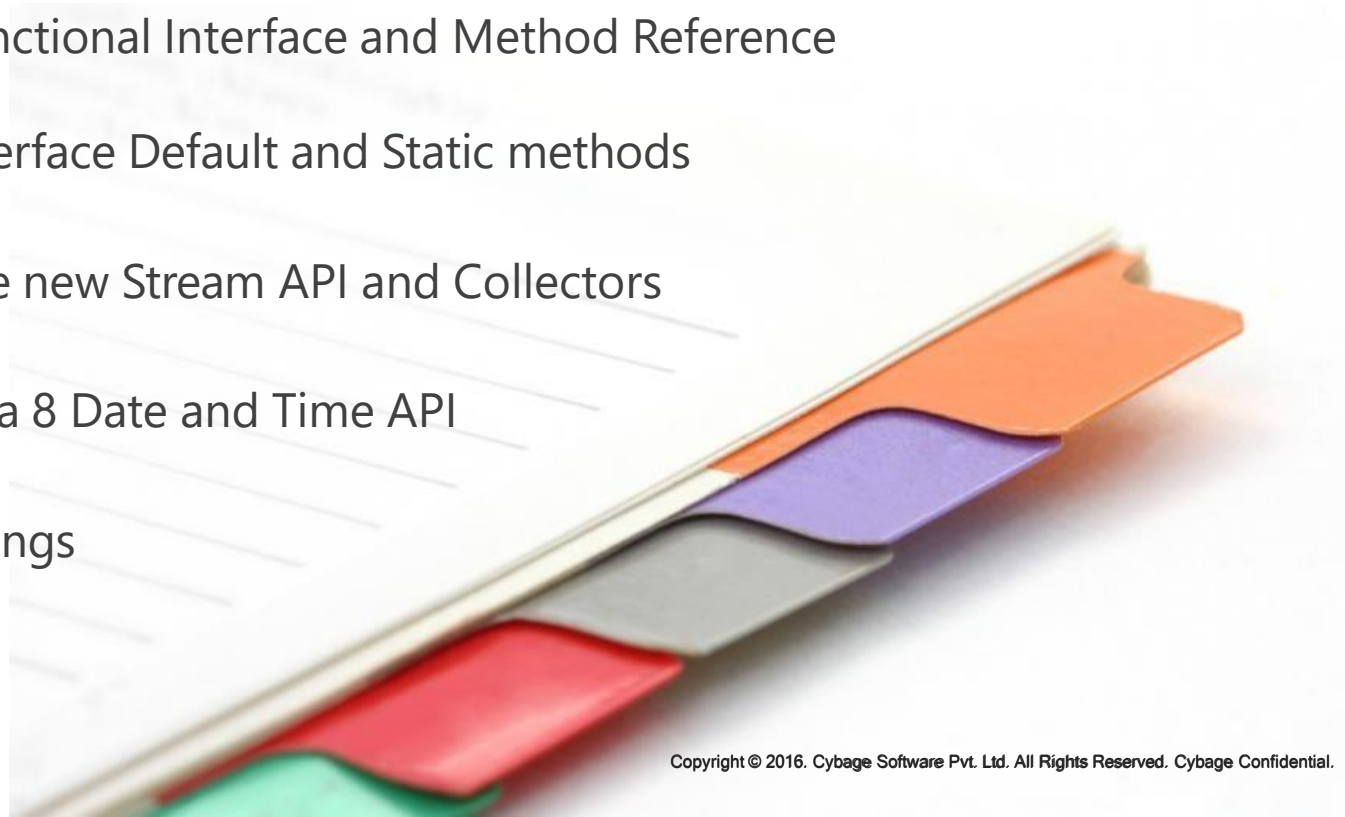
JAVA 8

Cybage®
Delivering Value Scientifically

Agenda

- Introduction to new features of JAVA8
- Lambda Expressions in JAVA8
- Functional Interface and Method Reference
- Interface Default and Static methods
- The new Stream API and Collectors
- Java 8 Date and Time API
- Strings



