javafx.concurrent

Class Task<V>

java.lang.Object
    java.util.concurrent.FutureTask<V>
        javafx.concurrent.Task<V>

All Implemented Interfaces:

java.lang.Runnable, java.util.concurrent.Future<V>, java.util.concurrent.RunnableFuture<V>, Worker<V>, EventTarget

```
public abstract class Task<V>
extends java.util.concurrent.FutureTask<V>
implements Worker<V>, EventTarget
```

A fully observable implementation of a `FutureTask`. Tasks exposes additional state and observable properties useful for programming asynchronous tasks in JavaFX, as defined in the `Worker` interface. An implementation of Task must override the `call()` method. This method is invoked on the background thread. Any state which is used in this method must be safe to read and write from a background thread. For example, manipulating a live scene graph from this method is unsafe and will result in runtime exceptions.

Tasks are flexible and extremely useful for the encapsulation of "work". Because `Service` is designed to execute a Task, any Tasks defined by the application or library code can easily be used with a Service. Likewise, since Task extends from FutureTask, it is very easy and natural to use a Task with the java concurrency `Executor` API. Since a Task is Runnable, you can also call it directly (by invoking the `FutureTask.run()` method) from another background thread. This allows for composition of work, or pass it to a new Thread constructed and executed manually. Finally, since you can manually create a new Thread, passing it a Runnable, it is possible to use the following idiom:

```
         Thread th = new Thread(task);
         th.setDaemon(true);
         th.start();
```

Note that this code sets the daemon flag of the Thread to true. If you want a background thread to prevent the VM from existing after the last stage is closed, then you would want daemon to be false. However, if you want the background threads to simply terminate after all the stages are closed, then you must set daemon to true.

Although `ExecutorService` defines several methods which take a Runnable, you should generally limit yourself to using the `execute` method inherited from `Executor`.

As with FutureTask, a Task is a one-shot class and cannot be reused. See `Service` for a reusable `Worker`.

Because the Task is designed for use with JavaFX GUI applications, it ensures that every change to its public properties, as well as change notifications for state, errors, and for event handlers, all occur on the main JavaFX application thread. Accessing these properties from a background thread (including the `call()` method) will result in runtime exceptions being raised.

It is **strongly encouraged** that all Tasks be initialized with immutable state upon which the Task will operate. This should be done by providing a Task constructor which takes the parameters necessary for execution of the Task. Immutable state makes it easy and safe to use from any thread and ensures correctness in the presence of multiple threads.

In Java there is no reliable way to "kill" a thread in process. However, when `cancel` is called on a Task, it is important that the Task stop processing. A "run-away" Task might continue processing and updating the message, text, and progress properties even after the Task has been cancelled! In Java, cancelling a Task is a cooperative endeavor. The user of the Task will request that it be cancelled, and the author of the Task must check whether is has been cancelled within the body of the `call` method. There are two ways this can be done. First, the Task author may check the isCancelled method, inherited from `FutureTask`, to see whether the Task has been cancelled. Second, if the Task implementation makes use of any blocking calls (such as NIO InterruptibleChannels or Thread.sleep) and the task is cancelled while in such a blocking call, an InterruptedException is thrown. Task implementations which have blocking calls should recognize that an interrupted thread may be the signal for a cancelled task and should double check the isCancelled method to ensure that the InterruptedException was thrown due to the cancellation of the Task.

Examples

The following set of examples demonstrate some of the most common uses of Tasks.

A Simple Loop

The first example is a simple loop that does nothing particularly useful, but demonstrates the fundamental aspects of writing a Task correctly. This example will simply loop and print to standard out on each loop iteration. When it completes, it returns the number of times it iterated.

```
     Task<Integer> task = new Task<Integer>() {
         @Override protected Integer call() throws Exception {
             int iterations;
             for (iterations = 0; iterations < 100000; iterations++) {
                 if (isCancelled()) {
                     break;
                 }
                 System.out.println("Iteration " + iterations);
             }
             return iterations;
         }
     };
```

First, we define what type of value is returned from this Task. In this case, we want to return the number of times we iterated, so we will specify the Task to be of type Integer by using generics. Then, within the implementation of the `call` method, we iterate from 0 to 100000. On each iteration, we check to see whether this Task has been cancelled. If it has been, then we break out of the loop and return the number of times we iterated. Otherwise a message is printed to the console and the iteration count increased and we continue looping.

