Java 8

Kabbaj Mohammed Issam

kabbaj.m@gmail.com

Histoire du Java

- Version JDK
  - 1.0 (1995)
  - 1.1 (1997)
  - 1.2 (1999)
  - 1.3 (2001)
  - 1.4 (2002)
  - 1.5 (2004)
  - 1.6 (2006)
  - 1.7 (2011)
  - 1.8 (2014)
  - 1.9 (2017)
  - 1.10 (2018)
  - 1.11 (2018)
  - 1.12 (2019)
  - 1.13 (2019)

JAVA 7

- Langage :
  - Numeric literals : **long amount = 1_234_567L;**
  - Diamond operator : **List<String> list = new ArrayList<>();**
  - String-in-switch : **case "Monday": …**
  - Try-with-resources : **try (InputStream is = …) { … }**
  - Multi catch : **catch (IOException | MyException e) { … }**
- API :
  - Mises à jour de JAXP, JAXB, JAX-WS
  - New IO 2 : manipulations de fichiers et répertoires, WatchService, etc.
  - Fork/join : parallélisation de tâches
  - InvokeDynamic : appels de langages tiers
  - Support Unicode 6
  - Améliorations Swing : Nimbus L&F, transparence, etc.
  - JDBC 4.1 qui supporte try-with-resources

JAVA 8

- Langage :
  - Méthodes par défaut et statiques dans les interfaces
  - Références de méthode
  - Expressions lambda
  - Annotations de type
- API
  - Date and time API
  - Interfaces fonctionnelles
  - Stream API
  - Améliorations dans la gestion de concurrence
  - Nashorn , un nouveau moteur JavaScript
  - Unicode 6.2, JDBC 4.2, JAXP 1.6, etc.
  - Base 64 encoder et decoder

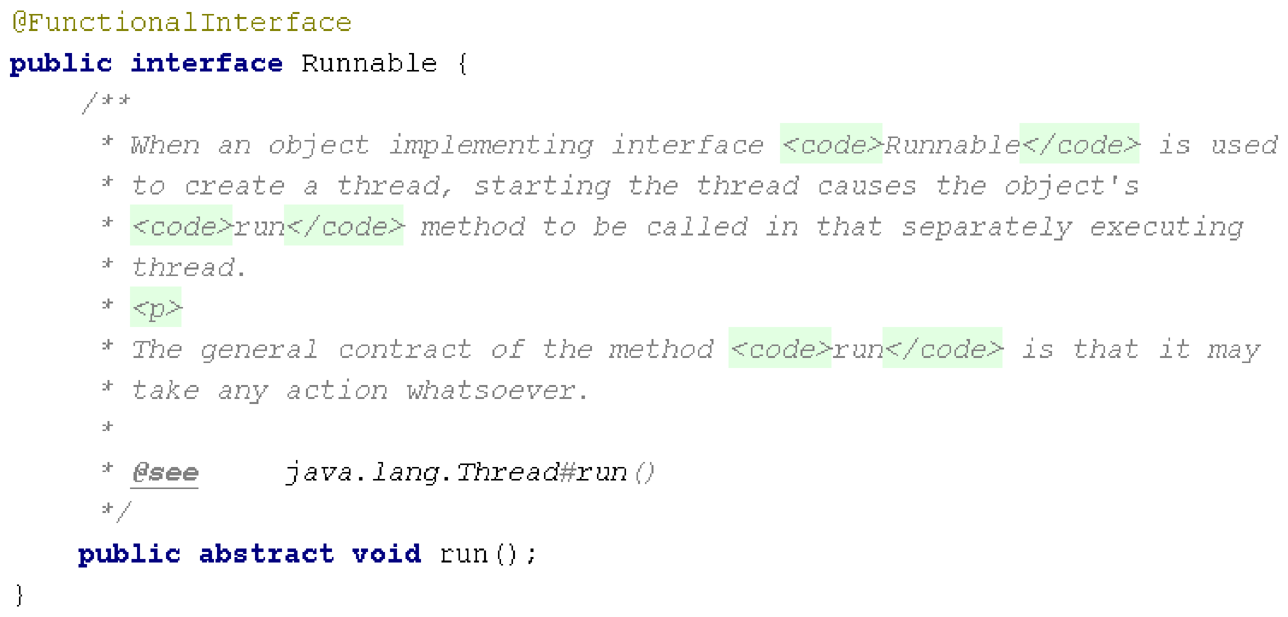
Interfaces

- Méthodes par défaut et méthodes statiques

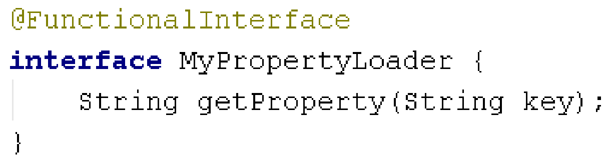
```java
public interface User {

    String getFirstName();

    String getLastName();

    // Default method also known as defender method
    default String getFullName() {
        return getFirstName() + " " + getLastName();
    }

    // Static method
    static boolean validateName(String name) {
        return name != null;
    }
}
```

