Java SE 8 is a big step forward for the Java Libraries
  - java.util.stream.* + java.util.function.*
  - Upgrades throughout java.util.*
- Together, perhaps the *biggest upgrade ever* to the Java programming model
- **Shipping today!**



Books

- “Java SE 8 for the Really Impatient” (Horstmann)
- “Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions” (Subramaniam)
- “Mastering Lambdas: Java Programming in a Multicore World” (Naftalin)
- “Java 8 in Action: Lambdas, Streams, and functional-style programming” (Urma, Fusco, Mycroft)
- “Java 8 Lambdas: Pragmatic Functional Programming” (Warburton)
- “Java in a Nutshell” (Evans, Flanagan)

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.