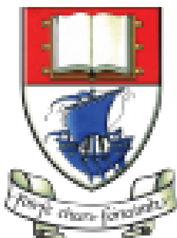
Java language evolution (JDK 7 – 9)

Produced by:

Dr. Siobhán Drohan (sdrohan@wit.ie)

Eamonn de Leastar (edeleastar@wit.ie)



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

2011

JAVA SE 7

Major Changes

Codename Dolphin; added small language changes including strings in switch. The JVM was extended with support for dynamic languages.

2014

JAVA SE 8

Major Changes

Language level support for lambda expressions and default methods and a new date and time API inspired by Joda Time.

2017

JDK 9

Major Changes

Project Jigsaw; designing and implementing a standard module system for the Java SE platform, and to apply that system to the platform itself and the the JDK.

Java™ 7

Java 7 – an outline of some changes

- Can now switch on Strings.
- Inclusion of try-with-resources.
- Multi-catch.
- Improved type inference.
- More new I/O APIs for the Java platform.

RECAP - switch control statement

Pre Java 7: can switch on int and char.
Post Java 7: can also switch on String

```
switch(expression) {
    case value: statements;
                break;
    case value: statements;
                break;
    further cases possible
    default: statements;
             break;
}
```

```
switch(dow.toLowerCase()) {
    case "mon":
    case "tue":
    case "wed":
    case "thu":
    case "fri":
        goToWork();
        break;
    case "sat":
    case "sun":
        stayInBed();
        break;
}
```

RECAP - try-with-resources

- Introduced in Java 7.
- It is a try statement that declares one or more resources.
    - A *resource* is an object that must be closed after the program is finished with it.
- The try-with-resources statement ensures that each resource is closed at the end of the statement.

RECAP - try-with-resources

```
static String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    //try with a finally block, pre Java 7.
    try {
        return br.readLine();
    }
    finally {
        if (br != null)
            br.close();
    }
}
```

```
static String readFirstLineFromFile(String path) throws IOException {
    //try-with-resources, Java 7.  br will be closed regardless of
    //whether the try statement completes normally or abruptly
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

RECAP - try-with-resources

- A try-with-resources statement can have catch and finally blocks just like an ordinary try statement.
- In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.
- All classes implementing the java.lang.AutoCloseable interface can be used inside the try-with-resources construct.

RECAP - Multiple Exception Handling

- In Java 7 and later, you can catch more than one type of exception with one exception handler i.e.
  - A single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

RECAP - Multiple Exception Handling

- In Java 7 and later, you can catch more than one type of exception with one exception handler i.e.
  - A single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|).

RECAP - Type Inference

Since Java 7, type inference applies to collections (<>) i.e.:

```
Map<String, String> myMap = new HashMap<>();
```

<> is required.

```
Map<String, String> myMap = new HashMap();
myMap.put("1", "Or
```

Type safety: The expression of type HashMap needs unchecked conversion to conform to Map<String,String>

4 quick fixes available:

- Add type arguments to 'HashMap'
  - Fix 3 problems of same category in file
- Infer Generic Type Arguments...
- Add @SuppressWarnings 'unchecked' to 'myMap'
- Add @SuppressWarnings 'unchecked' to 'main()'

Press 'F2' for focus

More new I/O APIs for the Java platform.

- Most important package:
    - **java.nio.file** which contains many practical file utilities, new file I/O related classes and interfaces.
- We will briefly look at:
    - **java.nio.file.Path** (interface)
    - **java.nio.file.Files** (class)

https://jaxenter.com/java-7-the-top-8-features-103625.html

java.nio.file.Path (interface)

A Java Path instance represents a *path* in the file system. A path can point to either a file or a directory. A path can be absolute or relative. Basically, this interface can be used in place of the java.io.File class.

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathExample {
	public static void main(String[] args) {

		Path path = Paths.get("c:\\data\\myfile.txt");
	}
}
```

http://tutorials.jenkov.com/java-nio/path.html

java.nio.file.Files (class)

This class consists exclusively of static methods that operate on files, directories, or other types of files.