Interfaces fonctionnelles

- Annotée @FunctionalInterface et avec une seule méthode abstraite



```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```



```
@FunctionalInterface
interface MyPropertyLoader {
    String getProperty(String key);
}
```

Interfaces fonctionnelles

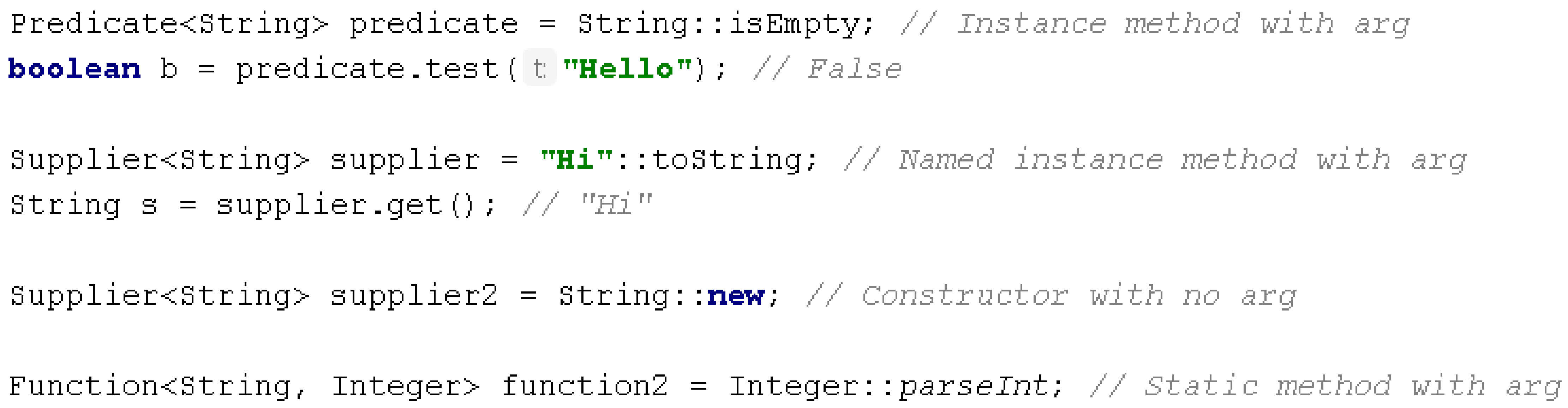
- Exemples de fonctions dans java.util.function :

Interface	Entrées	Sortie
Consumer<T>	T	
Supplier<T>		T
Predicate<T>	T	boolean
Function<T, R>	T	R
IntFunction<R>	int	R
BiFunction<T, U, R>	T, U	R

Interfaces fonctionnelles

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Références de méthode



```java
Predicate<String> predicate = String::isEmpty; // Instance method with arg
boolean b = predicate.test( t: "Hello"); // False

Supplier<String> supplier = "Hi"::toString; // Named instance method with arg
String s = supplier.get(); // "Hi"

Supplier<String> supplier2 = String::new; // Constructor with no arg

Function<String, Integer> function2 = Integer::parseInt; // Static method with arg
```

Références de méthode

- Cas d’usage :

```
class User {
    private String name;
    private String email;
```

```
class SecurityService {
    public String getCurrentUserName() {
        User user = getCurrentUserFromSessionOrWhatever();
        if (user != null) {
            return user.getName(); // The only interesting part
        }
        else {
            return null;
        }
    }

    public String getCurrentUserEmail() {
        return getAttribute(User::getEmail); // Much better!
    }

