|  |
| --- |
|  |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | **Class** | **Package** | **Subclass(same pkg)** | **Subclass (diff pkg)** | **World** | | **public** | **+** | **+** | **+** | **+** | **+** | | **protected** | **+** | **+** | **+** | **+** |  | | **no modifier** | **+** | **+** | **+** |  |  | | **private** | **+** |  |  |  |  | |
| |  | | --- | | **Dziedziczenie, override, super** | | **class** Pracownik {  **int** intVal1 = 1;  **int** intVal2 = 2;  **public** **int** DajIntVal1() {  **return** intVal1;  }  **public** **int** DajIntVal2Orig() {  **return** intVal1;  }  }  **class** Pracownik2 **extends** Pracownik {  **int** intVal1 = 21; // definicja zmiannej o tej samej nazwie przykrywa poprzenią zmianną  **int** intVal22 = 22;  **public** **int** DajIntVal1() {  **return** intVal1;  }  **public** **int** DajIntVal2() {  **return** intVal2;  }  **public** **int** DajSuperIntVal() {  **return** **super**.DajIntVal1();  }  **public** **int** DajIntVal22() {  **return** intVal22;  }  }   |  |  | | --- | --- | | P2.DajIntVal1()  P2.DajIntVal2()  P2.DajIntVal2Orig()  P2.DajSuperIntVal()  P2.DajIntVal22() | 21  2  1  1  22 | |  |  | | --- | | **Date and Time** | | |  |  | | --- | --- | | java.util.Date  java.sql.Date  java.text.SimpleDateFormat  java.util.Calendar  java.util.TimeZone  java.util.Locale  java.time.Instant  java.time.LocalDate  java.time.LocalTime  java.time.LocalDateTime  java.time.ZonedDateTime  java.time.ZoneId  java.time.ZoneOffset  java.time.Perioid  java.time.Duration  Local[Date][Time]  Konwersja | **data i czas w milisekundach – najstarszy format – DEPRECATED (D), NOT THREAD SAFE (NTS)**  **data i czas dla SQL – D/NTS**  java.sql.Date sqlDate = **new** java.sql.~~Date~~(utilDateNow.getTime());// Date -> SQL  java.util.Date utilDate = sqlDate; // SQL -> Date  **konwersja Date <-> String**  **data i czas ze strefą i lokalizacją – D/NTS**  **strefa czasowa**  **lokalizacja**  Calendar calendar1 = Calendar.*getInstance*(); // pobranie czasu now  Calendar calendar2 = Calendar.*getInstance*(TimeZone.getTimeZone("Poland")); // pobranie czasu ze strefą czsową  **int** year = calendar1.get(Calendar.***YEAR***); // pobranie roku z kalendarza  Date dateFromCalendar = calendar1.getTime(); // Calendar -> Date  Calendar calendar2.setTime(dateFromCalendar); // Date -> Calendar  Calendar calendar.add(Calendar.***HOUR***, 50); // Dodanie 50 godzin  **Czas w nanosekundach od EPOCH - 1970-01-01T00:00:00Z**  **UWAGA!!! Instant.now() oddaje czas dla UTC+00:00,** czyli jak w Polsce many 17.15 to instant odda 16.15  LocalDateTime ldt = LocalDateTime.*ofInstant*(timestamp, ZoneId.systemDefault());  **long** secondsFromEpoch = Instant.*ofEpochSecond*(0L).until(Instant.*now*(), ChronoUnit.SECONDS);  Instant oneHourLater = Instant.*now*().plusHours(1);  **Data, cześć czasu 0.00:000 – TS; \* ozbacza istnienie odpowiedników dala wszystkich Local[XXX]**  LocalDate ld = LocalDate.now();  \* ld = LocalDate.of(2015, 02, 20); // pobranie daty z liczb  \* ld = LocalDate.parse("2015-02-20"); // pobranie daty ze stringa  \* tomorrow = ld.plusDays(1); // dodanie iluś dni  \* tomorrow = ld.plus(1L, ChronoUnit.