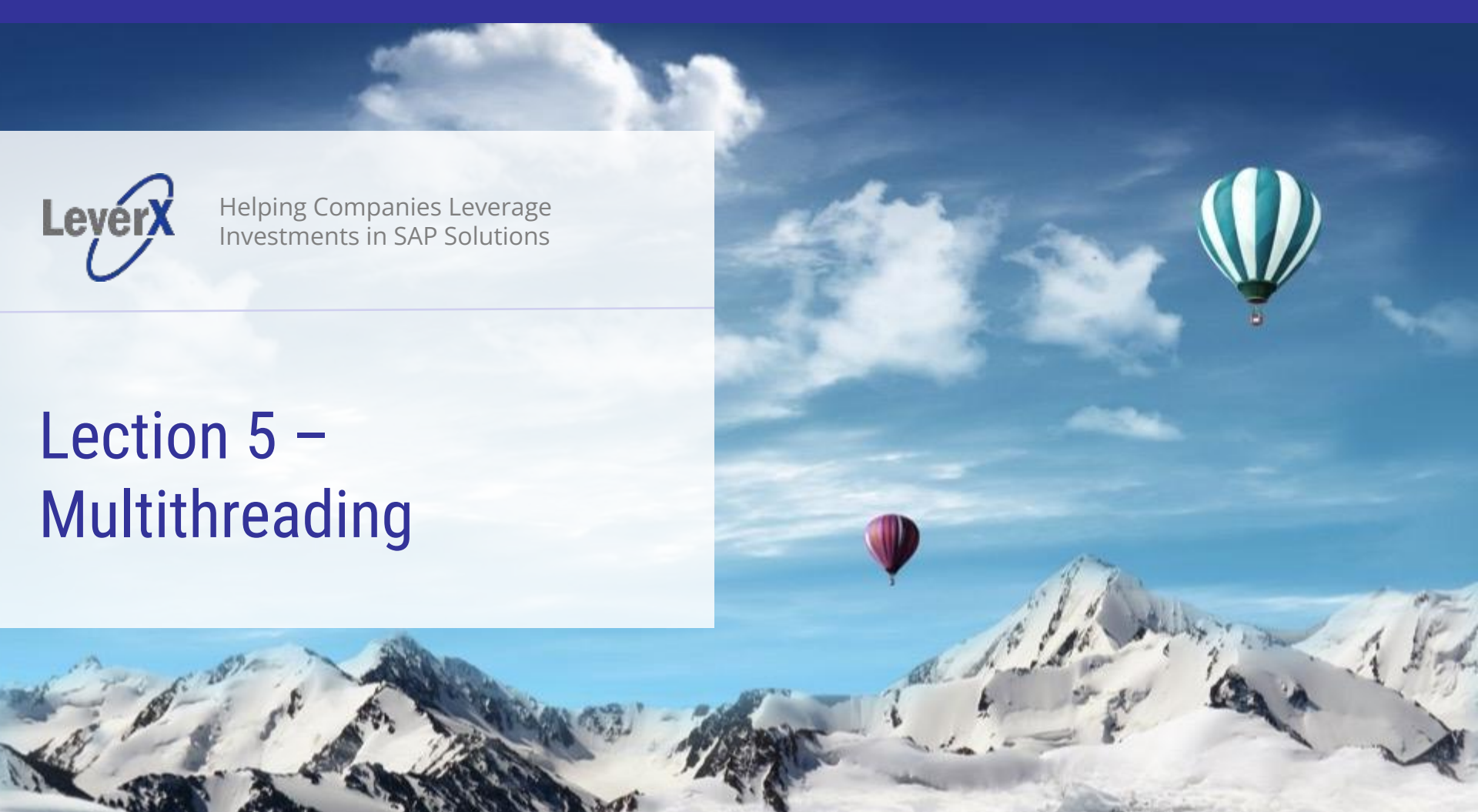




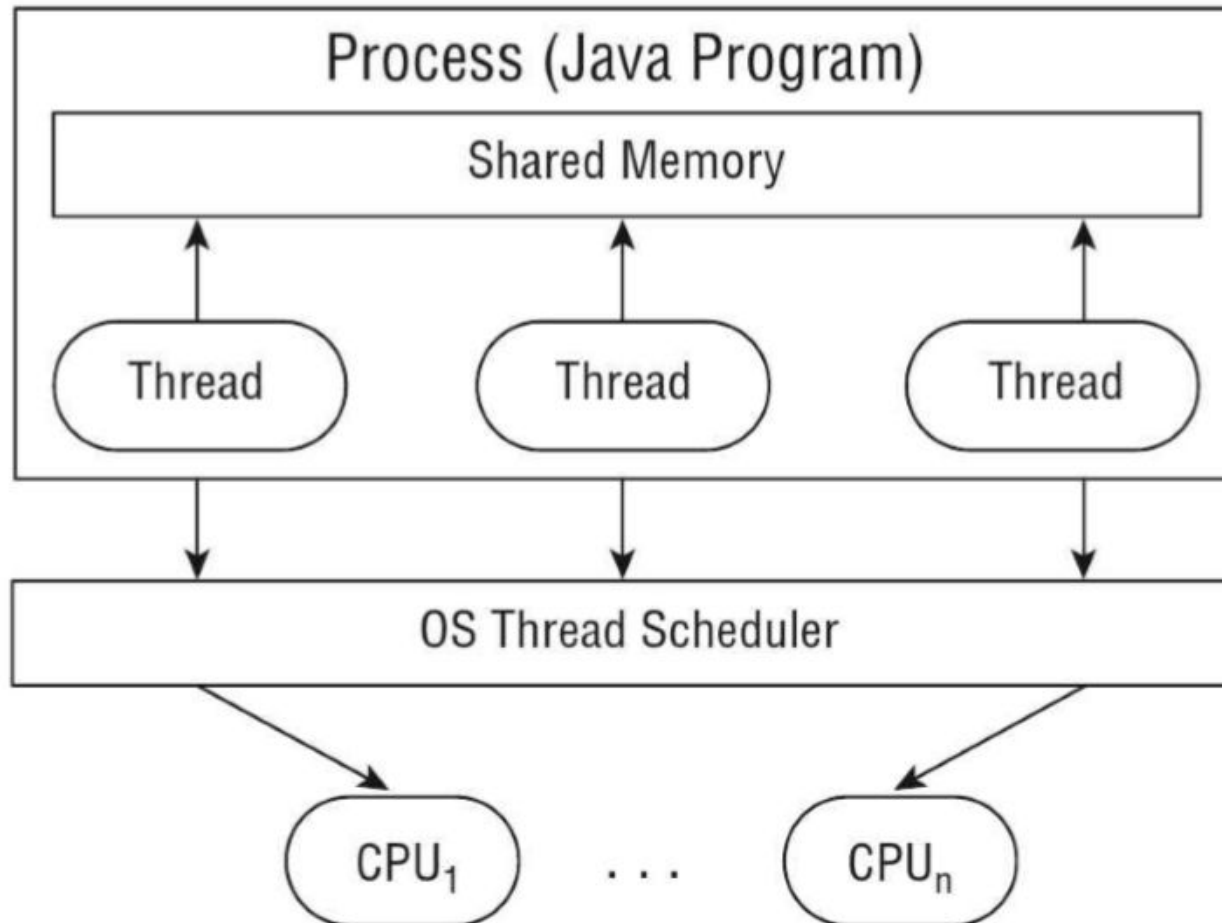
Helping Companies Leverage
Investments in SAP Solutions