    private <T> T getAttribute(Function<User, T> f) {
        User user = getCurrentUserFromSessionOrWhatever();
        if (user != null) {
            return f.apply(user);
        }
        else {
            return null;
        }
```

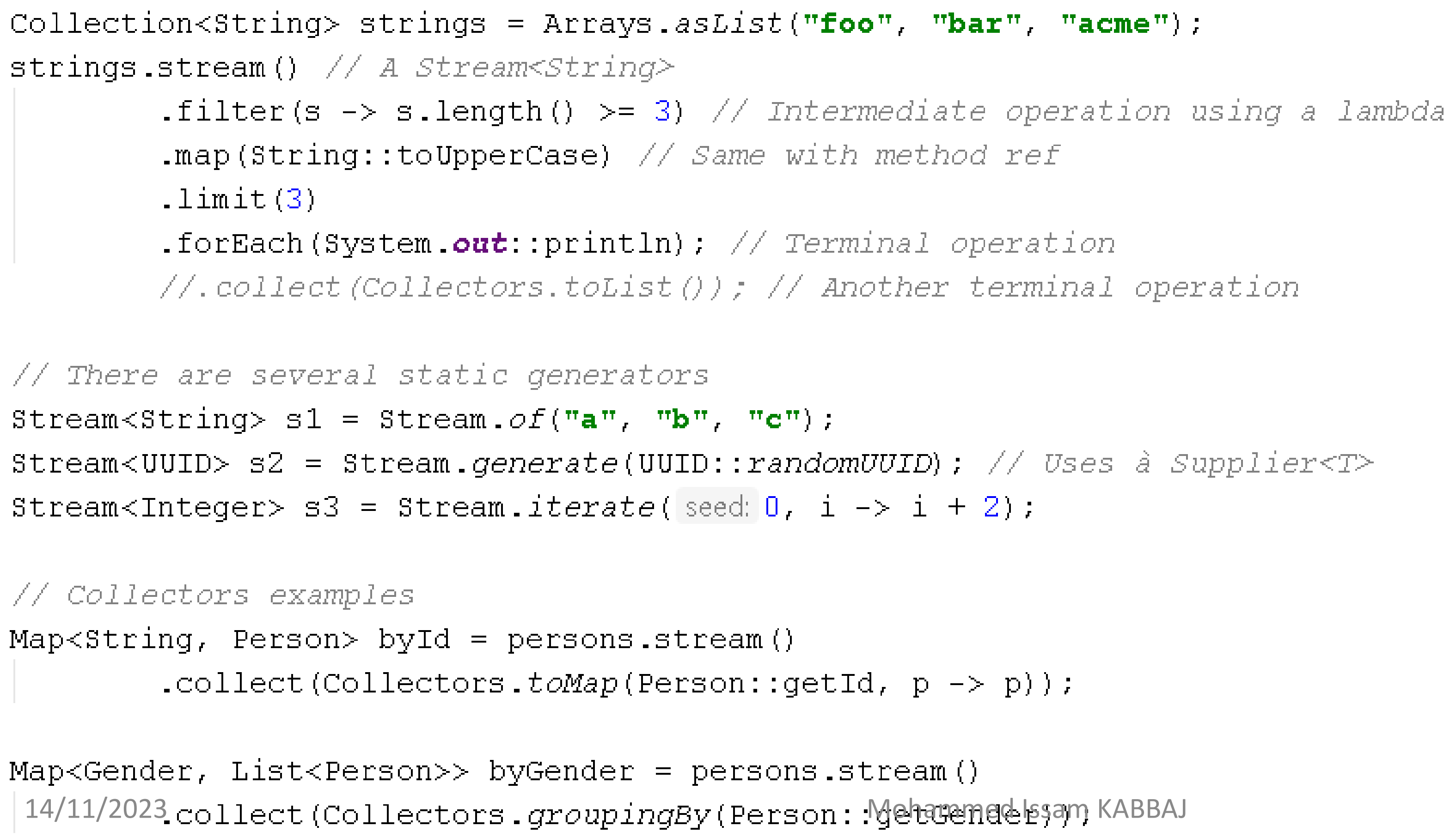
Expressions lambda

- Format : ([type1] arg1, [type2] arg2...) -> { body }

```
// Log a string
Consumer<String> logger = (String a) -> System.out.println(a);

// Check if a user exists and is valid
Predicate<String> validateUser = id -> {
    User user = loadUser(id);
    return user != null ? user.isValid() : false;
};

// Return the current date
Supplier<Date> now = () -> new Date(); // Date::new is fine too

// Return the difference between two integers
BiFunction<Integer, Integer, Integer> diff = (a, b) -> a - b;
```

Streams

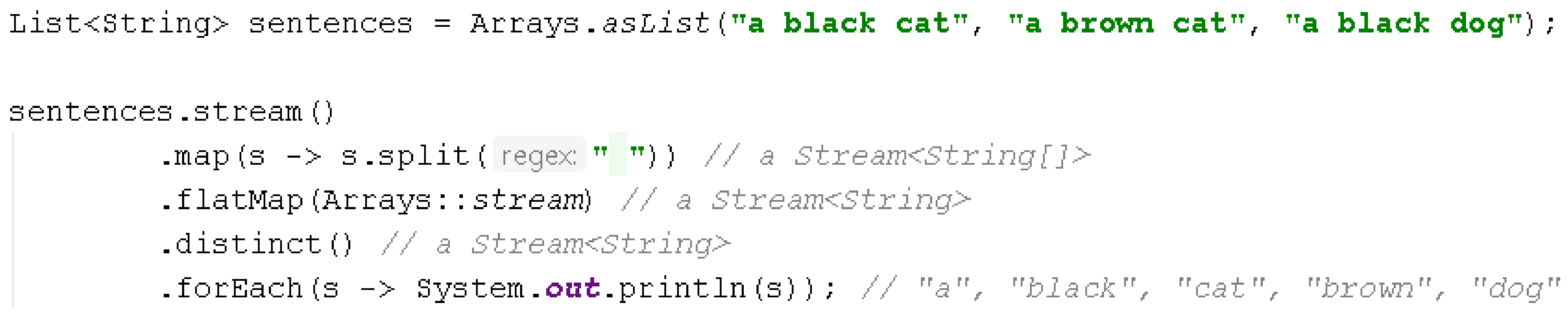
- Pour traiter des séquences d’éléments