It contains over 50 utility methods for File related operations which many developers would have wanted to be a part of earlier Java releases e.g.:

- **copy()** – copy a file, with options e.g. REPLACE_EXISTING.
- **move()** – move or rename a file to a target file.
- **newInputStream()** – Opens a file, returning an input stream to read from the file.
- **readAllBytes()** – Reads all the bytes from a file.

https://jaxenter.com/java-7-the-top-8-features-103625.html

Java™ 8

Java 8 - an outline of some changes

- Interfaces – default and static methods
- Lambdas
- Stream collection types (and new method reference, ::)
- Date/time improvements
- Optionals

RECAP: What is an interface?

A type in Java. Similar(ish) to a class

Can contain

- abstract method signatures
- constants (final static fields)
- default & static methods and their bodies (java 8+)
- Private methods and their bodies (java 9+)

Cannot contain

- Any fields other than constants
- Any constructors
- Any concrete methods except default and static(Java 8) and private (Java 9)

Java 8

Defining Interfaces (JDK 7)
Only abstract methods

IAddressBook.java

```
public interface IAddressBook
{
  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();
}
```

Methods are implicitly public and abstract

Defining Interfaces (JDK 8)

Can include default methods

Java 8 introduced **default methods** as a way to extend Interfaces in a backward compatible way.

They can be overridden in implementation classes.

```
public interface IAddressBook
{
  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();

 default String typeOfEntity(){
      return "Address book";
  }
}
```

IAddressBook.java

Defining Interfaces (JDK 8)
Can include static methods

Java 8 allows **static methods** as a way to organise utility methods in a convenient location.

They cannot be overridden in implementation classes.

```
public interface IAddressBook{
  static final int CAPACITY= 1000;

  void clear();
  IContact getContact(String lastName);
  void addContact(IContact contact);
  int numberOfContacts();
  void removeContact(String lastName);
  String listContacts();

  default String typeOfEntity(){
       return “Address book”;
  }
  static int getCapacity(){
        return CAPACITY;
  }
}
```

IAddressBook.java

Lambdas – new in JDK 8

- A Java lambda expression:
  - Is Java's first step into functional programming.
  - Is a **function** which can be created *without belonging to any class*.
  - Can be passed around as if it was an object and executed on demand.

http://tutorials.jenkov.com/java/lambda-expressions.html

Lambdas - Anonymous Inner Classes

- In Java, anonymous inner classes provide a way to implement classes that may occur only once in an application.
- Rather than writing a separate event-handling class for each event, you can write something like this.

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent e){
            System.out.println("Click Detected by Anon Class");
  }
});
```

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

Lambdas - Functional Interfaces

The code that defines the ActionListener is a **functional interface** i.e. one abstract method.

```
package java.awt.event;
import java.util.EventListener;

public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);

}
```

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent e){
            System.out.println("Click Detected by Anon Class");
  }
});
```

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

Lambdas - Syntax

Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

Expression takes two integer arguments, named x and y, and uses the expression form to return x+y.

Can be *either* a single expression or a statement block.

Body is **evaluated** and **returned**.

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

Lambdas - Example

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent e){
            System.out.println("Click Detected by Anon Class");
  }
});
```

