



# Java 8 New Features

---

A few of the more significant changes and additions.

James Brucker

# Interfaces

## □ Default methods

- instance methods can have method code!
- use the qualifier default
- class "inherits" the default implementation

## □ Static methods with code

Interfaces can have static methods, which must have code.

# Default Method in Interfaces

Add a `getCurrency` method to `Valuable` with default implementation.

```
public interface Valuable {  
    /** value of this item. This is abstract. */  
    public double getValue();  
  
    /** Get the currency. Default is "Baht". */  
    default public String getCurrency() {  
        return "Baht";  
    }  
  
    // ILLEGAL: Cannot override an existing method  
    // default String toString() { return "error!"; }  
}
```

# Static Method in Interface

Add a `getCurrency` method to `Valuable` with default implementation.

```
public interface Valuable {  
    /** value of this item. This is abstract. */  
    public double getValue();  
  
    /** Create money (static). */  
    static Valuable create(double value) {  
        return MoneyFactory.getInstance()  
            .createMoney(value);  
    }  
}
```

# Why add default methods?

Java added **new methods** to **existing interfaces**.

How to do that without **breaking** existing applications?

**Example:** Iterable has a new "forEach" method:

```
<<interface>>  
Iterable<T>  
  
iterator() : Iterator<T>  
  
forEach(c: Consumer<T>)
```

← New method

# Iterable has a forEach() method

The `Iterable` interface has a default `forEach()` method that invokes a `Consumer` object with each element of the `Iterable`.

```
<<interface>>  
Iterable<T>
```

```
iterator(): Iterator<T>  
forEach(c: Consumer<T>)
```



A default method

```
<<interface>>  
Consumer<T>
```

```
accept(arg: T): void
```

# Print a List of Students using loop

The old way:

```
List<Student> classlist =  
    Registrar.getStudents("01219113");  
  
// print each student using for-each loop  
for( Student s: classlist ) {  
    System.out.println(s);  
}
```

# Using forEach with a List

New way: use `forEach` and a `Consumer`

```
List<Student> classlist =  
    Registrar.getStudents("01219113");  
// print each student  
classlist.forEach(  
    // Consumer as Anonymous Class  
    new Consumer<Student>() {  
        public void accept(Student s) {  
            System.out.println(s);  
        }  
    }  
);
```

*Let's make the code shorter using a *Lambda expression*...*



# Using forEach with Lambda

Lambdas (anonymous functions) are another new Java 8 feature

```
List<Student> list =  
    Registrar.getStudents("01219113");  
// print each student  
    /* Consumer as Lambda Expression */  
list.forEach( (s) -> System.out.println(s) );
```

*Let's make this code even shorter using a Method Reference.*

# Method Reference for Lambda

The Lambda just calls `System.out.println(s)` with the same parameter (s), so we can refer to "println" directly.

```
List<Student> classlist =  
    Registrar.getStudents("01219113");  
// print each student  
    /* Consumer as Method Reference */  
classlist.forEach( System.out::println );
```

# Lambda Expressions

Lambdas are "anonymous functions".

A Lambda implements an interface with one abstract method.

```
// Runnable as anonymous class
Runnable mytask = new Runnable() {
    public void run( ) {
        System.out.println("I'm running");
    }
};
```

```
// Runnable as Lambda
Runnable mytask = () -> {
    System.out.println("I'm running");
};
```

# How to Invoke a Lambda?

Exactly the same as object defined using anonymous class.

```
// Runnable mytask as Lambda
Runnable mytask = () ->
    System.out.println("I'm running");

// Invoke the method:
mytask.run();
```

# Lambda Shortcuts

If the method has only **one statement**, you may omit { }.

```
// Runnable as Lambda
```

```
Runnable mytask = () -> {  
    System.out.println("I'm running");  
};
```

```
// Lambda shortcut notation
```

```
Runnable mytask = () ->  
    System.out.println("I'm running");
```

# Swing ActionListener

An ActionListener that reads a text field.

```
private JTextField inputField;

ActionListener inputListener =
    new ActionListener( ) {
        public void actionPerformed(ActionEvent e) {
            String input = inputField.getText();
            inputField.setText("");
            processInput( input );
        }
    };

inputField.addActionListener(inputListener);
```

# ActionListener using Lambda

Java can infer the parameter type (ActionEvent)

```
private JTextField inputField;

ActionListener inputListener =
    (event) -> {
        String input = inputField.getText();
        inputField.setText("");
        processInput( input );
    };

inputField.addActionListener(inputListener);
```

# Lambda that returns a value

Compare strings by length

```
Comparator<String> compByLength =  
    // anonymous class  
    new Comparator<String>() {  
        public int compare(String a, String b) {  
            return  
                Integer.compare(a.length(), b.length());  
        }  
    };  
};
```

Integer.compare(int a, int b) is same as:

```
Integer.valueOf(a).compareTo( Integer.valueOf(b) );
```



# Lambda that returns a value

Compare strings by length

```
Comparator<String> compByLength =  
    // Lambda express returning a value  
    (a,b) -> Integer.compare(a.length(), b.length());
```

Java can infer from "Comparator<String>" that a and b are type `String`.

So, we don't have to write it.

# Streams

---

All collections now support use of *streams*.  
New classes and interfaces are in package:

**`java.util.stream`**

Several new "functional interfaces" that are needed  
for streams:

**`java.util.function`**

# Simple Stream Examples

Create a stream from a list:

```
List<String> fruit = Arrays.asList("grape", "orange",  
    "banana", "apple", "durian", "pear", "guava");  
  
fruit.stream()  
    // creates a stream of elements in fruit  
    // what can you do with a Stream?
```

A stream is like a pipeline.

Each element of the stream is passed from one stream processor to the next.

# Stream methods

void forEach( <b>Consumer</b> )	<i>Consumes</i> each element of the stream.
Stream filter( <b>Predicate</b> )	filter elements in the stream using a boolean test, called a <i>Predicate</i>
Stream sorted( ) Stream sorted( <b>Comparator</b> )	sort elements in the Stream. This requires significant memory.
Stream<R> map( <b>Function</b> <T,R> mapper )	Apply a function to map elements from one type (T) to another (R).
T[ ] toArray( )	return elements as an array

# Filter the fruit

Write a test that returns true if string contains "a".

```
Predicate<String> hasLetterA =  
    new Predicate<String>() {  
        public boolean test(String s) {  
            return s.contains("a");  
        }  
    };
```

<<interface>>  
Predicate<T>

test(arg: T): boolean

# Filter the fruit

Write a Predicate that returns true if string contains "a".

```
Predicate<String> hasLetterA = s -> s.contains("a");  
fruit.stream().filter(hasLetterA).forEach( print );
```

```
<<interface>>  
Predicate<T>
```

```
test(arg: T) : boolean
```

# References

---

The Java Tutorial:

<https://docs.oracle.com/javase/tutorial>

Java 8 Features with Examples

<http://www.journaldev.com/2389/java-8-features-with-examples>

Has short examples of several features