Checking for isCancelled() in the loop body is critical, otherwise the developer may cancel the task, but the task will continue running and updating both the progress and returning the incorrect result from the end of the `call` method. A correct implementation of a Task will always check for cancellation.

A Simple Loop With Progress Notification

Similar to the previous example, except this time we will modify the progress of the Task in each iteration. Note that we have a choice to make in the case of cancellation. Do we want to set the progress back to -1 (indeterminate) when the Task is cancelled, or do we want to leave the progress where it was at? In this case, lets leave the progress alone and only update the message on cancellation, though updating the progress after cancellation is a perfectly valid choice.

```
     Task<Integer> task = new Task<Integer>() {
         @Override protected Integer call() throws Exception {
             int iterations;
             for (iterations = 0; iterations < 10000000; iterations++) {
                 if (isCancelled()) {
                     updateMessage("Cancelled");
                     break;
                 }
                 updateMessage("Iteration " + iterations);
                 updateProgress(iterations, 10000000);
             }
             return iterations;
         }
     };
```

As before, within the for loop we check whether the Task has been cancelled. If it has been cancelled, we will update the Task's message to indicate that it has been cancelled, and then break as before. If the Task has not been cancelled, then we will update its message to indicate the current iteration and then update the progress to indicate the current progress.

A Simple Loop With Progress Notification And Blocking Calls

This example adds to the previous examples a blocking call. Because a blocking call may thrown an InterruptedException, and because an InterruptedException may occur as a result of the Task being cancelled, we need to be sure to handle the InterruptedException and check on the cancel state.

```
    1. W Javie Stałe pole statyczne możemy zdefiniować dla: Klasy abstrakcyjnej: TAK Klasy konkretnej: TAK Interfejsu: TAK

2. Napisz funkcję, która robi coś takiego fun(2, 10, -3) → “-3,10,2”      public String fun(int ... args)     {   StringBuilder s = new StringBuilder();   for(int i = args.length - 1; i >= 0; --i)   {   s.append(String.valueOf(args[i]));   if(i == 0) continue;   s.append(", ");   }   return s.toString();   }

3. W Javie 8 adnotację możemy postawić przed deklaracją zmiennej lokalnej w metodzie. Prawda


Pytanie 4 Napisz statyczną metodę addNumber która: - Zwraca wartość integer - Ma parametr Connection conn oraz int liczba - wstawia do tabeli o nazwie  numbers  w polu o nazwie  number  wartość przekazaną w parametrze  value

Wykorzystaj słowa kluczowe: prepareStatement, executeUpdate, SQLException, close, setInt

public static int  addNumber(Connection conn, int value) { SQLException { PreparedStatement ps = null; try  {
ps = conn.prepareStatement("INSERT INTO numbers (number) VALUES (?)"); ps.setInt(1, 5); return  ps.executeUpdate();
}  finally  { if (ps != null) ps.close(); }
}
}

Pytanie 5 Napisz fragment kodu który dla zmiennej Connection o nazwie conn pobiera wszystkie pola tabeli Student. Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery("SELECT * FROM Student;");

Pytanie 6 Jak będzie wyglądać instrukcja do uzyskania obiektu typu Connection jeśli adres to localhost:3306, nazwa użytkownika root, a hasło root: DriverManager.getConnection("localhost:3306", "root", "root");

Pytanie 7 Obiekt typu Class dla tekstowej reprezentacji nazwy “com.mysql.jdbc.Driver” otrzymamy instrukcją: Class.forName(“com.mysql.jdbc.Driver”);

Pytanie 8 Popraw następujący kod, tak aby mógł się skompilować: int[5] arr = new int[5]; arr[5] = 4;

int[] arr = new int[5]; arr[4] = 4;

Pytanie 9 Napisz fragment kodu, który zawiera: - klasę z prywatnym bezparametrowym konstruktorem i publicznym konstruktorem przyjmującym int - uzyskaj dostęp do klasy poprzez nazwę, i wywołaj bezparametrowy konstruktor - uzyskaj dostęp do klasy za pomocą pola “class” i wywołaj konstruktor z parametrem

class X {     private X() { System.out.println("private"); }     public X(int x) { System.out.println("public " + x); }
}

public class Main {     public static void main(String[] args) throws ...     {         Class classByName = Class.forName("X");         Constructor c1 = classByName.getDeclaredConstructor();         c1.setAccessible(true);         c1.newInstance();

        Class classByObject = X.class;         Constructor c2 = classByObject.getConstructor(int.class);         c2.newInstance(6);     } }

Pytanie 10 Napisz fragment kodu, który inicjalizuje Hibernate w języku Java Configuration cfg = new Configuration().configure();

Pytanie 11 Czy  TreeSet jest podtypem  TreeSet ? Odpowiedź:  FAŁSZ

Pytanie 12 Jaki interfejs implementuje sterownik JDBC (Java Database Connectivity) Odpowiedź:  java.sql.Driver

Pytanie 13 Jaką metodą wybudza się wszystkie uśpione wątki? Odpowiedź:  notifyAll

Pytanie 14  W jaki sposób adnotujemy w JPA pole encji, które bedzie automatycznie numerowane? (pamiętaj o znaku@)  Odpowiedź:  @GeneratedValue(strategy = GenerationType.IDENTITY)

Pytanie 15  W języku Java enkapsulacja oznacza, że obiekt może zachowywać się w zależności od tego, w jakim kontekście został użyty. Odpowiedź:  FAŁSZ

Pytanie 16 W poniższym kodzie sum.java są dokładnie 3 pluskwy(logiczne, czasu wykonania lub kompilacji).
Zaznacz, w których liniach należy poprawić kod, aby błąd nie wystąpił, a program wypisał sumę tablicy a. Jeśli nie jesteś pewien, nie zaznaczaj. Nie ma punktów ujemnych, ale za każdą pomyłkę możesz stracić 33% punktacji.

public class Sum { Sum() { long [] a = {5L, -10, 20, 30}; // brak inicjalizacji zmiennej arr_sum long arr_sum; // a.length() to pole, a nie metoda powinno być a.length for ( int i = 0; i < a. length (); i++) arr_sum += a[i]; // nie istnieje println(String, long) // ‘+’ zamiast ‘,’ albo StringBuilder albo String.format System.out.println ("Sum of als elements =", arr_sum); }  public static void

Pytanie 17 Metodą domyślną w interfejsie Javy 8 poprzedzamy słowem default. Odpowiedź:  PRAWDA

Pytanie 18 Czy Long[] jest podtypem Object[]? Odpowiedź:  PRAWDA

Pytanie 19 Przyporządkuj nazwy strumieni wyjściowych do odpowiednich kategorii Każdy użyj tylko raz, pewne mogą pozostać niewykorzystane. Zapis sformatowanej linii do pliku tekstowego - PrintWriter Strumień znakowy - FileWriter Binarny zapis liczb np. float - DataOutputStream --------------- Binarny odczyt liczb float: DataInputStream Strumień bajtów: ? Strumień znaków: FileWriter ------- Konwe

 InputStreamReader : byte  ->char OutputStreamWriter: char -> byte


Pytanie 20 Instrukcja super(...) w konstruktorze może się pojawić wyłącznie w pierwszej jego linii. Odpowiedź:  PRAWDA

Pytanie 21 Hermetyzacja w Javie może być zrealizowana poprzez prywatne składowe klasy i zastosowanie funkcji dostępowych Odpowiedź: PRAWDA

Pytanie 22 Jaki napis wypisze poniższy fragment kodu?

Odpowiedź:  y

Pytanie 23 W Javie odwołanie do ostatniego elementu ArrayListy li to Odpowiedź:  li.get(li.size()-1)


Pytanie 24 Dla danej tablicy (tutaj tab) utwórz strumień Javy 8 zawierający tylko długości stringów non-null i niepustych.
```

Here we have added to the body of the loop a `Thread.sleep` call. Since this is a blocking call, I have to handle the potential InterruptedException. Within the catch block, I will check whether the Task has been cancelled, and if so, update the message accordingly and break out of the loop.

A Task Which Takes Parameters

25. Operacje terminalne na strumieniach Javy 8 zawsze zwracają void Fałsz " Terminal operations produces a non-stream, result such as primitive value, a collection or no value at all." Przykłady: toArray(), collect(), count(), reduce(), forEach(), forEachOrdered(), min(), max(), anyMatch(), allMatch(), noneMatch(), findAny(), findFirst() Operacje pośrednie (intermediate) z kolei zwracają strumień 26. W języku Java, jeden plik źródłowy może zawierać co najwyżej jeden publiczny interfejs. Odpowiedź: Prawda 27. W Javie Stałe pola statyczne możemy zdefiniować dla: Klasy abstrakcyjnej: Tak Klasy konkretnej: Tak Interfejsu: Tak 28. Java 8 wspiera wielodziedziczenie typów (dopuszcza następujący kod): class C extends A, B { /* */} Odpowiedź: Fałsz Tylko wiele interfejsów możemy rozszerzać 29. Jaką metodą wybudza się jeden uśpiony wątek? Odpowiedź: notify 31. Napisz funkcję przyjmującą jako argument tablicę stringów (zakładamy że argument ten jest != null oraz ma długość >0) i zwracającą string posiadający co najmniej 2 wystąpienia Jeśli takiego nie ma, to niech funkcja zwraca null. Jeśli więcej niż jeden string ma >= wystąpienia, to niech funkcja zwróci dowolny z nich. Przykłady: -argumentem jest tablica {"kot","pies","mysz","pies"}, funkcja zwraca "pies", -argumentem jest tablica {"kot","pies","mysz","koza"}, funkcja zwraca null. Odpowiedź: public String func(String[] tab) { LinkedList list = new LinkedList<>(); for (String s:tab) { if (list.contains(s)) { return s; } else list.add(s); } return null; } 32. Całkowite typy prymitywne Javy to: char, int, short, long, byte Dodatkowo zmiennoprzecinkowe, to: float, double jeszcze jest ……………………………: boolean 33. W języku java jeden plik źródłowy może zawierać co najwyżej jedną klasę publiczną. prawda 34. W języku java jeden plik źródłowy może zawierać co najwyżej jeden publiczny interfejs. prawda 35. Przyporządkuj nazwy strumieni wejściowych do odpowiednich kategorii. Każdy użyj tylko raz, pewne mogą pozostać niewykorzystane Binarny odczyt liczb float: DataInputStream Strumień bajtów: ? Strumień znaków: ? 36. Jeżeli w definicji typu adnotacyjnego umieścimy metaadnotację @Retention(RetentionPolicy.RUNTIME) oznaczać to, że będzie ona przetwarzana dopiero w czasie wykonania kodu Odpowiedź: Prawda 38. W Javie Klasę wewnętrzną możemy zdefiniować dla: Klasy konkretnej: Tak Interfejsu: Nie Klasy abstrakcyjnej: Tak 39. Napisz w Javie kod serwera dla aplikacji Klient Serwer działającej na Socketach. Serwer powinien przyłączyć klienta, wysłać klientowi pewien String, rozłączyć go i zakończyć działanie. Wykorzystaj słowa kluczowe: accept, close, flush, getOutputStream, IOexception, java.io, java.net, println, PrintWriter, ServerSocket, Socket Odpowiedź: //kod ze zdj, ocenione na 3 z 3 try{ // utworzenie gniazda String serverHost = …; int serverPort = …; Socket socket = new Socket(serverHost, serverPort) OutputStream sockOut = socket.getOutputStream(); InputStream sockIn = socket.getInputStream(); socktOut.write(…); socktIn.read(); sockOut.close(); sockIn.close(); socket.close(); } catch(UnknownHostException exc) { // nieznany host } catch(SocketException exc) { // wyjątki związane z komunikacją przez gniazda } catch(IOException exc) { // inne wyjątki we/wy } 40. Napisz funkcję przyjmującą tablicę stringów (zakładamy że ten argument != null i ma długość > 0) i zwracającą string który występuje w tej tablicy co najmniej dwa razy. Jeśli takiego nie ma zwróć null public String foo(String args[]) { HashSet set = new HashSet<>(); for(String s : args) if(set.contains(s)) return s; else set.add(s); return null; } 41. Słowo kluczowe "this" może być użyte w ciele konstruktora. Odpowiedź: Prawda 42. Jaki napis wypisze poniższy fragment kodu? Integer i = 0; System.out.println(i != null ? 