```
testButton.addActionListener(e -> System.out.println("Click Detected
                                                     by Lambda Listner"));
```

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html

Stream

A stream

- represents a sequence of objects from an input source, which supports aggregate operations e.g.
  - filter, reduce, find, match, etc..
- takes collections, arrays and I/O sources as input.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – for each

- Stream has provided a new method 'forEach' to iterate each element of the stream.

- The following code segment shows how to print 10 random numbers using forEach over an IntStream.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

ints() returns an unlimited IntStream of random int values.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – for each

- Stream has provided a new method ‘forEach’ to iterate each element of the stream.

- The following code segment shows how to print 10 random numbers using forEach over an IntStream.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

limit(10) returns an IntStream with 10 entries.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – for each

- Stream has provided a new method 'forEach' to iterate each element of the stream.

- The following code segment shows how to print 10 random numbers using forEach over an IntStream.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

forEach() performs an action for each element in the IntStream

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – for each

- Stream has provided a new method 'forEach' to iterate each element of the stream.

- The following code segment shows how to print 10 random numbers using forEach over an IntStream.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

Method reference (::) here refers to the static method println within the containing class. More information here: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – map

- The ‘map’ method is used to map each element to its corresponding result.

- The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

stream() returns a sequential Stream of the numbers collection.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – map

- The 'map' method is used to map each element to its corresponding result.

- The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

map() returns a Stream consisting of the results of applying the given function to the elements of the numbers collection.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – map

- The 'map' method is used to map each element to its corresponding result.

- The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

map() returns a Stream consisting of distinct elements in the Stream *(uses Objects.equals(Object))*.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – map

- The ‘map’ method is used to map each element to its corresponding result.
- The following code segment prints unique squares of numbers using map.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList =
        numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

collect() returns a mutable list of the elements in the Stream.

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – filter

- The ‘filter’ method is used to eliminate elements based on a criteria.

- The following code segment prints a count of empty strings using filter.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

stream() returns a sequential Stream of the strings collection

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – filter

- The ‘filter’ method is used to eliminate elements based on a criteria.

- The following code segment prints a count of empty strings using filter.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

filter() returns a Stream consisting of the elements that match the predicate (i.e. are empty).

https://www.tutorialspoint.com/java8/java8_streams.htm

Stream – filter

- The ‘filter’ method is used to eliminate elements based on a criteria.

- The following code segment prints a count of empty strings using filter.

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

count() returns an int representing the number of elements in the Stream.

https://www.tutorialspoint.com/java8/java8_streams.htm

Date/time improvements

Old Date/Time API (**java.util.Date**):

- **Not thread safe** − java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.

- **Poor design** − Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.

- **Difficult time zone handling** − Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

Date/time improvements

New Date/Time API (**java.time**):

- **Local** − Simplified date-time API with no complexity of timezone handling.
- **Zoned** − Specialized date-time API to deal with various timezones.
- **Joda** – based on the Joda component's approach.

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

Date/time improvements (Local)

```
// Get the current date and time
LocalDateTime currentTime = LocalDateTime.now();
System.out.println("Current DateTime: " + currentTime);

LocalDate date1 = currentTime.toLocalDate();
System.out.println("date1: " + date1);

Month month = currentTime.getMonth();
int day = currentTime.getDayOfMonth();
int seconds = currentTime.getSecond();
System.out.println("Month: " + month +", day: " + day +", seconds: " + seconds);
```

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;
```

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

Date/time improvements (Local)

```
// Get the current date and time
LocalDateTime currentTime = LocalDateTime.now();
System.out.println("Current DateTime: " + currentTime);

LocalDate date1 = currentTime.toLocalDate();
System.out.println("date1: " + date1);

Month month = currentTime.getMonth();
int day = currentTime.getDayOfMonth();
int seconds = currentTime.getSecond();
System.out.println("Month: " + month +", day: " + day +", seconds: " + seconds);
```

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;
```

Console Output

```
Current DateTime: 2017-10-16T19:53:55.053
date1: 2017-10-16
Month: OCTOBER, day: 16, seconds: 55
```

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

Date/time improvements (Zoned)

```
// Get the current date and time
ZonedDateTime date1 =
    ZonedDateTime.parse("2017-10-03T10:15:30+05:30[Asia/Karachi]");
System.out.println("date1: " + date1);

ZoneId id = ZoneId.of("Europe/Paris");
System.out.println("ZoneId: " + id);

ZoneId currentZone = ZoneId.systemDefault();
System.out.println("CurrentZone: " + currentZone);
```

```
import java.time.ZonedDateTime;
import java.time.ZoneId;
```

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

Date/time improvements (Zoned)

```
// Get the current date and time
ZonedDateTime date1 =
    ZonedDateTime.parse("2017-10-03T10:15:30+05:30[Asia/Karachi]");
System.out.println("date1: " + date1);

ZoneId id = ZoneId.of("Europe/Paris");
System.out.println("ZoneId: " + id);

ZoneId currentZone = ZoneId.systemDefault();
System.out.println("CurrentZone: " + currentZone);
```