```
Collection<String> strings = Arrays.asList("foo", "bar", "acme");
strings.stream() // A Stream<String>
        .filter(s -> s.length() >= 3) // Intermediate operation using a lambda
        .map(String::toUpperCase) // Same with method ref
        .limit(3)
        .forEach(System.out::println); // Terminal operation
        //.collect(Collectors.toList()); // Another terminal operation

// There are several static generators
Stream<String> s1 = Stream.of("a", "b", "c");
Stream<UUID> s2 = Stream.generate(UUID::randomUUID); // Uses à Supplier<T>
Stream<Integer> s3 = Stream.iterate( seed: 0, i -> i + 2);

// Collectors examples
Map<String, Person> byId = persons.stream()
        .collect(Collectors.toMap(Person::getId, p -> p));

Map<Gender, List<Person>> byGender = persons.stream()
        .collect(Collectors.groupingBy(Person::getGender));
```

flatMap

- Pour aplatir des collections :
  - { { « A », « B »}, { }, {« C »} } => { « A », « B », « C » }



```
List<String> sentences = Arrays.asList("a black cat", "a brown cat", "a black dog");

sentences.stream()
        .map(s -> s.split( regex: " ")) // a Stream<String[]>
        .flatMap(Arrays::stream) // a Stream<String>
        .distinct() // a Stream<String>
        .forEach(s -> System.out.println(s)); // "a", "black", "cat", "brown", "dog"
```

Optional

- API fonctionnelle pour gérer null

```java
class Parent {
    List<Child> children;

class Child {
    int age;
```

```java
public int getOldestSonAge(Parent parent) {
    int age = 0;
    if (parent != null) {
        List<Child> children = parent.getChildren();
        if (children != null) {
            for (Child child : children) {
                if (child.getAge() > age) {
                    age = child.getAge();
                }
            }
        }
    }
    return age;
}
```

```java
public int getOldestSonAgeWithOptional(Parent parent) {
    return Optional.ofNullable(parent) // an Optional<Parent>
            .map(Parent::getChildren) // an Optional<List<Child>>
            .orElseGet(ArrayList::new) //.orElse(new ArrayList())
            .stream()
            .map(Child::getAge)
            .max(Integer::compareTo) // an Optional<Integer>
            .orElse( other: 0);
}
```

Optional

- Itérer sur des optional

```
List<Optional<String>> strings = Arrays.asList(Optional.empty(),
        Optional.of("foo"), Optional.of("bar"));

// Get the first non empty element
String s1 = strings.stream()
        .filter(Optional::isPresent)
        .map(Optional::get)
        .findFirst().get();

// Alternative
String s2 = strings.stream()
        .flatMap(o -> o.isPresent() ? Stream.of(o.get()) : Stream.empty())
        .findFirst().get();

// Alternative
String s3 = strings.stream()
        .flatMap(o -> o.map(Stream::of).orElseGet(Stream::empty))
        .findFirst().get();

// Java 9 alternative
//String s4 = strings.stream().flatMap(Optional::stream).findFirst().get();
```

Date and time API

```java
// Date
LocalDate currentDate = LocalDate.now();
int dayOfMonth = currentDate.getDayOfMonth();
LocalDate firstAug2014 = LocalDate.of( year: 2014,  month: 8,  dayOfMonth: 1);

// Time
LocalTime afterMidday = LocalTime.of( hour: 13,  minute: 30,  second: 15); // 13:30:15
LocalTime nowInLosAngeles = LocalTime.now(ZoneId.of("America/Los_Angeles"));

// Date and time
LocalDateTime secondOc2014 = LocalDateTime.of( year: 2014,  month: 10,  dayOfMonth: 2,  hour: 12,  minute: 30);

// Other classes: Instant, Period, Duration, etc.

// Operations
boolean isAfter = currentDate.isAfter(firstAug2014); // true
LocalDate tomorrow = currentDate.plusDays(1);
LocalDate lastDayOfMonth = currentDate.with(TemporalAdjusters.lastDayOfMonth());
Period period = Period.between(currentDate, tomorrow);
LocalDate fromIsoDate = LocalDate.parse("2014-01-20");
String formatted = currentDate.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"));
```