Modern Java

Improvements and new features

Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

Agenda

- Introduction
- Lambdas
  - Why?
  - How?
  - Functional interfaces
- Streams
  - Creating
  - Tranformations
  - Reductions
  - Parallel streams
- Date and Time
- Later versions of Java

Introduction

The evolution of Java

- Java was first introduced in 1995 with version 1.0.
- Sun Microsystems lead the development from version 1.0 to 6.0 (2006).
- During Oracle's acquisition of Sun in 2009-2010 there was a rather long pause in development.
- Oracle released version 7 in 2011 with only minor additions.
  - Most notably a new file I/O and JVM support for dynamic languages.
- The start of "modern Java" can be seen with Java 8 in 2014.
  - Most visible was the inclusion of *lambda expressions*, discussed more later.

Java 9 and forward

- Java 9 was released in 2017 after a lengthy debate on how modules were to be be implemented.
- The module concept is the most visible *feature* of Java 9, however...
- The most talked about change was in the future release schedule.
- From Java 9, Java will be released on a six month basis.
  - Partly due to *one* feature holding the release of Java 9.
- This means that Java 10 was released in March 2018, Java 11 (LTS) in September 2018 and Java 12 in March 2019
- Java 13 (current) was released in September 2019 and Java 14 is due in March 2020.

Oracle JDK and OpenJDK

- Oracle is today working towards there being no technical difference between Oracle's JDK and OpenJDK.
  - The difference being that support by Oracle is given to the Oracle JDK.
  - It is also possible to get long term support by Oracle for paying customers.
- Support for public Java is now only six months, however many other companies package OpenJDK with both features and support.
  - For example Red Hat and AdoptOpenJDK.
- There are still some differences between the commercial JDK and OpenJDK but Oracle is opening up previously closed sourced software to iron out the differences.

New features in Java 8

- A number of new features in Java 8 will be presented next as this was a milestone release.
  - Additions in other versions will be covered later in the lecture.
- The by far largest contribution to Java 8 was the inclusion of *lambda expression*.
  - This makes use of *closure*, which allows for a more functional way of programming.
  - Functional programming is declarative, meaning it says “what” to do rather than “how” to do it.
- Streams give a new way to work with collections.
  - Works by using lambda and a number of functions.
  - Streams are very easy to parallelise and can most often be done automatically.

Lambdas

Lambdas

- Lambda gives functional programming to the object oriented language of Java.
  - Previously, Java supported procedural, object oriented and generic programming.
- Using lambda, the developer can concentrate on *what* to do, not *how*.
  - The “how” is now optimised by the compiler and JVM.
- The key points about lambdas are:
  - A lambda expression is a block of code with parameters.
  - Use lambda when execution of a block of code can be done later.
  - Lambda expressions can be converted into functional interfaces.

Why use lambdas?

- Simply put, a lambda expression is a piece of code that you can pass around.
- This means that execution can be done later and repeatedly.
- The syntax of a lambda expression is similar to that of a method.

```
(parameter1, parameter2...) -> { Block of code }
```

- If no parameters are used, the parenthesis can be left empty.
- If the block of code is just one line, then the curly brackets can be left out.

Example lambdas

- A simple lambda returning a string:

```
() -> "The End!"
```

- A lambda returning the comparison between two strings using `compare`:

```
(str1, str2) -> Integer.compare(str1.length(), str2.length())
```

- If only one parameter is used, then the parenthesis can be left out as well:

```
value -> System.out.println(value+"")
```

Functional interfaces

- The lambda expressions themselves are not doing much, they need to be assigned to something.
- This “something” must be a *functional interface*.
  - It is not even possible to assign it to `Object`!
- A functional interface is a an interface with *a single abstract method* (SAM).
- A SAM acts as the object oriented equivalent of a function in a functional programming language.
- The lambda expression is then supplied as the implementation to that method.
- Variables are then declared *of that type* to store the lambda expression.