```
import java.time.ZonedDateTime;
import java.time.ZoneId;
```

Console Output

```
date1: 2017-10-03T10:15:30+05:00[Asia/Karachi]
ZoneId: Europe/Paris
CurrentZone: Etc/UTC
```

https://www.tutorialspoint.com/java8/java8_datetime_api.htm

Optionals: **java.util.Optional<T>**

Optional:

- Is similar to what **Optional** is in Guava.
- is a container object which is used to contain not-null objects.
- object is used to represent null with absent value.
- has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values.

https://www.tutorialspoint.com/java8/java8_optional_class.htm

```java
import java.util.Optional;

public class Java8Tester {
   public static void main(String args[]){

      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      //Optional.ofNullable - allows passed parameter to be null.
      Optional<Integer> a = Optional.ofNullable(value1);

      //Optional.of - throws NullPointerException if passed parameter is null
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }

   public Integer sum(Optional<Integer> a, Optional<Integer> b){

      //Optional.isPresent - checks the value is present or not
      System.out.println("First parameter is present: " + a.isPresent());
      System.out.println("Second parameter is present: " + b.isPresent());

      //Optional.orElse - returns the value if present otherwise returns
      //the default value passed.
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - gets the value, value should be present
      Integer value2 = b.get();
      return value1 + value2;
   }
}
```

https://www.tutorialspoint.com/java8/java8_optional_class.htm

```java
import java.util.Optional;

public class Java8Tester {
   public static void main(String args[]){

      Java8Tester java8Tester = new Java8Tester();
      Integer value1 = null;
      Integer value2 = new Integer(10);

      //Optional.ofNullable - allows passed parameter to be null.
      Optional<Integer> a = Optional.ofNullable(value1);

      //Optional.of - throws NullPointerException if passed parameter is null
      Optional<Integer> b = Optional.of(value2);
      System.out.println(java8Tester.sum(a,b));
   }

   public Integer sum(Optional<Integer> a, Optional<Integer> b){

      //Optional.isPresent - checks the value is present or not
      System.out.println("First parameter is present: " + a.isPresent());
      System.out.println("Second parameter is present: " + b.isPresent());

      //Optional.orElse - returns the value if present otherwise returns
      //the default value passed.
      Integer value1 = a.orElse(new Integer(0));

      //Optional.get - gets the value, value should be present
      Integer value2 = b.get();
      return value1 + value2;
   }
}
```

Console Output

```
First parameter is present: false
Second parameter is present: true
10
```

https://www.tutorialspoint.com/java8/java8_optional_class.htm

Java™ 9

Java 9 - an outline of some changes

- Interfaces – private methods
- Collection factory methods
- Try with resources improvements
- Stream API improvements
- REPL (Shell)
- Module system

Defining Interfaces (JDK 9)

With private methods

Java 9 allows **private methods** as a way to avoid writing duplicate code (i.e. promote re-usability) and also to hide interface implementation.

The methods can be private and private static and are written in the same way you would write a private method in a class.

```
public interface IAddressBook{
  static final int CAPACITY= 1000;

  void clear();
  IContact getContact(String lastName);
  void addContact(IContact contact);
  int numberOfContacts();
  void removeContact(String lastName);
  String listContacts();

  default String typeOfEntity(){
       return "Address book";
  }
  static int getCapacity(){
        return CAPACITY;
  }

  private static void displayDetails(){
     //method implementation in here
  }
}
```

IAddressBook.java

Collection Factory Methods

- Often you want to create a collection (e.g., a List or Set) in your code and directly populate it with some elements…
  - That leads to repetitive code where you instantiate the collection, followed by several `*add*` calls.
- With Java 9, several so-called collection factory methods have been added:

```
Set<Integer> ints        = Set.of(1, 2, 3);
List<String> strings     = List.of("first", "second");
Map<String, String> map = Map.of("foo", "a", "bar", "b", "c");
```

- NOTE: Immutable collections are created (i.e. cannot add to it) and the collection implementation is selected by Java (e.g. ArrayList, LinkedList).

https://www.pluralsight.com/blog/software-development/java-9-new-features

try-with-resources improvement

Java SE 7 example

```
void testARM_Before_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (BufferedReader reader2 = reader1) {
   System.out.println(reader2.readLine());
 }
}
```

Java 9 example

```
void testARM_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (reader1) {
   System.out.println(reader1.readLine());
 }
}
```

Improvements to avoid verbosity and improve readability.

https://www.journaldev.com/13121/java-9-features-with-examples#private-methods

Stream API improvement

- Addition of extra methods…as a sample, we will look at the **takeWhile** method:
  - takes a predicate as an argument and returns a Stream of subset of the given Stream values until that Predicate returns false for first time. If first value does NOT satisfy that Predicate, it just returns an empty Stream.