***DAYS***); // dodanie iluś jednostek czasu  int twelve = ld.getDayOfMonth(); // pobranie miesiąca z daty  boolean leapYear = ld.isLeapYear(); // czy rok przestępny  \* boolean isAfter = ld.isAfter(LocalDate.parse("2016-06-11")); // pobranie data po zadanym dniu  **Czas – TS**LocalTime sevenThirty = LocalTime.*parse*("06:30").plus(1, ChronoUnit.HOURS);  LocalTime maxTime = LocalTime.***MAX***; //23:59:59.99  **Data i czas** – TS  DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("e yyyy-MM-dd HH:mm:ss");  LocalDateTime ldt = LocalDateTime.*parse*("3 1969-07-16 09:32:11", formatter); // pobranie daty ze stringa w zadanym formacie  **Data i czas ze strefą**  ZonedDateTime zdt = ZonedDateTime.*parse*("2015-05-03T10:15:30+01:00[Europe/Paris]");  **Strefa czasowa**  ZonedDateTime zdt = ZonedDateTime.*of*(ldt, zoneId);  **Offset strefy czasowej od UTC**  ZoneOffset offset = ZoneOffset.*of*("+02:00"); //UTC+2:00  OffsetDateTime offSetByTwo = OffsetDateTime.*of*(ldt, offset); //ldt z offsetem +2:00  Reprezentuje ilość czasu dla dat w róznych jednostkach (lata, dni)  **int** days = Period.*ofDays*(5);  LocalDate finalDate = ldt.plus(days);  **int** five = Period.*between*(ldt, finalDate).getDays();  Reprezentuje ilość czasu dla czasu w róznych jednostkach (godziny, minuty)  String localDateString = ldt.format(DateTimeFormatter.***ISO\_DATE***);  Konwersja z formatu bez strefy do formatu ze strefą pobierze czas jako w danej strefie, czyli jeśli mamy w time 14:00 i damy z polską strefą to dostaniemy 12.00+02:00  LocalDateTime.*ofInstant*(date.toInstant(), ZoneId.*systemDefault*()); //dla Instant 12:00 da 13:00  LocalDateTime.*ofInstant*(calendar.toInstant(), ZoneId.*systemDefault*()); | |  |  | | |  |  |  |  | | --- | --- | | **PROSTY ENUM** | **ENUM Z WARTOŚCIAMI** | | **public** **enum** MealEnum1  {  PIZZA,  HAMBURGER;  }  toString(): PIZZA  name(): PIZZA  ordinal: 0; | **public** **enum** MealEnum3  {  PIZZA("P"),  HAMBURGER("H");    **private** String value;    **public** String getValue() {  **return** value;  }  MealEnum3(String val){  value = val;  }  }  getValue(): „P” |  |  | | --- | | **ARRAYS** | | String[] stArr1 = **new** String[2];  Arrays.fill(stArr1, "c");  String[] stArr2 = {"a", "b"};  List<String> stLst2 = Arrays.asList(**new** String[]{"a", "b"}); //!! Rozmiar listy fixed !!  Stream.of(stArr1). …  Arrays.equals(x, y) // Tylko dla one dimension! True gdy dokładnie takie same elementy. Dla obiektów sprawdza adresy.  Arrays.deepEquals(x, y) // Także multi dimension, bo sprawdza w głąb.  Arrays.stream(x [, startInclusive[, endExclusive]]). //abcde: 2,2=’’; 2,4=’cd’; liczone od ‘0’  Arrays.stream(x , start, x.length)  Arrays.stream(intArr1).forEach(a -> System.out.print(a+",")); //intArr1 to obiekty Integer  BinaryOperator<Integer> op1 = (a, b) -> a + b + 1;  Arrays.parallelPrefix(intArr1, 2, 5, op1); //1 2 3 4 5 6 → 1 2 3 8 14 6 |  |  |  | | --- | --- | | **LITERAŁY** | | | Liczby pomijają „\_” rozdzielący tysiące  int/Int – domyślny – brak litery  Long – dla Long konieczne l/L  long - dla long niekonieczne  Float – dla Float konieczne f/F  float - dla float niekonieczne  double/Double – domyślny – możliwe d/D,  \b → \u0008: backspace BS  \t → \u0009: horizontal tab HT  \n → \u000a: linefeed LF  \f → \u000c: form feed FF  \r → \u000d: carriage return CR  \" → \u0022: double quote "  \' → \u0027: single quote '  \\ → \u005c: backslash \ | 1\_1 → 11, 12\_344 → 12344  2, 0372, 0xDada\_Cafe  0L, 0777L, 0xC0B0L, ~~123~~  jw. + 123  1e1f, 2.f, .3f, 0f, 3.14f, 6.027e+3f3, ~~0.123~~  jw. + 0.123  1e1, 2., .3, 0.0, 1D, 3.14, 1e-9d, 1e137 | |
| |  | | --- | | **toString()** | | Obiekty liczbowe = typy proste  **int** x = 3; → ‘3’  **double**/**float** x = 3; → ‘3.0’  **double**/**float** x = 3.00; → ‘3.0’  **float** 3.0000001; → ‘3.0000001’  **float** 3.00000001; → ‘3.0’  **int**[] a1 = **new** **int**[2]; → [I@641856  Integer[] a2 = **new** Integer[2]; → [Ljava.lang.Integer;@1b58ff9e  Object x = **null**; → ‘**null**’  TestClass t1 = **new** TestClass(); → [**package**].ToStringTest$TestClass@641856  @Override  **public** String toString() {  **return** "...”;  } |  |  | | --- | | **Annotacje** | | Annotacje służą:   * informaowaniu compilera o błędach i ostrzeżeniach * manipulowaniu kodem podczas kompilacji * modyfikacji i sprawdzania zachowania w runtime  |  |  | | --- | --- | | @Override  @Deprecated  @SuppressWarnings([‘x’/{‘x’,’y’}])  @SafeVarargs  @FunctionalInterface  ***Meta annotations***  @Target  @Retention  @Inherited  @Repetable  ***Spring annotations***  ***Custom annotation*** | Wymusza sprawdzenie, czy to faktycznie w dziedziczonej klasie jest taka funcja. Blokuje kompilację. Nie jest niezbędna, ale zabezpiecza.  Oznacza kod jako deprecated. Używane przez kompilator do warningów i w api  Wyłącza sprawdzanie pewnych typów warningów, np: "unchecked", "deprecated". Lista zależy od wersji. Ide obsługuje zazwyczaj więcej opcji: „**javac –X”**  Pozwala ignorować bwarnini związane z Varargs- coś jak @SuppressWarnings  Dodane nad interface polnuje aby była tylko jedna metoda (bo tak działają functional interfaces). Działa podobnie jak @Override, czyli zabezpiecza.  **Używanye w definicjach custom annotation.** <https://www.baeldung.com/java-default-annotations>  Wskazyje typy z jakimi annotaja ma być dostępna. @Target({ElementType.*CONSTRUCTOR*})  Kiedy annotacja ma być widoczna.   * *RetentionPolicy.SOURCE –*visible by neither the compiler nor the runtime * *RetentionPolicy.CLASS* – visible by the compiler * *RetentionPolicy.RUNTIME –*visible by the compiler and the runtime   Podowuje, że annotacja dołaczona do klasy bazowej działą także w klasie dziedziczącej  Pozwala na wielokrotne użycie jednej annotacji nad tym samym elementem.  **public** **@interface** Schedules {Schedule[] value();}  @Repeatable(Schedules.**class**) **public** **@interface** Schedule {String time() **default** "09:00";}  @Schedules({@Schedule(), @Schedule(time = "23:00")}) **void** alarm() {...}  <https://www.journaldev.com/16966/spring-annotations>  -> @Repetable; Kod Runtime używa reflekcji; https://www.baeldung.com/java-custom-annotation |   l | |  |  |  | | --- | | **TransactionTemplate** | | \* TransactionTemplate TT może przeplatać się z @Transactional (zakładam, ze ob mają ten sam PlatformTransactionManager)  \* W Lambda można wywoływać @Transactional i będą respektować typy PROPAGATION  \* TT samoczynnie wykonuje na koniec Commit lub Rollback  **Wykonuje kod pod transakcją**  @Autowired TransactionTemplate tt;  @Autowired EntityManager em;  …  tt.execute( status → {  em...  **return** **null**; - zwracanie wartości jest konieczne  });  status.setRollbackOnly(); //ustawia rollback  **Customowa inicjalizacji TransactionTemplate**  *\* PlatformTransactionManager wpływa na transakcje, czyli jeśli będą różne dla różnych TT, to mogą być problemy z transakcjami działającymi równocześnie (REQUIRES\_NEW)*  @Autowired **private** PlatformTransactionManager tm;  …  transactionTemplate = **new** TransactionTemplate(tm);  transactionTemplate.setIsolationLevel(TransactionDefinition.***ISOLATION\_REPEATABLE\_READ***);  transactionTemplate.setPropagationBehavior(TransactionDefinition.***PROPAGATION\_REQUIRES\_NEW***);  transactionTemplate.setTimeout(1000);  transactionTemplate.setReadOnly(**true**); //ustawia optymalizację do transakcji read-only  **Commit / Rollback**  **\* Po wymuszeniu Commit lub Rollback nie można kontynuować wlewania w zakończonej transakcji**  @Autowired **private** PlatformTransactionManager tm;  …  status → {…  tm.rollback(status);  } |  |  | | --- | | **Generics** | | **Postawy.**  *T rozszerza String i implementuje Iterator*  **public** **static** <T **extends** String{klasa} & Iterator{**interface**}, W> **void** GenericTest1(T t, W w) {...}  ***? - wildcard***  *W Java Każda klasa (X) pochodzi od Object, ale List<X> NIE POCHODZI od List<Object>, dlatego za <T extends List<Object>> NIE MOŻNA podstawić np. List<String>. Można to pominąć używając „?”*  **public** **void** genericTest2(List<? **extends** Object> t) {...} |  |  | | --- | | **Interfejsy funkcyjne (IF)** | | java.util.function.\*  **Definiowanie funkcji**  Function<Integer, Integer> funct = i -> i++;  **Używanie FI**  **public** **void** funcIntTest(Function<Integer, Integer> f, Integer a, Integer b) {  b = f.apply(a); //Każdy interface ma własną funkcję, „Function” ma „apply”  }  ...  funcIntTest(func, 1, 2);  funcIntTest(i -> i++, 1, 2);  **Implementacja FI**  **class** ImplPredicateInteger **implements** Predicate<Integer>{  @Override  **public** **boolean** test(Integer t) {...}  }  **class** ImplPredicateT<T> **implements** Predicate<T>{  @Override  **public** **boolean** test(T arg0) {...}  }  **FI dostarczone z Java**   |  |  | | --- | --- | | Function<T,R> | R=apply(T) - przyjmyje obiekt T i Oddaje R | | BiFunction<T,R,U> | jak Function, ale przyjmyje 2 obiekty | | UnaryOperator<T> | jak function, ale T = R | | BinaryOperator<T> | jak BiFunction, ale T = R = U | | Predicate<T> | *bool*=test(T) – przyjmuje obiekt T i oddaje *boolean* | | Supplier<T> | T=get() - nie przyjmuje, ale oddaje, taki generator | | Consumer<T> | void=accept(T) – przyjmuje i coś robi, ale nic nie oddaje |   **Tworzenie nowego FI**  @FunctionalInterface  **public** **interface** CustomFunctionalInterface<T> {  **public** T change(T t);  }  @FunctionalInterface  **public** **interface** CustomFunctionalInterface<T> {  **public** T change(T t);  }  **Użycie**  CustomFunctionalInterface ff ↔ CustomFunctionalInterface<Object> ff  CustomFunctionalInterface<String> ff | |