Predefined functional interfaces

- In the Java API, a number of generic functional interfaces are supplied in `java.util.function`.
- The simplest of those is `Runnable` which does not take any parameters nor returns any value.
- To create an object of that kind with a simple lambda expression, write:

```
Runnable really = () -> System.out.println("Really!");
```

- The object `really` now holds the expression – but it hasn't executed it!
- To execute call the `run()` method on the object.

```
really.run();
```

Other interfaces

- As stated, a number of functional interfaces are predefined and they all have the same basic structure.
- Some of the interfaces are:

Name	Parameter types	Return type	Abstract method
Runnable	none	void	run
Supplier$< T >$	none	T	get
Consumer$< T >$	T	void	accept
Function$< T, R >$	T	R	apply
BiFunction$< T, U, R >$	T, U	R	apply
Predicate$< T >$	T	boolean	test
BiPredicate$< T, U >$	T, U	boolean	test

- It is also possible to create your own functional interfaces.
  - Tag them with the `@FunctionalInterface` for the get better compiler checks.

Example

```
public static void main(String[] args) {
  BiFunction<String, String, Integer> comp =
    (str1, str2) -> Integer.compare(str1.length(), str2.length());
  int res = comp.apply("Anakin Skywalker", "Darth Vader");

  Consumer<String> valueOutput = value -> System.out.println(value+"");
  valueOutput.accept(res+"");

  Supplier<String> output = () -> "The End!";
  valueOutput.accept(output.get());
  Runnable really = () -> System.out.println("Now, really -- The End!");
  really.run();
}
```

Output:

```
1
The End!
Now, really -- The End!
```

Method references

- If a method already exists that does what is needed, it can be converted to a lambda expression.
- This is done using the new operator `::`
- There are three principal cases:
  - *object::instanceMethod*
  - *Class::staticMethod*
  - *Class::instanceMethod*
- It is also possible to refer to the constructor of a class by using `::new`
- Notice that in all cases we still need a functional interface to supply the value to the method.

Example

- In the previous example we used a `Consumer` object to print a value on screen.

```
Consumer<String> valueOutput =
  value -> System.out.println(value+"");
valueOutput.accept(res+"");
```

- It is possible to do this even cleaner using method references, as only the method `println` is used.

```
Consumer<String> moreOutput = (System.out::println);
moreOutput.accept("Let's end this...");
```

- The output is displayed when the `accept` method is executed.

Longer example

- Just to show that more than “one-liners” can be used, here is a (rather stupid) longer example:

```
int[] array = new int[]{1, 2, 3, 4, 5};

 Runnable change = () -> {
     for(int i = 0; i < array.length; i++)
         if(array[i]%2==0)
             array[i]=array[i]+1;
 };

change.run();
```

- The content of the array is changed to `1 3 3 5 5` after running the expression.

Lambdas and inner classes

- Lambdas can also be used *instead of* inner classes.
  - This was one of the first uses and also in the beginning considered to be the only implementation.
- Inner classes are often used for actions in GUIs as they implement only one interface.
  - This is the case in Swing as well as in JavaFX.
- In the example we implement the `EventHandler` interface for a JavaFX button.
  - JavaFX button's use `setOnAction` to respond to an event.
- In the later JavaFX lectures you will see and understand this better.

Example

- The following code (as done before Java 8):

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

- Can be replaced with the following in Java 8:

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

- The slim notation is possible since the only thing that can be instantiated in this method is an `EventHandler`.

Streams

The Stream API

- To make it easier to use lambdas on collections of values, the *stream* API was introduced.
- This gives a more declarative way of working with values – you say what should be done, not how.
- Another benefit of this is that it is much easier to parallelise the operation.
  - This is simply done by *saying* that you want it to be done in parallel and the implementation works it out.
  - In contrast to earlier attempt of this in Java, the chance that it will be run in parallel is much higher.
- With the Stream API an additional type is introduced, `Optional`, which is a safer `null`.

Streams

- A stream looks superficially like an iterator.
- Streams move elements through a sequence of processing steps – known as a *stream pipeline*
  - The process begins from a data source (for example an array or collection)
  - In the process, various intermediate operations are preformed on the elements, ending with a terminal operation.
  - A stream pipeline is formed by chaining method calls.
- In a stream, no values are stored, they may be stored in an underlying structure.

More on streams

- A stream is never mutated, it always returns a new stream with the new result.
- The intermediate operations are also called *aggregated operations* or *stream operations*.
- They are all defined in the `Stream<T>` interface.
- Working with streams is done in three stages:
  1. Create the stream
  2. Specify intermediate operations for transformation(s)
  3. Apply a terminal operation that produces a result

Initial example

- To calculate the sum of the content in an array something like the following could be done:

```
double[] numbers = {1, 24, 36, 42, 75, 4, 93, 31, 5, 36, 9};

double total = 0;
for (double element : numbers) {
    total += element;
}
```

- The same can be done using the new Java 8 additions as:

```
double total2 = Arrays.stream(numbers).sum();
```

- The first is using what is called *external iteration* which is error prone as it leaves everything to the developer.
- The other is using *internal iteration* which means that the library descides what is the best way to do it.

Next example

- Just to show how it works, here is an example where all the words longer than seven in the bible are counted (55824).

```
String contents = new String(Files.readAllBytes(
    Paths.get("theBible.txt")), StandardCharsets.UTF_8);
List<String> words = Arrays.asList(contents.split("[\\P{L}]+"));

// Using external iterator
int counterIter = 0;
for(String w: words){
    if(w.length() > 7)
        counterIter++;
}
System.out.println("Number of words: " + counterIter);

// Using internal iterator (lambda)
long counterStream = words.stream().filter(w->w.length() > 7).count();
System.out.println("Number of words: " + counterStream);
```

Intermediate operations

- Some of the more commonly used *intermediate* stream operations are:
  - `filter` – returns a stream based on a condition.
  - `distinct` – returns a stream with unique elements.
  - `limit` – returns a stream with a specific number of elements.
  - `map` – maps each element to a new, for example the square of each element.
  - `sorted` – sorts and returns the stream.
- For each stream pipeline zero or more intermediate operations can be applied.

Terminal operations

- Common *terminal* stream operations include:
  - `forEach` processes each element.

    Reduction operations (returns *one* value)
  - `average`, `count`, `max` and `min`.

    Container returning:
  - `collect` and `toArray`

    Search operations:
  - `findFirst`, `findAny`, `anyMatch` and `allMatch`
- Only one terminal operation can be used.
- It always returns the result of the entire action.

Stage 1: Create a stream

- Any collection, array or iterator can be turned into a stream.
- The `Stream` class has a method called `of()` for the conversion.
- It is also, as seen previously, possible to call the `stream()` method for the collection.
- A number of other methods are also available:
  - `empty()` returns an empty stream
  - `generate()` returns an *infinite* stream
  - `iterate()` returns an infinite, but sequential, stream
- There are many others, but most often the stream is created with the three stages, as seen in the example.

More on creating

- In addition to using `stream` on the collection, there are also several classes for common streams.
  - Including `IntStream`, `LongStream` and `DoubleStream`.

```
int[] values = {6, 9, 2, 1, 3, 7, 8, 4};
System.out.println(IntStream.of(values).average().getAsDouble());
```

- These interfaces all inherit from `BaseStream` and can be used in the same way as a normal `Stream`.

Stage 2: Transformations

- A stream transformation reads data from one stream and puts the transformed data in another.
- The previously used `filter()` produces a new stream that matches a certain condition.
  - This can be seen when looking at the signature, it takes a `Predicate<T>` as input (which produces a boolean).
- For changing the content (or really, to produce a new and changed stream) use `map`.
- The input to the method is the function (lambda) to execute for change.

```
Stream<Character> firstChars = words.stream().map(s -> s.charAt(0));
firstChars.forEach(value -> System.out.print(value + ""));
```

- Prints out the first letter in each word.

Another example

- To change words to lower case and count the number of words starting with ’q’ use `map()` and `filter()`.

```
long qWords = words.stream().map(String::toLowerCase)
    .filter(s->s.startsWith("q")).count();
System.out.println("Words starting with q: " + qWords);
```

- Assuming the same text as before, the number or words starting with ’q’ is 300.
  - In a way, it concatenates several streams into one complete.

Other useful transformations

- To limit the size of a stream (remember that generated streams are infinite), use `limit()`.

```
Stream<Double> randomDoubles = Stream.generate(Math::random).limit(100);
```

- The method `skip()` does the opposite, namely to skip the *n* first elements in a stream.
- Real concatenations can be done using the `concat()` method – but only if the stream is finite!
- `distinct()` returns only the unique entries in a stream.

```
long uniqueWords = words.stream().distinct().count();
```

- All things being equal in the text, 13556 is returned as the count.

Stage 3: Result

- As the last part we will look at how to get something from the stream.
- This part is also known as *reduction* as it reduces the stream to something that can be used in the program.
  - This is why they are called terminal operations.
- Several examples have used the `count()` reduction that calculates a number.
- Many of the different reductions return the new type `Optional`.
- The object has a method `isPresent()` that says if a result was obtained or not.
- To reach the the value itself (if present) use the `get()` method.

Example

- The following code searches a list (converted to a stream) for the name 'David Gilmour':

```
List<String> names = asList("Roger Waters", "David Gilmour",
    "Nick Mason", "Richard Wright");
Optional<String> found = names.stream()
    .filter(s -> s.contains("David Gilmour")).findFirst();

if(found.isPresent())
    System.out.println("The name " + found.get() + " was found!");
else
    System.out.println("The name was not found!");
```

- In this case, the name will be found and displayed.

Collecting the result

- The transformations result in a modified stream.
- It is often the case that we need this stream as a collection to continue working on it.
- The easiest way to do this is to use the `collect()` method as our reduction.
- This method can use an `Collector` class to return a collection class.
  - Easy to list or to set using `toList()` or `toSet()` – more are available.
- It is also possible to specify the data type by using the `toCollection()` method.

Example

- The following takes the words of the bible and makes them lower case, finds all starting with 'q' and turns everything to a list.

```
List<String> uniqueWordsList = words
                                .stream()
                                 .map(String::toLowerCase)
                                 .distinct()
                                 .filter(s->s.startsWith("q"))
                                 .collect(Collectors.toList());
```

- The list can then be worked with as normal, for example printing it.

```
for(String w : uniqueWordsList)
    System.out.println(w);
```

- Or using lambda...

```
uniqueWordsList.stream().forEach(System.out::println);
```

Parallel streams

- One of the advantages of streams is that they are easy to parallelise.
- The easiest way to do this is to use `parallelStream()` instead of the normal `stream()`.
- Even though the JVM handles the implementation, it is important that the stream *can* be parallelised.
  - The operations need to be *stateless* and able to run in any order.
- Notice that even if the end result needs an order, it is possible to parallelise.
  - The stream will be partitioned into segments that are reassembled.
- Be certain, though, that the collection used for the stream isn't modified in any way.

Example

- This snippet first counts the words of the Veda in sequence and then in parallel.

```
String newContents= new String(Files.readAllBytes(Paths.get("veda.txt")),
      StandardCharsets.UTF_8);
List<String> newWords = Arrays.asList(contents.split("[\\P{L}]+"));

long startTime1 = System.nanoTime();
long numberOfWords = newWords.stream().count();
long stopTime1 = System.nanoTime();
System.out.println("Number of words in the Veda: " + numberOfWords +
      " in " + (stopTime1 - startTime1));

long startTime2 = System.nanoTime();
long numberOfWordsP = newWords.parallelStream().count();
long stopTime2 = System.nanoTime();
System.out.println("Number of words in the Veda: " + numberOfWords +
      " in " + (stopTime2 - startTime2));
```

- 4,240,954 nanoseconds (0.0042 seconds) versus 2,605,946 nanoseconds (0.0026 seconds).

Much more

- There are many more parts to streams than we can fit here.
- Searching with your favourite search engine will help you to find it.
- Things not covered in this lecture:
  - Several operations like `boxed`.
  - Functions like `identity`
  - More of `Collectors` like `groupingBy`
- The best way to learn is to try it out.

Date and Time API

The new date and time API

- The old API for date and time has needed to be replaced for a long time.
- It had a number of problems.
  - It wasn't thread safe.
  - Years in `Date` start at 1900.
  - Months start at 1 but week days a 0 – not very intuitive.
- Numerous third party libraries have been developed to make time easier.
- Starting with Java 8, a new – standard – way of handling time was introduced.

The time line

- In the Java 8 date and time API a day has exactly 86,400 seconds.
- A point in time is represented by `Instant` which has en *epoch* (or origin) at midnight January 1, 1970.
  - Calculated from Greenwich Royal Observatory in London.
  - Which is the UTC standard.
- `Instant` goes back a billion years for time calculations.
- Time will “end” in Java 8 at the year 1,000,000,000 on December 31.
  - This should be sufficient for most needs...

Time measuring in Java 8

- Previously we saw that `System.nanoTime` could be used to measure time.
- In Java 8 the preferred way is actually using `Instant`:

```
Instant start = Instant.now();
doSomethingMethod();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

- Using `Duration` the time elapsed can be better calculated.
- The time can be recalculated to seconds, minutes, hours and so on.

Arithmetic operations

- To help calculations there are a number of operations predefined for `Instant`.
  - `plus()` and `minus()` to add or subtract `Instant` or `Duration`.
  - `plusNanos()`, `plusMillis()` and so on for adding an amount of time unit.
  - The same for minus...
  - `multiplyedBy()`, `dividedBy()` and `negated()`.
  - `isZero()` and `isNegative()`.
- Notice that all times in Java 8 are *immutable* so all methods return a new instance.

Local dates

- `Instant` works with time calculated from the epoch, but that is not really usable for humans.
- In Java 8 there are two kinds of “human” times
  - Local date/time
  - Zoned time
- The zoned time is more precise as it takes into account the clock time as well as the time zone.
- Use local time if zone is not required!
  - The launch of Apollo 11 was done at an exact time and place and requires a time zone.
  - Your birthday does not need to be that exact.
- Local date classes also has a number of methods to help calculations.

Example

```
LocalDate today = LocalDate.now();
System.out.println("Today it is " + today);
System.out.println("Tomorrow will be " + today.plusDays(1));
System.out.println("Next week it will be " + today.plusWeeks(1));
System.out.println("The day is " + today.getDayOfWeek());

if(today.isLeapYear())
    System.out.println("This is a leap year.");
else
    System.out.println("This is not a leap year.");

Period until = today.until(LocalDate.of(2020, Month.FEBRUARY, 21));
System.out.println("Time before Star Wars: The Clone Wars Season 7: " +
        until.getYears() + " year(s), " + until.getMonths() +
        " months and " + until.getDays() + " days.");
```

Output

- The, not too surprising, output of the program is:

```
Today it is 2020-02-03
Tomorrow will be 2020-02-04
Next week it will be 2020-02-10
The day is MONDAY
This is a leap year.
Time before Star Wars: The Clone Wars Season 7: 0 year(s), 0 months and 18 days.
```

- Also notice how the program uses the class `Period` which handles time between two different times.
- Some of the methods could possibly return non-existing dates but this is handled by the implementation.
  - Adding one month to the last of January will give the last day in February, not 30 or 31.

Date adjusters and local time

- The class `TemporalAdjuster` helps in adjusting dates.
  - For example to get the first Tuesday of a month.

```
LocalDate firstThursday = LocalDate.of(2014, Month.March, 1)
      .with(TemporalAdjuster.nextOrSame(DayOfWeek.TUESDAY));
```

- Several other such methods exist for `firstDayOfMonth()`, `lastInMonth()` and so on.
- A part from a local date, it is also possible to create *local time*.

```
LocalTime bedtime = LocalTime.of(22, 30);
LocalTime wakeup = bedtime.plusHours(8);
```

- It too has a number of methods (like `plusHours()` shown), similar to the local date.

Zoned time

- Time based on earth's rotation and astral observation is irregular.
- Even more messy is the notion of *time zones* since it is an all human creation.
- Java uses the Internet Assigned Numbers Authority (IANA) database of time zones.
  - This is updated several times per year, mainly due to changing rules for daylight savings time...
- Each time zone has an ID such as `Europe/Stockholm`.
  - In total there are close to 600 time zones
- To set a zoned time for the launch of Apollo 11, write:

```
ZonedDateTime apollo11Launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
  ZoneId.of("America/New_York"));
```

More on date and time...

- There are a number of additional parts to the date and time API
  - New formatters, for example `DateTimeFormatter`.
  - The possibility to print time according to other locales (French, Swedish and so on).
  - Interoperability with legacy code for easy migration.
  - `Clock` makes it possible to use the machine clock for time measuring.
- It is also possible to use other calendars, the following shows today's date according to Thai buddist time (ThaiBuddhist BE 2557-04-04).

```
ThaiBuddhistDate tdate = ThaiBuddhistDate.from(LocalDateTime.now());
```

Resources

- The Internet is a good source of information about Java 8.
- A good starting point is the official Java 8 page at Oracle.
  `http://docs.oracle.com/javase/8/`
- Coreservlets has a nice set of presentations about mainly lambdas and streams
  `http://www.coreservlets.com/java-8-tutorial/`
- A few printed sources are available too, most notably the book “Java SE 8 for the really impatient” by Cay S. Horstmann.
  - Much of this lecture is based on this book.
- Much can be learned by looking at the API or searching the Internet.

Later Versions of Java

Java 9

- As stated, Java 9 was released in 2017.
- The major new addition in Java 9 was the new module system.
- For a long time it has been problematic to package Java applications with dependencies.
  - This is the reason for tools like Maven to exist.
- The module system in Java makes it easier to tell what parts of Java, internal or external, that a project depends on.
- Likewise, it makes it possible to create modules and define what parts are visible for use.
- In the end, this makes for smaller executables only containing the parts used.

JSR-376

- The reason for modules is defined in JSR-376:
  - Reliable configuration – no more use of brittle classpaths.
  - Strong encapsulation – components can decided what parts of its implementation that are accessible by other components.
  - Scalability – only the parts needed are used.
  - Greater platform integrity – strong encapsulation also guards the use of internal API usage.
  - Improved performance.
- In order for this to work, the *entire* JDK has been modularised.
  - All parts, like `base`, `sql`, `xml` and so on, have been made into modules.

Export and require

- Each module *exports* a number of packages and *requires* another set of packages.
- This information is put in a file called `modules-info.java` or can be queried with `java --describe-module [name_of_module]`
- The command `java --describe-module java.sql` will return (for Java 10):

```
java.sql@10.0.1
exports java.sql
exports javax.sql
exports javax.transaction.xa
requires java.base mandated
requires java.logging transitive
requires java.xml transitive
uses java.sql.Driver
```

- The last line declares services the module uses.

More on modules

- There is much more to learn on modules *but*...
- In reality nothing *has* to be done or used.
- This means that we can use Java just as before modules were introduced.
- However, know that modularising Java applications in the long run makes for easier development, depolyment and execution.
  - Especially when creating executable `jar`-files.
- More on modules can be found on the Internet.

JShell

- Another feature of Java 9 was the introduction of *JShell*.
- JShell is a Java **REPL**, read-evaluate-print loop.
- It makes it possible to type code snippets that are immediately read and evaluated with printing of the result.
- This makes for a spelendid learning tool as well as for testing out “stuff”.
- Code snippets can be expressions, statements (individual or multi-line) or classes.
- Most of the Java language is available.
- The application itself is available in most IDEs as well as from the command line by simple running `jshell`.

Example

- This is a small example of running JShell and entering a output statement:

```
tanmsi@vader:~$ jshell
|  Welcome to JShell -- Version 10.0.2
|  For an introduction type: /help intro

jshell> System.out.println(”Hello JSHell”);
Hello JSHell

jshell>
```

- Notice that it loops as the prompt appears again as soon as the statement is executed.

Another example

- The following example shows how to use variables in JShell:

```
jshell> int number1 = 10;
number1 ==> 10

jshell> int number2 = 20;
number2 ==> 20

jshell> int sum = number1 + number2;
sum ==> 30

jshell> System.out.println(”The sum is ” + sum);
The sum is 30
```

- There is even rudimentary support for auto-completion using the tab key.

Much, much more...

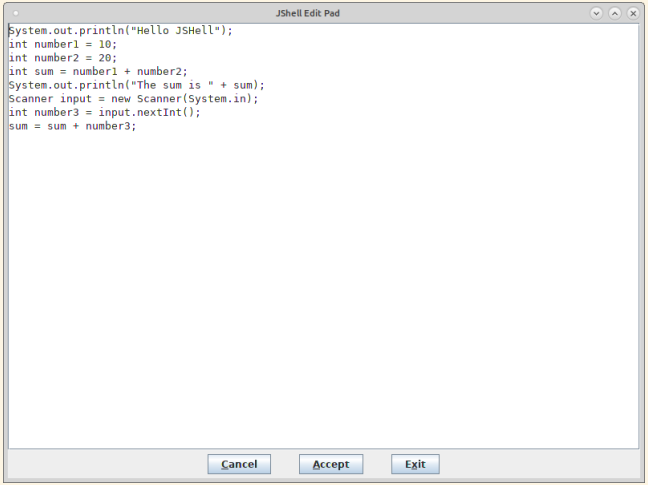
- There is of course much more that can be done.
  - Declaring and using classes, `if` statements and iteration, getting input.
- When reading from the keybord, the `Scanner` class does not need to be imported as it is done automatically.

```
jshell> Scanner input = new Scanner(System.in);
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
   \E][infinity string=\Q∞\E]

jshell> int number3 = input.nextInt();
42
number3 ==> 42
```

- The complete listing can be seen with `/list` and everything can be resetted using `/reset`
- There is even an (ugly) editor available by typing `/edit`!

JShell editor



Java 10

- This version var introduced in March 2018 and has 12 new features.
  - Better garbage collector and memory optimisations for example.
- The most talked about is probably “Local-variable type inference”
- This makes it possible to replace the left hand side of a variable declaration with `var` if the type can be infered by the right hand side.
- This is especially useful when working with streams when the result can be somewhat obscured.
- You should, however, *not* use it only because you are lazy...

Example

- We will leave for you to find out how `SecureRandom` works and what the lambda expression does...

```
public class Ten {
    public static void main(String[] args) {
        var number = 10;
        number++;
        System.out.println(number);

        var list = new ArrayList<Integer>();
        list.add(1); list.add(2); list.add(3);

        System.out.println(list.get(2));

        var randomNumbers = new SecureRandom();
        var numbers = randomNumbers.ints(10, 1, 7).mapToObj(String::valueOf).collect(Collectors.joining(" "));
        System.out.println(numbers);
    }
}
```

- Output:

```
11
3
4 1 6 1 3 2 6 6 4 1
```

Java 11

- The eleventh version of Java was released in September 2018.
- It, again, contains many minor (or major, depending how you see it) features.
  - More work on the garbage collector, flight recorder (data collection framework for JVM) and heap profiling.
- The most important addition is perhaps the new HTTP client with HTTP/2 support.
- New methods have been added to the `String` class.
  - **boolean isBlank()**: Returns true if the string is empty or contains only white space codepoints, otherwise false.
  - **String repeat(int)**: Returns a string whose value is the concatenation of this string repeated count times.
  - **String strip()**: Returns a string whose value is this string, with all leading and trailing whitespace removed.
  - **String stripLeading()**: Returns a string whose value is this string, with all leading whitespace removed.

Java 11 removes stuff...

- Java 11 is, however, mostly remembered for what was removed:
  - Java EE libraries as Java EE was handed over to Eclipse as Jakarta EE.
  - Corba modules for being obsolete.
  - JavaFX is now a separate package not part of standard Java.
- The last part will have an impact later in the course...
- It is now available from `https://openjfx.io/` and needs to be used as a third party library.
  - Much more on this later in the course.

Java 12 and forward

- Java 12 was released in March 2019.
  - More optimisations to the garbage collector.
  - `switch` will be possible to use as a statement (preview).

```
int ass = switch(getAssignments()) {
    case 'A' -> 5;
    case 'B' -> 4;
    case 'C' -> 3;
    case 'D' -> 2;
    default -> 1;
};
```

- Several other smaller fixes where added, but nothing major impacting all users.

Java 13 and onwards

- The current version is Java 13 and it was released in September 2019.
- Not many improvements for the average developer.
  - Again, many improvements to the garbage collection.
- The only new (preview) feature is *text blocks*:

```
String someText = """
Sometimes it is easier to put
everything in a block without
all the newline markers.
""";
```

- The next version is 14 which will be released in March 2020.
- Developement is done in the open, but not much apart from garbage collection related is going to be released.
- Some of the preview features are rumoured to be set as standard.