Lection 5 – Multithreading



MULTITHREADING

Process model



```
8  ▶
9  ⚠
10
11
12  ⚠
13
14
15
16  Executor exec = Executors.newCachedThreadPool();
17  // or Executors.newFixedThreadPool(numberOfThreads);
18  exec.execute(hellos);
19  exec.execute(goodbyes);
20
21  // ...
22  // Hello 158
23  // Hello 159
24  // Goodbye 105
25  // Hello 160
26  // Goodbye 106
27  // Hello 161
28  // ...
29  }
```

```
10 ExecutorService exec = Executors.newFixedThreadPool( nThreads: 3);
11
12 Callable<String> task = () -> {
13     // do smth
14     if (Thread.currentThread().isInterrupted()) {
15         return null;
16     }
17     // do smth else
18     return "";
19 };
20
21 Future<String> result = exec.submit(task);
22 // V get() throws InterruptedException, ExecutionException;
23 // V get(long timeout, TimeUnit unit) throws ...;
24 // boolean cancel(boolean mayInterruptIfRunning)
25 // boolean isCancelled()
26 // boolean isDone()
```

```
8 private static /*volatile*/ boolean done = false;
9
10 public static void main(String[] args) {
11
12     Runnable hellos = () -> {
13         for (int i = 1; i <= 100; i++)
14             System.out.println("Hello " + i);
15         done = true;
16     };
17     Runnable goodbye = () -> {
18         int i = 1;
19         while (!done)
20             i++;
21         System.out.println("Goodbye " + i);
22     };
23
24     Executor executor = Executors.newCachedThreadPool();
25     executor.execute(hellos);
26     executor.execute(goodbye);
27     // Result:
28     // ...
29     // Hello 97
30     // Hello 98
31     // Hello 99
32     // Hello 100
33     // .. and not completed yet
34 }
```

```
8      private static volatile int[] count = new int[]{0};
9
10     public static void main(String[] args) {
11
12         Executor exec = Executors.newCachedThreadPool();
13
14         for (int i = 1; i <= 100; i++) {
15             int taskId = i;
16             Runnable task = () -> {
17                 for (int k = 1; k <= 1000; k++) count[0]++;
18                 System.out.println(taskId + ":" + count[0]);
19             };
20
21             exec.execute(task);
22         }
23         // 2:1312
24         // 1:1312
25         // 5:2432
26         // 3:3066
27         // ...
28     }
```


Strategies for Safe Concurrency:

- 1. Confinement: just say 'no' when it comes to sharing data among task**
- 2. Immutability: it's safe to share immutable objects**
- 3. Locking: by granting only one task at a time to access a data structure, one can keep it from being damaged.**


```
10 List<String> coll = Arrays.asList("Abc", "Eklmn", "Xys");
11 long result = coll.parallelStream().filter(s -> s.startsWith("A")).count();
12
13 Arrays.parallelSort((String[]) coll.toArray(),
14     Comparator.comparing(String::length));
15
16 int[] shortWords = new int[3];
17 coll.parallelStream().forEach(s -> {
18     if (s.length() < 3) {
19         shortWords[s.length()]++;
20     } // ERROR! DON'T DO THIS - RACE CONDITION
21 });
22 System.out.println(Arrays.toString(shortWords));
```

```
6 public static void main(String[] args) {  
7  
8     ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();  
9     map.put("key", 123L);  
10    map.putIfAbsent("key", 321L); // {"key" : 123L}  
11  
12    Long oldValue = map.get("key");  
13    Long newValue = oldValue == null ? 1 : oldValue + 1;  
14    map.put("key", newValue); // Potential error, don't do this  
15  
16    // use this instead  
17    map.compute( key: "key", (k, v) -> v == null ? 1 : v + 1);  
18  
19    // or this  
20    map.merge( key: "key", value: 1L, (existingV, newV) -> existingV + newV);  
21 }
```

Метод	Обычное действие	Действие при ошибке
<code>put()</code>	Вводит элемент в хвост очереди	Блокирует операцию, если очередь заполнена
<code>take()</code>	Удаляет элемент из головы очереди и возвращает его	Блокирует операцию, если очередь пуста
<code>add()</code>	Вводит элемент в хвост очереди	Генерирует исключение типа <code>IllegalStateException</code> , если очередь заполнена
<code>remove()</code>	Удаляет элемент из головы очереди и возвращает его	Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>element()</code>	Возвращает элемент из головы очереди	Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>offer()</code>	Вводит элемент в очередь и возвращает логическое значение <code>true</code>	Возвращает логическое значение <code>false</code> , если очередь заполнена
<code>poll()</code>	Удаляет элемент из головы очереди и возвращает его	Возвращает пустое значение <code>null</code> , если очередь пуста
<code>peek()</code>	Возвращает элемент из головы очереди	Возвращает пустое значение <code>null</code> , если очередь пуста

```
9 public class Slide9 {
10
11     static AtomicLong nextNumber = new AtomicLong();
12
13     static List<Long> listOne = Arrays.asList(321L, 123L, -222L);
14     static List<Long> listTwo = Arrays.asList(900L, -300L, 200L);
15     static AtomicLong maxNumber = new AtomicLong();
16
17     public static void main(String[] args) throws InterruptedException {
18         Executor executor = Executors.newCachedThreadPool();
19         executor.execute(() -> {
20             System.out.println(nextNumber.incrementAndGet());
21         });
22
23         /* ***** */
24
25         executor.execute(createFindMaxFunction(listOne));
26         executor.execute(createFindMaxFunction(listTwo));
27
28         Thread.sleep(5000);
29         System.out.println(maxNumber);
30     }
31
32     private static Runnable createFindMaxFunction(List<Long> numbers) {
33         return () -> {
34             for (Long number : numbers) {
35                 maxNumber.updateAndGet(x -> Math.min(x, number));
36             }
37         };
38     }
39 }
```

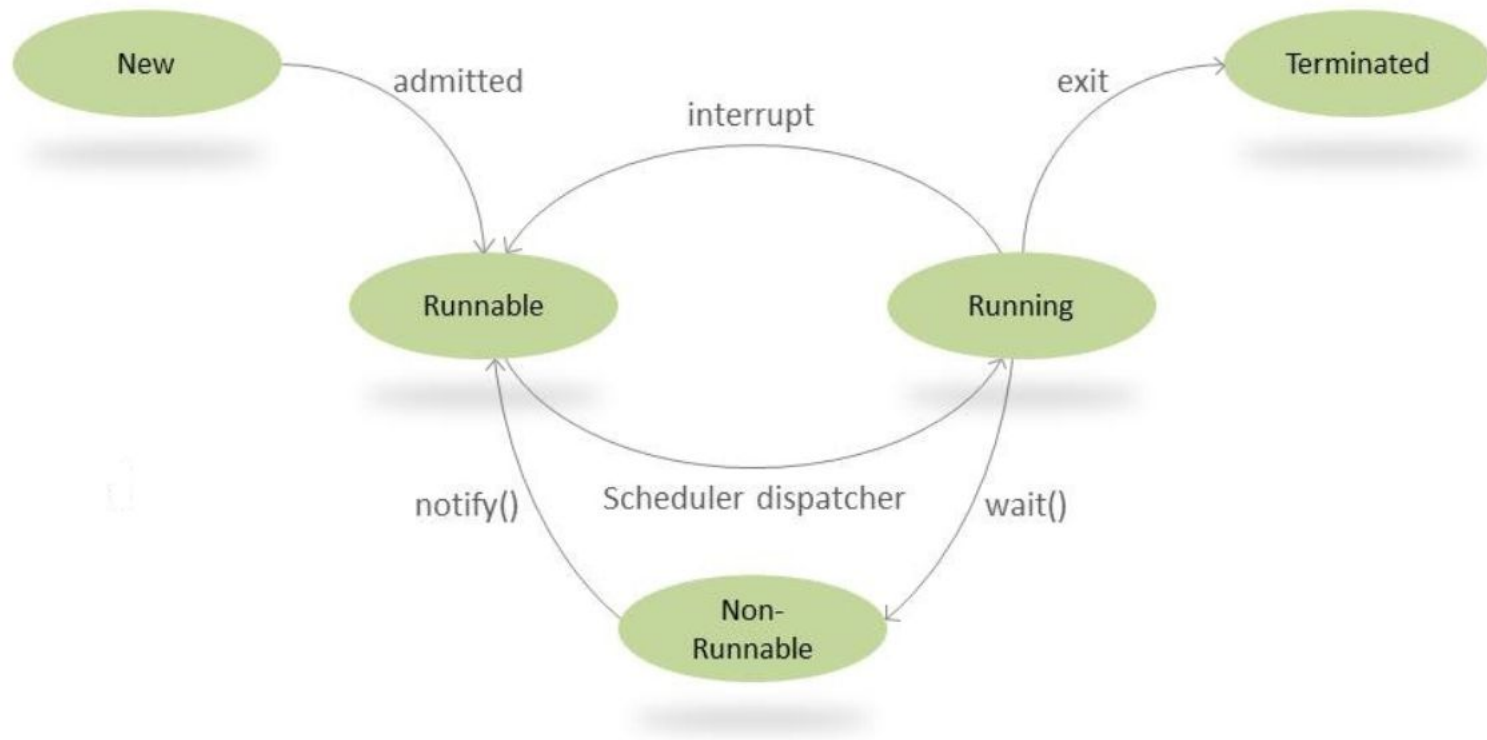
Multithreading



```
8 public class Slide10 {
9     public static void main(String[] args) throws InterruptedException {
10         // 1. initialize shared variable
11         ClassWithLock classWithLock = new ClassWithLock();
12
13         // 2. create some job
14         Runnable runnable = () -> {
15             for (int i = 0; i < 5000; i++) {
16                 classWithLock.incrementCount();
17             }
18         };
19
20         // 3. and execute this job twice
21         runTwice(runnable);
22
23         // 4. wait a little bit (without it output will be 0 or smth)
24         Thread.sleep(5000);
25         System.out.println(classWithLock);
26     }
27
28     private static void runTwice(Runnable runnable) {...}
29
30 }
31
32
33
34
35 class ClassWithLock {
36     private long count = 0;
37     private Lock countLock = new ReentrantLock();
38
39     public void incrementCount() {
40         countLock.lock();
41         try {
42             count++;
43         } finally {
44             countLock.unlock();
45         }
46     }
47
48     public String toString() { return "" + count; }
49
50 }
51 }
```









```
6 public class Slide11 {
7     public static void main(String[] args) throws InterruptedException {
8         // 1. initialize shared variable
9         ClassWithSync classWithSync = new ClassWithSync();
10
11         // 2. create some job
12         Runnable runnable = () -> {
13             for (int i = 0; i < 5000; i++) {
14                 classWithSync.incrementCount();
15             }
16         };
17
18         // 3. and execute this job twice
19         runTwice(runnable);
20
21         // 4. wait a little bit (without it output will be 0 or smth)
22         Thread.sleep(5000);
23         System.out.println(classWithSync);
24     }
25
26     private static void runTwice(Runnable runnable) {...}
27
28 }
29
30 class ClassWithSync {
31     private long count = 0;
32     public synchronized void incrementCount() { count++; }
33     public String toString() { return "" + count; }
34 }
35
36 class Flag {
37     private boolean done;
38     public synchronized void set() { done = true; }
39     public synchronized boolean get() { return done; }
40 }
```



Multithreading



```
15 public static void main(String[] args) {
16     WaitAndNotify shared = new WaitAndNotify();
17
18     Runnable puts = () -> {
19         for (int i = 1; i <= 10; i++) {
20             try { shared.put(i); } catch (InterruptedException e) { }
21         }
22     };
23     Runnable gets = () -> {
24         for (int i = 1; i <= 10; i++) {
25             try { System.out.println(shared.get()); } catch (InterruptedException e) { }
26         }
27     };
28
29     execute(puts, gets);
30 }
31
32
33 class WaitAndNotify {
34     boolean flag = false; int value;
35
36     synchronized void put(int i) throws InterruptedException {
37         while (flag) { wait(); }
38         flag = true;
39         value = i;
40         notifyAll();
41     }
42
43     synchronized int get() throws InterruptedException {
44         while (!flag) { wait(); }
45         flag = false;
46         notifyAll();
47         return value;
48     }
}
```

```
4  ▶  
5  ⬆  
6
7
8
9
10
11
12  
13
14
15
16
17
18  
19
20  
21  

public static void main(String[] args) throws InterruptedException {
    Runnable task = () -> {
        try {
            Thread.sleep( millis: 7000 );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("After 7 seconds");
    };
    Thread thread = new Thread(task);
    thread.start();

    thread.join();
    System.out.println("Main thread is trying to do smth!");
    // After 7 seconds (after 7 seconds :)
    // Main thread is trying to do smth (only after previous line)
    // Without 'join' order will be reversed
}
```

```
4  public static void main(String[] args) {  
5      Runnable taskWithCheck = () -> {  
6          while (true) {  
7              if (Thread.currentThread().isInterrupted()) return;  
8              // do smth  
9          }  
10     };  
11  
12     Runnable taskThatWantToSleep = () -> {  
13         try {  
14             while (true){  
15                 // do smth  
16                 Thread.sleep( millis: 1000);  
17             }  
18         } catch (InterruptedException ex) {  
19             // do nothing  
20         }  
21     };  
22 }
```

```
7 public static final ThreadLocal<NumberFormat> currencyFormat =  
8     ThreadLocal.withInitial(NumberFormat::getCurrencyInstance);  
9  
10 public static void main(String[] args) {  
11     String amountDue = currencyFormat.get().format(number: 3.99);  
12     System.out.println(amountDue); // $3.99  
13 }  
14 }
```

GENERIC (PART 2)

Generics (Part 2)



```
6  ▶ public class Slide5 {
7  ▶     public static void main(String[] args) {
8      List<Manager> managers = new ArrayList<>();
9      // List<Employee> employees = managers; // Compile error, incompatible
10     // employees.add(new Employee());
11
12     addNewEmployee(managers.toArray(new Manager[1])); // this is ok, but ...
13     }
14
15  @ public static void addNewEmployee(Employee[] employees) {
16     employees[0] = new Employee( name: "Sergey Petrovich");
17     // Runtime error (ArrayStoreException)
18     }
19     }
20     class Manager extends Employee {
21         public Manager(String name) { super(name); }
22         public String getName() { return "Manager: " + name; }
23         public String toString() { return getName(); }
24     }
25     class Employee {
26         protected String name;
27         public Employee(String name) { this.name = name; }
28         public String getName() { return "Employee: " + name; }
29     }
```


Generics (Part 2)



```
7  ▶  public static void main(String[] args) {
8      List<Employee> employees = new ArrayList<>();
9      employees.add(new Manager( name: "Dzianis"));
10     printNames(employees);
11 }
12
13 @  public static void printNames(List<? extends Employee> employees) {
14     for (int i = 0; i < employees.size(); i++) {
15         Employee employee = employees.get(i);
16         System.out.println(employee.getName());
17     }
18
19     // Compile errors:
20     // employees.add(new Employee("LeverX Employee"));
21     // employees.add(new Manager("LeverX Manager"));
22     employees.add(null);
23 }
```


Generics (Part 2)



```
7  ▶ public class Slide7 {
8  ▶  public static void main(String[] args) {
9      List<Employee> employees = new ArrayList<>();
10     employees.add(new Manager( name: "Dzianis"));
11
12     Predicate<Object> predicate = e -> e.toString().startsWith("Manager");
13     printNames(employees, predicate); // but printNames(employees, e -> ...); works
14 }
15
16 @ public static void printNames(List<? extends Employee> employees,
17     Predicate<? super Employee> predicate) {
18     for (int i = 0; i < employees.size(); i++) {
19         Employee employee = employees.get(i);
20         if (predicate.test(employee)) {
21             System.out.println(employee);
22         }
23     }
24 }
25 }
```

Generics (Part 2)



```
6 public class Slide8 {
7
8     @
9     public static <T> void printNames(T[] elements,
10                                     Predicate<T> predicate) {
11         for (T e : elements) {
12             if (predicate.test(e)) {
13                 System.out.println(e.toString());
14             }
15         }
16
17     @
18     public static boolean hasNulls(List<?> elements) {
19         for (Object e : elements) {
20             if (e == null) return true;
21         }
22         return false;
23     }
24 }
```

Generics (Part 2)



```
7  class Entry<K extends Comparable<? super K> & Serializable, V extends Serializable> {
8      private K key;
9      private V value;
10
11      public Entry(K key, V value) {...}
12
13
14
15
16      public K getKey() { return key; }
17
18
19
20      public V getValue() { return value; }
21
22  }
23
24
25  // Will be compiled to:
26  // class Entry {
27  //
28  //     private Comparable key;
29  //     private Serializable value;
30  //     ...