| |  | | --- | | **Validation** | | * 'javax.validation:validation-api:2.0.0.Final' * 'org.springframework.boot:spring-boot-starter-actuator' – konieczny dla podpowiedzi w Ecplipse * implementacja javax.validation:validation-api. Brak powoduje: *javax.validation.NoProviderFoundException: Unable to create a Configuration, because no Bean Validation provider could be found. Add a provider like Hibernate Validator (RI) to your classpath.*   'org.hibernate.validator:hibernate-validator:6.0.2.Final'  *'org.hibernate.validator:hibernate-validator-annotation-processor:6.0.2.Final'*   * implementacja SpEL. Brak powoduje: HV000183: Unable to load ‘javax.el.ExpressionFactory'. Check that you have the EL dependencies on the classpath, or use ParameterMessageInterpolator instead   'javax.el:javax.el-api:3.0.0'  'org.glassfish.web:javax.el:2.2.6'   * @[Not]Null()   @Assert[True/False]() - {nullOk} - dla **boolean**  @Size() - {nullOk} - liczba elementów. Ustaewienie tylko max=2 oznacza (0,2). Dla pustych param zawsze spełnia: **stringi** i **kolekcje**  @NotEmpty() – jw, ale sprawdza czy zero  @[Min/Max](x) - {nullOk} - większ od min / mniejsze od max. **TYLKO liczby całkowite**  @Decimal[Min/Max] - {nullOk} - jw, ale pozwala na sprawdzanie **stringowej** reprezentacji. Inclusive zawiaduje < czy <=  @NotBlank – nie null i przynajmniej jeden niebiały znak: **stringi**  @Positive[OrZero] / @Negative[OrZero] - x>=0 / x<=0: **liczby całkowite** i **ułamki**  @Past[OrPresent] / @Future[OrPresent] - {nullOk} - jw, ale dla **date**/**time**  @Email() - czy poprawny adres email   * **Używanie**   ValidatorFactory factory = Validation.buildDefaultValidatorFactory();  Validator validator = factory.getValidator();  Set<ConstraintViolation< **class** >> res = validator.validate( object );  **if** (res.size() > 0) {  **for** (ConstraintViolation<ValidatedObj> violation : res) {  violation.getMessage();  //getPropertyPath() - nazwa sprawdzanego obiektu  //Value – wartość obiektu  }  }   * Sprawdzać można obiekty i obiekty proste * Wszystkie powyższe jako parametry przyjmuje „message”, „groups” i „payload” * Jesli typ nieodpowiedni do walidacji to i tak się skompilue, ale na walidacę wyskoczy: „*javax.validation.UnexpectedTypeException: HV000030: No validator could be found for constraint 'javax.validation.constraints.Email' validating type 'java.lang.Integer'. Check configuration for 'i5'*” * Groups – grupuje constainty i pozwala spawdzać tylko constrainty z wybranych grup. Grypy wyznaczany przez interface   **public** **interface** ValGroup1 {**pusty**}  ...  @Max(value = 2, groups = ValGroup1.**class**)  @Email(groups = {ValGroup1.**class**, ValGroup2.**class**})  ...  Set<ConstraintViolation<*objectClass*>> res = validator.validate(*object*, ValGroup1.**class**, ValGroup2.**class**);   * Payload – zaawansowana obsługa constraintów, np. dodatkowe akcje (jakiś println) lub po prosty jako znacznik i sprawdzenie czy dany violation „zawiera znacznik” <https://www.logicbig.com/how-to/code-snippets/jcode-bean-validation-javax-validation-payload.html> * Custom constraints - <https://docs.jboss.org/hibernate/validator/4.1/reference/en-US/html/validator-customconstraints.html#validator-customconstraints-constraintannotation> | |