```
Stream.of(1,2,3,4,5,6,7,8,9,10)
        .takeWhile(i -> i < 5 )
        .forEach(System.out::println);
```

Console Output

```
1
2
3
4
```

https://www.journaldev.com/13121/java-9-features-with-examples#private-methods

REPL (Read Evaluate Print Loop i.e. Shell)

- The jshell is used to easily execute and test Java constructs like class, interface, enum, object, statements etc.

On your command line, start the shell by typing jshell.

Then enter any Java 9 statements you wish.

```
G:\>jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro


jshell> int a = 10
a ==> 10

jshell> System.out.println("a value = " + a )
a value = 10
```

https://dzone.com/articles/jshell-in-five-minutes & https://www.journaldev.com/13121/java-9-features-with-examples#repl

REPL (Read Evaluate Print Loop i.e. Shell)

We can declare a method in a similar way as Flow control, and press for each new line:

```
1  jshell> String helloWorld() {
2   ...> return "hello world";
3   ...> }
4  | created method helloWorld()
```

Then call it:

```
1  jshell> System.out.println(helloWorld());
2  hello world
```

https://dzone.com/articles/jshell-in-five-minutes & https://www.journaldev.com/13121/java-9-features-with-examples#repl

REPL (Read Evaluate Print Loop i.e. Shell)

We can also define classes in JShell:

```
1  jshell> class HelloWorld {
2   ...> public String helloWorldClass() {
3   ...> return "helloWorldClass";
4   ...> }
5   ...> }
6  | created class HelloWorld
```

And assign and access them:

```
1  jshell> HelloWorld hw = new HelloWorld();
2  hw ==> HelloWorld@27a5f880
3  | created variable hw : HelloWorld
4
5  jshell> System.out.println(hw.helloWorldClass());
6  helloWorldClass
```

https://dzone.com/articles/jshell-in-five-minutes & https://www.journaldev.com/13121/java-9-features-with-examples#repl

Module System

- One of the biggest changes in Java 9; it is part of the **Jigsaw** Project.
- With JDK9, you can separate your code into individual modules.
- *"A module is a named, self-describing program component that consists of one or more packages (and data)"* https://jaxenter.com/new-features-in-java-9-137344.html
  - Each module needs a **module-info.java** file. It is placed in the root directory of the module. Within this file, you can declare:
    - which modules your code is dependent upon i.e. jdk modules, external jars, etc.
    - which packages are allowed to see/use your module.

Java™ 10??

It’s coming…in March 2018!

Want to experiment further…

Some good code examples to experiment with can be found on these websites:

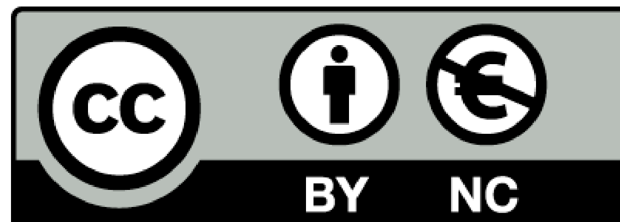
- https://www.tutorialspoint.com/java8/index.htm (Java 8)
- https://www.journaldev.com/12850/java-9-private-methods-interfaces (Java 8&9)
- https://www.journaldev.com/13121/java-9-features-with-examples (Java 7 to 9)

A webinar that might be of interest:

- https://dzone.com/articles/real-world-java-9-webinar

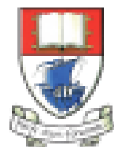
Articles:

- https://dzone.com/guides/java-development-and-evolution
- https://jaxenter.com/new-features-in-java-9-137344.html
- https://www.pluralsight.com/blog/software-development/java-9-new-features
- https://aboullaite.me/wrapping-up-java-9-new-features/
- https://dzone.com/articles/java-9-the-exciting-bits



Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see http://creativecommons.org/licenses/by-nc/3.0/


Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit