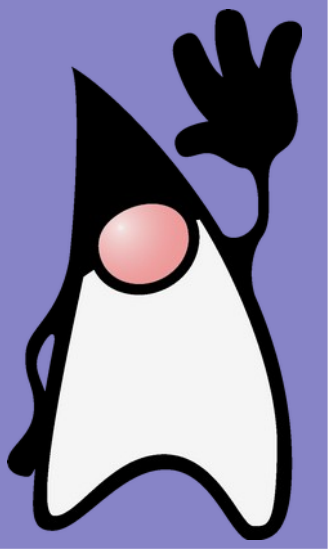
Java

Threads



Overview

- “mutable” string
  - instances of `String` are immutable
- do not extend `String`
  - `String`, `StringBuffer`, `StringBuilder` are final
- `StringBuffer`
  - safe for multiple threads
- `StringBuilder`
  - unsafe for multiple threads
  - since Java 5
- they have the same methods
  - everything for `StringBuffer` holds also for `StringBuilder`
- primary methods – `append` and `insert`
  - defined for all types

Implementation of + on Strings

- till Java 8
- via the class `StringBuilder`

  the expression `x = "a" + 4 + "c"`
  is transformed into
  `x = new StringBuilder("a").append(4).append("c").toString()`

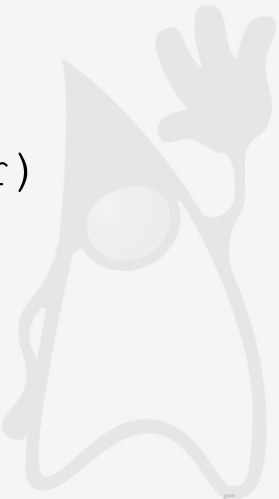
- since Java 9
- via `StringConcatFactory` and `invokedynamic` byte-code instruction
  - in detail in NPRG021

Constructors

- `StringBuffer()`
  - an empty buffer
- `StringBuffer(String str)`
  - a bufer containing str
- `StringBuffer(int length)`
  - an empty buffer with initial capacity of length
    - capacity is automatically adjusted during work with the buffer
- `StringBuffer(CharSequence chs)`
  - `CharSequence`
    - an interface
    - implemented by Strung, StringBuffer, StringBuilder,...

Methods

- `StringBuffer append(`*`type`*` o)`
  - defined for all primitive types, Object, String a StringBuffer
  - converts the parameter to the String and appends it
  - returns **`this`**
- `StringBuffer insert(int offset,` *`type`* `o)`
  - defined for all types as append
  - inserts the string to the given position
  - offset must be >=0 a < current length of the string in the buffer
- `StringBuffer replace(int start, int end, String str)`
  - replaces given chars by the given string
- `StringBuffer reverse()`
  - reverse the buffer

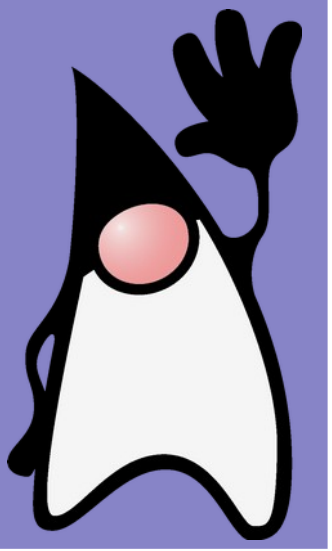


Methods

- `StringBuffer delete(int start, int end)`
  - deletes the chars from the buffer
  - start, end – indexes to the buffer
- `StringBuffer deleteCharAt(int i)`
  - removes a char at the given position
- `char charAt(int i)`
  - returns a char at the given position
- `int length()`
  - current length of the string in the buffer
- `String substring(int start)`
- `String substring(int start, int end)`
  - return a substring

Java

Standard collections



Overview

- a collection ~ an object holding other objects
- e.g. – an array
  - commonly an array is not enough
    - many advantages (built-in type, fast access, elements of primitive types,...)
    - disadvantages – e.g. fixed size
- Java collection library
  - a set of interfaces and classes providing dynamic arrays, hash tables, trees,...
  - a part of the package **java.util**
    - there are also different things in the java.util package

Collections and Java 5

- Java < 5
  - elements of collections – the type **Object**
    - primitive types cannot be used
- Java 5
  - collections as generic types
    - work even without <> – "raw" types
  - primitive types cannot be used also
    - List<int> – compile time error
  - methods have remained the “same”

About arrays yet: java.util.Arrays

- `java.util.Arrays`
  - a set of methods for manipulating arrays
  - part of the collection library
- methods
  - static only
  - most of them defined for all primitive types and the Object class
- `int binarySearch(`*`type`*`[] arr, `*`type`*` key)`
  - searching the given key in the array
  - binary search
  - the array must be sorted increasingly
  - returns the index of the key if the key is present or
    the negative value of the index, at which the key would be inserted into the array

About arrays yet: java.util.Arrays

- `boolean equals(type[] a1, type[] a2)`
  - compares whether the arrays have the same length and contain the same elements
  - elements are the same if
    `(e1==null ? e2==null : e1.equals(e2))`
- `void fill(type[] arr, typ val)`
  - fills the array with the parameter `val`
- `void fill(type[] arr, int from, int to, typ val)`
  - fills the given part of the array with the parameter `val`
- `void sort(type[] arr)`
  - sorts the array increasingly
  - quicksort for primitive types, mergesort for Object
- `void sort(type[] arr, int from, int to)`
  - sorts the given part of the array

Sorting arrays

- `void sort(Object[] arr)`
  - elements of the array must be comparable, i.e. must implement the interface java.lang.Comparable<T>
    - the method int compareTo(T o)
- `void sort(T[] arr, Comparator<? super T> c)`
  - elements still have to be comparable
  - for comparing, the c parameter is used
  - the interface java.util.Comparator<T>
    - int compare(T o1, T o2)
  - for searching
    - int binarySearch(T[] a, T key, Comparator<? super T> c)
- `void parallelSort(`*typ*`[] a)`
  - parallel mergesort
    - ForkJoinPool

java.util.Arrays

- `typ[] copyOf(typ[] original, int newLength)`
- `typ[] copyOfRange(typ[] original, int from, int to)`
- `<T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType)`
  - a copy of an array

- `<T> List<T> asList(T... a)`
  - array => list

Basic collections

- two basic kinds – the interfaces **Collection** and **Map**
- **Collection<E>**
  - group of objects
  - **List<E>**
    - holds objects in some order
  - **Set<E>**
    - holds objects without duplications
  - **Queue<E>** (since Java 5)
    - a queue of objects
  - **Deque<E>** (since Java 6)
    - double ended queue
- **Map<K,V>**
  - group of tuples key–value
- for each kind there is always at least one implementation
  - but typically several of them

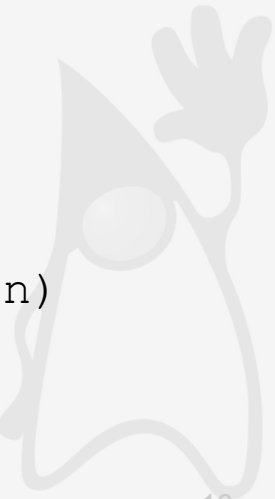
Collections hierarchy

- collections do not implement directly the particular interface
- they implement classes AbstractSet, AbstractList, AbstractMap, AbstractQueue, AbstractDeque
  - abstract classes
  - provide basic functionality of the particular collection
  - each implementation of interfaces Set, List,... should extend the particular Abstract class

- using collections
  - typically through the interface of the particular collection
  - e.g. `List<Xyx> c = new ArrayList<>()`
  - possible to easily change the implementation

Iterator<E>

- collections need not to support direct access to the elements
- collections have the method
  - Iterator<E> iterator()
  - returns an object of the type Iterator<E>, which allows iterating over all elements in the collection
- methods
  - `E next()` – next element in the collection
  - `boolean hasNext()` – true, if there are next elements
  - `void remove()`
    - removes the last returned element from the collection
    - `default` since Java 8 (throws UnsupportedOperationException)
  - `default void forEachRemaining(Consumer<? super E> action)`
    - since Java 8



Iterator<E>

- implementation of the iterator and its relation to elements of the collection depends on the particular implementation

```
List c = new ....
...
Iterator e = c.iterator();
while (e.hasNext()) {
  System.out.println(e.next());
}
```

- **for** cycle for collections with the iterator
  - i.e. implementing the `Iterable` interface

```
for (x:c) {
  System.out.println(x);
}
```

Iterable<T>

- `Iterator<T> iterator()`
  - returns an iterator

- `default void forEach(Consumer<? super T> action)`
  - performs the given action for each element
  - since Java 8

- `default Spliterator<T> spliterator()`
  - returns a spliterator
  - since Java 8

Collection<E>

- `boolean add(E o)`
  - adds object to the collection
  - returns false if addition failed
  - optional method
- `boolean addAll(Collection<? extends E> c)`
  - adds all methods
  - returns true if at least on object was added
  - optional method
- `void clear()`
  - removes all objects
  - optional method
- `boolean contains(E o)`
  - returns true if the given object is in the collection

Collection<E>

- `boolean containsAll(Collection<?> c)`
  - returns true if all objects are in the collection
- `boolean isEmpty()`
- `Iterator<E> iterator()`
- `boolean remove(E o)`
  - returns true if the object has been removed
  - optional method
- `boolean removeAll(Collection<?> c)`
  - tries to remove objects from the collection
  - returns true if at least on object was removed
  - optional method
- `boolean retainAll(Collection<?> c)`
  - removes all object that are not in `c`
  - optional method

Collection<E>

- `int size()`
  - number of objects in the collection
- `default Spliterator<E> spliterator()`
- `default boolean removeIf(Predicate<? super E> filter)`
  - removes objects that satisfy the given predicate
- `Object[] toArray()`
  - returns an array with all objects in the collection
- `T[] toArray(T[] a)`
  - returns an array with all objects in the collection
  - the returned array is of the same type as `a`

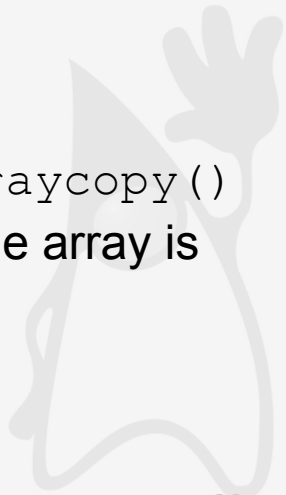
```
List<String> c;
....
String[] str = c.toArray(new String[0]);
```

Which size put here?

Collection<E>

- `T[] toArray(IntFunction<T[]> generator)`
  - equivalent to `toArray(T[] a)`
  - argument – a reference to an "array constructor"
    `String[] y = x.toArray(String[]::new);`
    - internally equivalent to `toArray(new String[0])`

- why is `toArray(zero_sized_array)` faster
  - internally creates a new array, which is immediately filled via `System.arraycopy()`
  - if `System.arraycopy()` is called immediately after the array creation, the array is not filled with default values



List<E>

- extends Collection
- contains objects in a particular order
- one object can held several times
- has the method `E get(int index)`
  - returns an object on the given position
- `default void sort(Comparator<? super E>)`
  - od Java 8
- in addition to the Iterator, it offers the ListIterator
- ListIterator
  - extends Iterator
  - allows
    - iterating in the reverse order – methods `previous()`, `hasPrevious()`
    - adding and replacing objects – methods `add()`, `set()`

List<E>

- two implementations
- **ArrayList**
  - implemented by an array
  - fast random access
  - slow addition in the middle
- **LinkedList**
  - fast sequential access
  - slow random access
  - has additional methods
    - addFirst()
    - removeFirst()
    - addLast()
    - removeLast()
    - getFirst()
    - getLast()

Set<E>

- extends Collection
- no new method
- each object can be held just once
- several implementations
- **HashSet**
  - fast searching in the collection
  - does not keep order of objects
- **TreeSet**
  - Set implemented by Red-Black trees
  - implements **SortedSet**
    - objects are sorted
    - allows returning of a subcollection (subset)
- **LinkedHashSet**
  - as HashSet but keeps order of objects

Queue<E>

- extends Collection
- a queue of elements
- commonly FIFO
- can have a fixed size
- 2 forms of methods for the same functionality
  - throws an exception if the operation fails (add, remove, element)
  - returns a special value if the operation fails (offer, poll, peek)
- `add(E e), offer(E e)`
  - add an element
- `E remove(), E poll()`
  - remove and return an element
- `E element(), E peek()`
  - return an element but do not remove it

Deque<E>

- double ended queue
- extends Queue
- similar methods like the Queue, but twice
  - for the start of the queue
  - for the end of the queue

- **`addFirst(E)`** **`offerFirst(E)`**
- **`removeFirst()`** **`pollFirst()`**
- **`getFirst()`** **`peekFirst()`**

- **`addLast(E)`** **`offerLast(E)`**
- **`removeLast()`** **`pollLast()`**
- **`getLast()`** **`peekLast()`**

Map<K,V>

- does not extend **Collection**
- collection of tuples key-value
  ~ associative array
- each key is contained only once
- methods
  - `V put(K key, V value)`
    - associates the key with the value
    - returns the previous value or null
  - `V get(K key)`
    - returns associated value
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object val)`
  - `Set<K> keySet()`
  - `Collection<V> values()`

Map<K,V>: implementation

- several implementations
- **HashMap**
  - via hash table
  - constant time for adding and removing
- **LinkedHashMap**
  - as HashMap
  - plus keeps order when iterated (order by insertion or LRU)
  - slightly slower
    - but iterating is faster
- **TreeMap**
  - via Red-Black trees
  - implements the interface SortedMap
    - objects are sorted

HashMap<K,V>

- objects must correctly implement the method `hashCode()`
- two same objects (by the equals method) must return always the same hashcode
- different object need not to return different hashCode
- hashing with chains
  - different objects with the same hashCode will be in the same chain
- HashMap has at the beginning some capacity
- usage factor = stored objects / capacity
- if a particular factor (default is 0.75) is reached, capacity is increased and the hash map re-hashed
  - because of performance

Class Collections

- similar to the Arrays class
- static methods for Collections manipulation
- methods
  - binarySearch
  - fill
  - sort
  - rotate
  - shuffle
  - reverse
  - ...

Synchronization

- most of the collections are not immune to concurrent threads
- safe (synchronized) collections are created by the methods like unmodifiable
- methods in the Collections
  - synchronizedCollection
  - synchronizedList
  - synchronizedSet
  - synchronizedMap

Unmodifiable colections

- methods of the Collections
  - unmodifiableCollection
  - unmodifiableList
  - unmodifiableSet
  - unmodifiableMap
- have a single parameter (the particular type of a collection)
- returns "read-only" collection with the same content as the given collection

Unmodifiable colections

- the **`of`** method for easy creation
  - since Java 9
  - for all collection types
    - List, Set, Map

```
List<String> list = List.of("foo", "bar", "baz");
Set<String> set = Set.of("foo", "bar", "baz");
Map<String, String> map = Map.of("foo", "a", "bar", "b",
"baz", "c");
```

- 12 overloaded methods **`of`**
  - **`of()`**, **`of(E e)`**, **`of(E e1, E e2)`**, …, till **`of`** with 10 arguments
  - **`of(E... elems)`**

Arrays.asList() vs. List.of()

- `Arrays.asList()` internally uses the supplied array
  - changes to the array -> changes to the list
  - changes to the list -> changes to the array
  - but no additional element can be added to the list

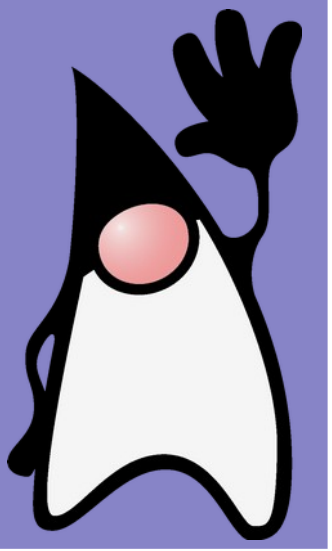
- `List.of()`
  - completely unmodifiable list

"Old" collections (Java 1.0, 1.1)

- since Java 1.2 collections re-created (List, Set, Map)
- old collection
  - should not be used
    - but sometimes have to
  - included do the new version (i.e. also implement List, Set or Map)
- Vector
  - like ArrayList
- Enumeration
  - like Iterator
- Hashtable
  - like HashMap
- ...

Java

java.util.stream



Overview

- since Java 8
  - use of lambda expressions

- processing collections
  - a programmer define what to achieve
  - scheduling of operations is left for an implementation
  - functional approach
    - “map & reduce”
- streams of data
  - can be obtained from collections, arrays,...

- in fact a replacement for the iterator
  - the iterator prescribes traversing strategy
  - it does not allow for parallelization

Example

- List<String> words = …// a list of words

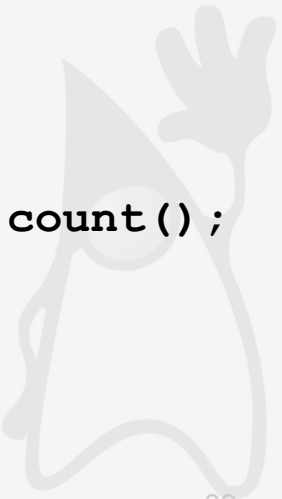
- number of words with length bigger than 10
  - via iterator

```
int count = 0;
for (String w : words) {
   if (w.length() > 10) count++;
}
```

  - via stream

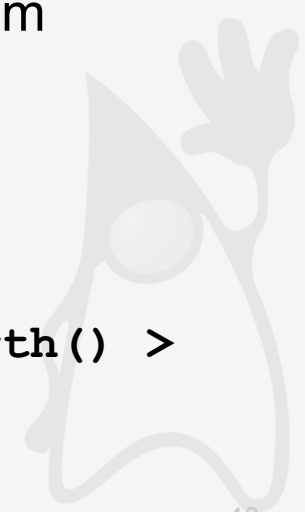
```
long count = words.stream().filter(w -> w.length() > 10).count();
```

- both solutions are correct
- but the iterator prescribes traversing and cannot be parallelized



Properties of streams

- java.util.stream.Stream<T>
  - interface
- a stream does not store its elements
  - they are stored in an underlying collection or generated
- stream operations do not modify their source but create a new stream
- when possible, stream operations are lazy
- can be easily parallelized
  - `long count = words.parallelStream().filter(w -> w.length() > 12).count();`



Stream operations

- stream pipeline
  - a sequence of stream operations

- two types of stream operations
  - intermediate
    - creates a new stream
    - lazy
    - does not begin until the terminal operation of the pipeline is executed
  - terminal
    - (in almost all cases) eager
    - consumes the stream pipeline
    - does not produce a stream

Stream operations

- operation parameters – functional interfaces
  - actual parameters – lambdas

- package **`java.util.function`**
  - **`Function<T, R>`**
    `R apply(T t)`
  - **`Predicate<T>`**
    `boolean test(T t)`
  - **`Supplier<T>`**
    `T get()`
  - **`UnaryOperator<T> extends Function<T,T>`**
    `T apply(T t)`
  - **`BinaryOperator<T> extends BiFunction<T,T,T>`**
  - ...

A stream creation

- *collection*.stream()
- *collection*.parallelStream()

- methods of the Stream interface
  - `static <T> Stream<T> of(T... values)`
  - `static <T> Stream<T> empty()`
  - `static <T> Stream<T> generate(Supplier<T> s)`
    - generates infinite streams

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

A stream creation

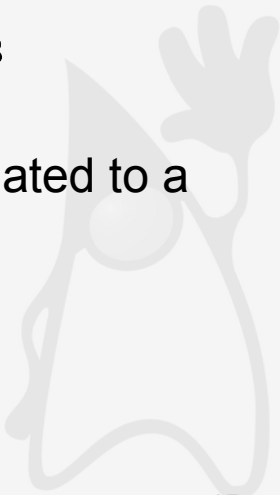
- methods of the Stream interface (cont.)
  - **`static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`**
    - generates infinite streams
    - seed – first element
    - next elements – f(seed), f(f(seed)),...
- java.nio.files.Files
  - **`static Stream<String> lines(Path path)`**
- ...

Intermediate operations

- **`Stream<T> filter(Predicate<? super T> predicate)`**
  - returns a stream with elements that match the given predicate

- **`<R> Stream<R> map(Function<? super T,? extends R> mapper)`**
  - returns a stream with the results of applying the given function to the source stream elements

- **`<R> Stream<R>  flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`**
  - as map(), but for the functions returning a stream and results are concatenated to a single stream, i.e. the result is not a stream of streams
  - one-to-many mapping



Intermediate operations

- **`<R> Stream<R> mapMulti(BiConsumer<? super T,? super Consumer<R>> mapper)`**
  - similar to flatMap (one-to-many)
  - mapper parameter – a function generating elements

```
Stream<Number> numbers = ... ;
List<Integer> integers = numbers.
     mapMulti((number, consumer) -> {
         if (number instanceof Integer i)
             consumer.accept(i);
         }).collect(Collectors.toList());
```

  - preferred if
    - generated “sub”-stream is small, or
    - an imperative approach for generating result elements is easier

Intermediate operations

- `Stream<T> skip(long n)`
- `Stream<T> limit(long maxSize)`
- `static <T> Stream<T>  concat(Stream<? extends T> a, Stream<? extends T> b)`

- `Stream<T> distinct()`
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`

Terminal operations

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`
- `Optional<T> findFirst()`
- `long count()`

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `T reduce(T identity, BinaryOperator<T> accumulator)`

Terminal operations

- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`

- `<R,A> R collect(Collector<? super T,A,R> collector)`
- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
  - předpřipravené kolektory
    - toList, toSet, toMap

Terminal operations

- `void forEach(Consumer<? super T> action)`
- `void forEachOrdered(Consumer<? super T> action)`

“Primitive” streams

- the `Stream<T>` interface
  - cannot be directly used with primitive types
- `IntStream`
  - for int but also for byte, short, char, boolean
- `LongStream`
- `DoubleStream`
  - for double and float

- methods of `Stream<T>`
  - `IntStream mapToInt(ToIntFunction<? super T> mapper)`
  - `LongStream mapToLong(ToLongFunction<? super T> mapper)`
  - `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`

Parallel streams

- What is printed out?

```
List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4);
int sum = listOfNumbers.parallelStream().
                                    reduce(5, Integer::sum);
System.out.println(sum);
```

- Be carefull, which operations can be run in parallel

Parallel streams

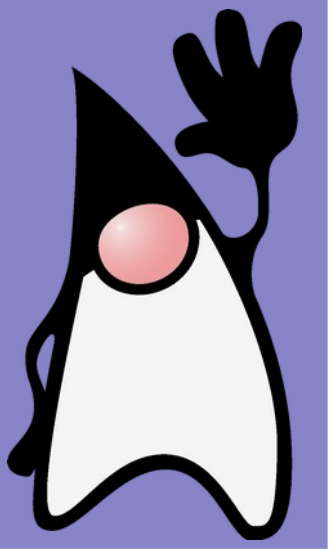
- When to use?
  - managing threads, splitting data,… are expensive operations!
  - for small data parallel stream will have worse performance

- for **estimations** – NQ model
  - N… number of source data elements
  - Q… amount of computation performed per data element

  - N*Q > 10000
    - use parallel
  - N*Q < 10000
    - use sequential

Java

About functional programming in general



Functional programming

- a function in FP ~ “a mathematical function”
  - takes arguments
  - returns a result(s)
  - **no side-effects!!!**
    - **WARNING: I/O operations are also side-effects**
  - no exception thrown
    - can be considered as side-effects too
  - lazy if possible

- data (lists) are non-modifiable
  - functions return new ones

Lazy functions

- example

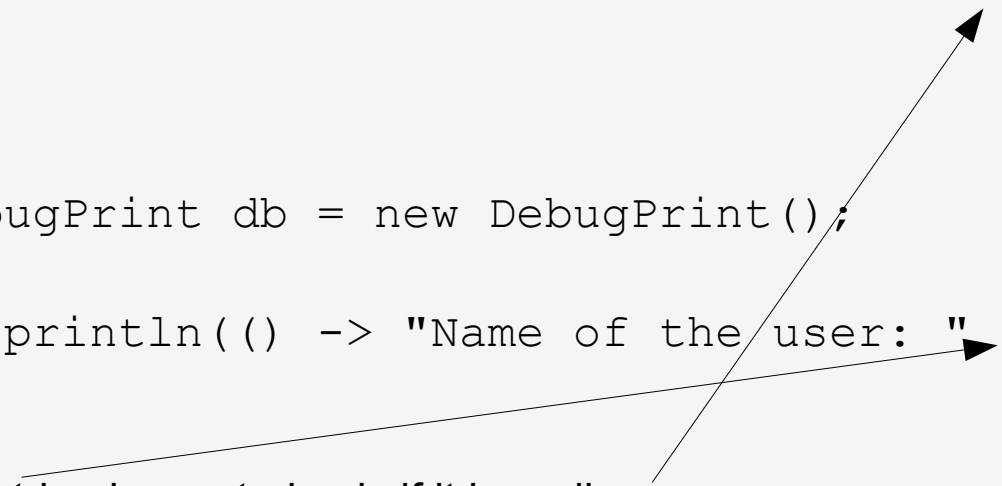
```
class DebugPrint {
  private boolean debug;
  public void setDebug(boolean d) { debug = d; }
  public void println(String s) {
    if (debug) { System.out.println(s); }
  }
}
...
DebugPrint db = new DebugPrint();
...
db.println("Name of the user: " + userName);
```

- the string is necessary only if `debug == true`
  BUT it is created always

Lazy functions

- better

```
class DebugPrint {
  private boolean debug;
  public void setDebug(boolean d) { debug = d; }
  public void println(Supplier<String> c) {
    if (debug) { System.out.println(c.get()); }
  }
}
...
DebugPrint db = new DebugPrint();
...
db.println(() -> "Name of the user: " + userName);
```



- the string is created only if it is really necessary

Not throwing exceptions

- a special value returned in case of error
- null is not ideal
  - calls cannot be chained

- Optional<T>
  - class
  - a container for value that can be null
  - methods
    - boolean isPresent()
    - T get()
    - void ifPresent(Consumer<? super T> consumer)
    - ...
  - new instances
    - static <T> Optional<T> empty()
    - static <T> Optional<T> of(T value)
    - static <T> Optional<T> ofNullable(T value)