```

REGULAR EXPRESSIONS

```
8 Pattern pattern1 = Pattern.compile("[x-z]+");
9 // Search will symbols from x to z (inclusive)
10 // Search will be only in lowercase characters
11 // To make search case insensitive use Pattern.CASE_INSENSITIVE
12
13 Matcher matcher1 = pattern1.matcher( input: "x y z 1 2 3 4 ");
14 System.out.println(matcher1.find()); // true
15
16 Matcher matcher2 = pattern1.matcher( input: "X Y Z 1 2 3 4");
17 System.out.println(matcher2.find()); //false
18
19 Pattern pattern2 = Pattern.compile("[a-zA-Z0-9]");
20 // Search all lowercase/uppercase chars + numbers
21
22 Matcher matcher3 = pattern2.matcher( input: "A B C D X Y Z " +
23 | "a b c d x y z 1 2 3 4");
24 System.out.println(matcher3.find()); //true
```

.	Соответствие одиночному символу
^regex	Поиск регулярного выражения с совпадением в начале строки
regex\$	Поиск регулярного выражения с совпадением в конце строки
[abc]	Поиск любого символа, заключенного в квадратные скобки
[abc] [vz]	Находит значение символа a, b или c, за которыми следуют v или z
[^ xyz]	Когда символ располагается перед остальными символами в квадратных скобках, он «отрицает» шаблон. Данный шаблон соответствует любому символу, кроме x, y или z.
[a-d1-7]	Диапазоны: соответствует букве между a и d и цифрами от 1 до 7, но не d-1.
X Z	Находит X или Z
\$	Конец строки
^	Начало строки
(re)	Создает группу из регулярных выражений, запоминая текст для сравнения
(?: re)	Действует как (re), но не запоминает текст

Regex	Значение
<code>\d</code>	Любая цифра (эквивалентно <code>[0-9]</code>)
<code>\D</code>	Любой символ, кроме цифер
<code>\s</code>	Символ пробела, сокращение от <code>[\t \n \x0b \r \f]</code>
<code>\S</code>	Любой символ, кроме пробела.
<code>\w</code>	Символы, соответствующие словам, сокращение от <code>[a-zA-Z_0-9]</code>
<code>\W</code>	Символы, не образующие слов, сокращение <code>[\w]</code>
<code>\b</code>	Соответствует границе слова, где символом слова является <code>[a-zA-Z0-9_]</code>
<code>\B</code>	Соответствует границам символов, не являющихся словами
<code>\G</code>	Точка предыдущего соответствия

Regex	Значение	Использование
*	Происходит ноль или более раз, сокращенно {0,}	X* не находит ни одной или нескольких букв X, <sbr/>.* Находит любую последовательность символов.
+	Происходит один или несколько раз, сокращенно {1,}	X+ Находит одну или несколько букв X
?	Не происходит или происходит один раз,? является сокращением для {0,1}.	X? не находит ни одной буквы X или только одну.
{X}	Происходит X раз	\d{3} ищет три цифры.
{X,Y}	Происходит не менее X, но не более Y раз	\d{1,4} означает, что \d должно встречаться как минимум один раз и максимум четыре
*?	? после квантификатора делает его ленивым квантификатором. Он пытается найти наименьшее совпадение. Это останавливает регулярное выражение при первом совпадении.	

```
7 public static void main(String[] args) {  
8     // Greedy quantifier  
9     Matcher matcher = Pattern.compile("g+g").matcher( input: "ggg");  
10    while (matcher.find()) {  
11        System.out.println("Pattern found from " + matcher.start() +  
12            " to " + (matcher.end() - 1));  
13    }  
14    // Pattern found from 0 to 2  
15  
16    // Possessive quantifier  
17    matcher = Pattern.compile("g++g").matcher( input: "ggg");  
18    while (matcher.find()) {  
19        System.out.println("Pattern found from " + matcher.start() +  
20            " to " + (matcher.end() - 1));  
21    }  
22    // No output  
23  
24    // Reluctant quantifier  
25    matcher = Pattern.compile("g+?").matcher( input: "ggg");  
26    while (matcher.find()) {  
27        System.out.println("Pattern found from " + matcher.start() +  
28            " to " + (matcher.end() - 1));  
29    }  
30    // Pattern found from 0 to 0  
31    // Pattern found from 1 to 1  
32    // Pattern found from 2 to 2  
33 }
```

B.Goetz, T.Peierls, J.Bloch, ... - Java Concurrency in Practice (2006)

See you next time



Thank you!