Cybage®
Delivering Value Scientifically

Introduction to New JAVA8

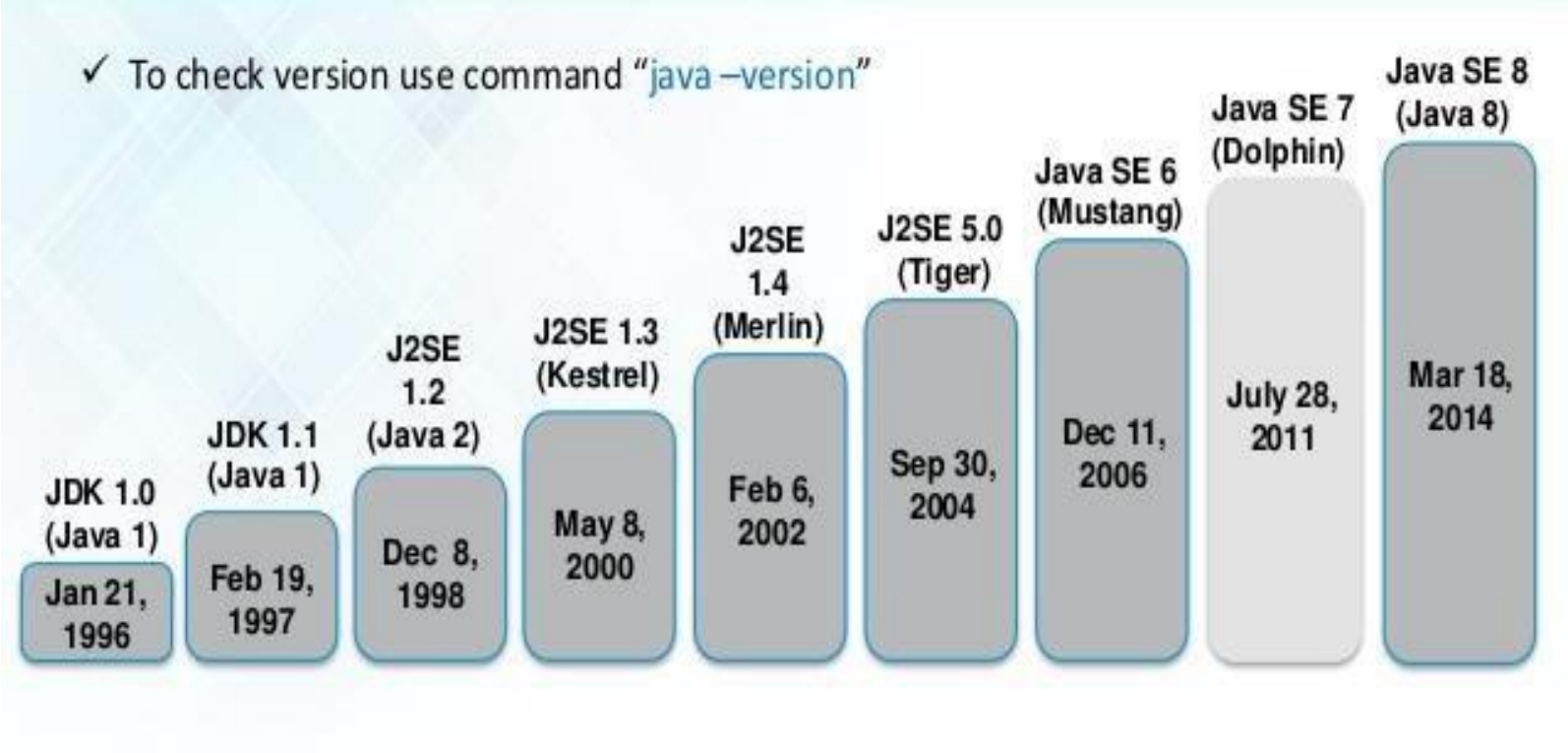
- Java 8 was a major release in terms of language and APIs.

- Includes several ideas from functional programming like :

i. lambda
ii. stream API

Cybage®
Delivering Value Scientifically

Evolution of JAVA 8



Cybage®
Delivering Value Scientifically

New JAVA 8!!

Functional Interface

- An interface with exactly one abstract method becomes Functional Interface.
- @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces.

Cybage®
Delivering Value Scientifically

Functional Interface

- Functional Interface can be annotated and its optional.

```
@FunctionalInterface
public interface MyfunctionalInterface{
  someMethod();
};
```

- Its just for the convenience , compiler can tell whether the interface is functional or not.

Cybage®
Delivering Value Scientifically

Valid Examples of Functional Interface

- **public interface Runnable{**
  run();
  };

- **public interface Comparator<T>{**
  int compare(T t1 ,T t2);
  };

LAMDA EXPRESSION

Cybage®
Delivering Value Scientifically

Advantages Lambda Expression

- Enables functional programming
- Readable and more concise code
- Easier to use API
- Enable support for parallel processing

Cybage®
Delivering Value Scientifically

Lambda Expression

- It's an anonymous function
- It comprises of:
  - set of parameters,
  - a lambda operator (->) and
  - a function body.

(set of parameters) -> function body;

Cybage®
Delivering Value Scientifically

Valid Lambda Expressions

```
n -> n % 2 != 0;
```

```
(char c) -> c == 'y';
```

```
(x, y) -> x + y;
```

```
(int a, int b) -> a * a + b * b;
```

```
() -> 42 () -> { return 3.14 };
```

```
(String s) -> { System.out.println(s); };
```

```
() -> { System.out.println("Hello World!"); };
```

Functional Interface

- An interface with exactly one abstract method becomes Functional Interface.
- @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces.

- A new package **java.util.function** has been added with bunch of functional interfaces to provide target types for lambda expressions and method references.

Four categories of Functional Interface

Inside Java.util.function we have 4 categories of interfaces:

1. Supplier

```
@FunctionalInterface
public interface Supplier<T>{
  T get();
};
```

It takes an object and provide a new object.

Categories of Functional Interface

2. Consumer/BiConsumer

```
@FunctionalInterface
    public interface Consumer<T>{
      void accept(T t);
     };
```

Its reverse of Supplier, it accepts an object but doesn't return anything.

```
@FunctionalInterface
    public interface BiConsumer< T , U >{
      void accept( T t , U u );
     };
```

This also accepts the object of different type.

Categories of Functional Interface

3. Predicate/BiPredicate

```
@FunctionalInterface
    public interface Predicate< T >{
      boolean test( T t );
     };
```

This method accept an Object and returns a boolean.

```
@FunctionalInterface
    public interface BiPredicate< T t , U u>{
      boolean test( T t , U u);
     };
```

This method accepts two objects of different type and returns a boolean.

Categories of Functional Interface

4. Function/BiFunction

```
@FunctionalInterface
      public interface Function < T , R >{
        R apply( T t );
       };
```

Represents a function that accept one arguments (T) and produces a result (R)

```
@FunctionalInterface
      public interface BiFunction < T , U , R >{
        R apply( T t ,U u );
       };
```

Represents a function that accepts two arguments (T , U) and produces a result (R)

Cybage® Delivering Value Scientifically

Default Methods in Interface

- Java 8 introduces “Default Method” or (Defender methods) new feature, which allows Interface to define implementation to methods which will be used as default in the situation where a concrete Class fails to provide an implementation for that method.

```
public interface OldInterface {
    public void existingMethod();

    default public void newDefaultMethod() {
        System.out.println("New default
        method"
                + " is added in interface");
    }
}
```

Stream API – Java 8 Feature

- The Stream API is used to process collections of objects.
- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined.
- **Following are the different operations On Streams-**

Intermediate Operations:

map
filter
sorted

Terminal Operations:

collect
forEach
reduce

Stream API – Java 8 Feature

map :

```
List number = Arrays.asList(2,3,4,5);

List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

filter :

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

sorted:

```
List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());
```

Stream API – Java 8 Feature

collect:

```
List number = Arrays.asList(2,3,4,5,3);

Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

forEach:

```
List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

reduce:

```
List number = Arrays.asList(2,3,4,5);

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

DEMO

Cybage®
Delivering Value Scientifically

Terminal operations

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach		Consumes data

Terminal Vs Intermediate Call

Terminal vs Intermediate Call

A terminal operation must be called to trigger the processing of a Stream

No terminal operation = no data is ever processed

Date and Time API

- *LocalDate*
- *LocalTime* and
- *LocalDateTime*
- Duration
- Period
- *ZonedDateTime*

Cybage® Delivering Value Scientifically

Strings in Java 8

- **StringJoiner :** used to construct a sequence of characters separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix.

- **Streaming a String : chars()** creates a stream for all characters of the string

eg -> "hey duke".chars().forEach(c -> System.out.println((char)c));

Any Questions?

Thank you!