1<<2 : 6 >> 1); Odpowiedź: 4 Objaśnienie: 0b100=2^2 = 4 0b1000=2^3 = 8 0b10000=2^4 = 16 44. Słowa kluczowe używane przy wyjątkach to: try, catch, ***, throw, *** Odpowiedź: throws, finally 46. Można stworzyć konstruktor klasy abstrakcyjnej. Odpowiedź: Prawda 47. Obiekt w Javie może dziedziczyć stan z kilku KLAS bazowych. Odpowiedź: Fałsz 48. Przyporządkuj nazwy strumieni wejściowych do odpowiednich kategorii. Każdy użyj tylko raz, pewne mogą pozostać niewykorzystane. Konwersja char - byte: OutputStreamWriter -- do sprawdzenia!! Strumień obiektów, np Date: ObjectInputStream Odczyt linii pliku tekstowego: BufferedReader 49. Metodę domyślną interfejsu w javie oznaczamy słowem Odpowiedź: default Dodatkowe info Java Jeżeli w definicji typu adnotacyjnego umieścimy metadnotację @Retention(RetentionPolicy.RUNTIME) oznacza to że będzie ona przetwarzana dopiero w momencie wykonywania kodu W języku java plik źródłowy musi mieć tę samą nazwę co zawarta w nim klasa publiczna, natomiast klasa niepubliczna może mieć inną. W javie 8 adnotacją możemy opatrzeć argument metody W javie klazulę extends możemy umieścić w nagłówku dla: klasy konkretnej, klasy abstrakcyjnej, interfejsu. Strumienie wejsciowe: Binarny odczyt liczby float: DataInputStream Strumień bajtów: FileInputStream Strumien znaków: BufferedReader W javie klasę wewnętrzną można zdefiniować dla: klasy konkretnej, klasy abstrakcyjnej, interfejsu @Override - adnotacja metody, wykorzystywana podczas dziedziczenia, na przykład klasa Object ma metodę hashCode(), i w naszym obiekcie można ją nadpisać. @Deprecated - porzucona funkcjonalność, może się popsuć w kolejnej wersji @Test - przy testach JUnit @SuppressWarnings - ignoruje ostrzeżenia Tablice nie mogą mieć generycznych typów: Pair[] table = new Pair<>[10]; // !!! Collection - kolekcja zawierająca typy implementujące interfejs Andrzej / dziedziczące z klasy Andrzej Klasy strumieni wejścia/wejścia - bazowe klasy abstrakcyjne: InputStream, OutputStream - sekwencje bajtów: FileInputStream, FileOutputStream - sekwencje dowolnych typów danych: DataInputStream, DataOutputStream - obiekty: ObjectInputStream, ObjectOutputStream Klasy operacji na tekście: FileReader, FileWriter Obsługa niestandardowego kodowania: opakowanie FileReader w InputStreamReader Odczyt z konsoli: Scanner Serializacja: implementacja interfejsu Serializable , metody readObject() i writeObject() w obiekcie strumienia Programowanie sieciowe Adresowanie localhosta: InetAddress a0 = InetAddress.getLocalHost(); Host odległy: InetAddress address = InetAddress.getByName(" www.oreilly.com "); Nie uwierzytelniony aplet nie może wykonywać operacji getByName i getAllByName ! Strumienie filtrujące oraz transformujące są przykładem dekoratorów PrintWriter jest właściwą klasą do zapisu plików tekstowych! Klasy InputStreamReader oraz OutputStreamWriter pozwalają na przezroczystą dla programisty konwersję źródłowego strumienia bajtowego na znaki Unicode i na odwrót. Gniazda umożliwiają niskopoziomową komunikację sieciową. Gniazdo to jeden z końców dwukierunkowego połączenia pomiędzy klientem a serwerem. Pakiet datagramowy (zwykle UDP) jest reprezentowany przez klasę DatagramPacket. Gniazda bezpołączeniowe (ang. datagram sockets) umożliwiają przesyłanie i odbieranie datagramów. Najczęściej są to pakiety UDP. Przy tym połączeniu nie rozróżniamy klienta i serwera, ponieważ połączenie nie jest zestawiane (RPC) Remote Procedure Call - Zdalne wywołanie procedur (RMI) Remote Method Invocation - Zdalne wywoływanie metod System RMI pozwala na to, aby obiekt działający na jednej maszynie wirtualnej wywoływał metody obiektu działającego na innej maszynie wirtualnej Javy Protokół RMI zapewnia,że obiekty zdalne mogą być zniszczone dopiero, kiedy nie ma do nich ani zdalnych, ani lokalnych referencji. RMI do przekazywania obiektów przez sieć wykorzystuje mechanizm serializacji. Głównym problemem przy RMI i zarządzaniu pamięcią jest rozproszone zwalnianie pamięci (distributed garbage collection) . Rozwiązaniem tego jest licznik referencji dla każdego obiektu zdalnego. RMI pozwala na szybkie i łatwe tworzenie aplikacji rozproszonych. Ze względu na silny związek z Javą RMI jest w stanie współdziałać z dziedziczeniem. Wywołania RMI są blokujące, jak zwykle w RPC, jednak wbudowana wielowątkowość pozwala na uruchomienie innych funkcji programu w trakcie obliczeń na maszynie zdalnej Bazy danych Rodzaje sterowników JDBC (Java Database Connectivity) 1) mostek JDBC-ODBC 2) mieszany kod rodzimy klienta bazy danych i Javy 3) czysty kod Javy komunikujący się niezależnym od bazy danych protokołem sieciowym, tłumaczonym przez pośredni serwer 4) czysty kod Javy komunikujący się zależnym od bazy danych protokołem sieciowym Metody obiektu typu ResultSet do pobierania wartości pobranych rekordów first() – przejście do pierwszego rekordu last() – przejście do ostatniego rekordu next() – przejście do następnego rekordu previous() – przejście do poprzedniego rekordu getFloat("nazwa kolumny") – pobranie wartości float getInt("nazwa kolumny") – pobranie wartości integer getDate("nazwa kolumny") – pobranie wartości typu Date getString("nazwa kolumny") – pobranie wartości String Zadania Scala 1. Co zwróci następująca konstrukcja (proszę zaznaczyć też typ): for(i <- 0 untril 4 by 2; c <- "ab") yield (c+i).toChar Napisz w scali wyrażenie tworzące zmienną x o wartości -1 jeśli y > 0, "!!!" w przeciwnym wypadku. Jaki będzie typ x? def manOf[T: Manifest](t: T): Manifest[T] = manifest[T] var y = -20 var x = if (y > 0) -1 else "!!!" println(manOf(x)) Typ: Any 2. W języku Scala 1. Wartość(value) b jest ArrayBufferem. Co oznacza kod: b.sortWith(_>_)? 2. Mamy następujące dwie klasy w Scali: class Osoba1 {var wiek = 20} oraz class Osoba2 {val wiek = 20} Czy można ustawić wiek obiektu klasy Osoba1 i Osoba , a jeśli tak to w jakis sposób? Odpowiedź: 1. Oznacza to, że należy posortować wartości malejąco. 2. Nie można ustawić wieku osoby klasy Osoba2 gdyż val jest niemutowalną zmienną. Można natomiast ustawić wiek Osoba1; val os: Osoba1 =new Osoba1(); os.wiek = 30; 3. Co zwróci następująca konstrukcja: for(c<- "abc"; i <- 1 to 2) yield (c+i).toChar Odpowiedź: bccdde 4. Napisz w Scali kod zapisujący w kolekcji listowej (np. Vector) wszystkie pary liczb (i,j), 1<= i, j <= 5 których suma jest nie nieparzysta, Użyj for: Odpowiedź: var e = for(i <- Range(0, 2); j <- Range(0, 5) if (i + j) % 2 == 1) yield (i, j) 5. Jak jest różnica między val a = myArray(10) a val a = new myArray(10) Zakładając że myArray to jakiś typ tablicowy Odpowiedź: Pierwsza wersja: używa metody apply z obiektu towarzyszącego klasy myArray. Drugie wywołuje konstruktor 6. Co zwróci następująca konstrukcja (proszę zaznaczyć też typ): for(i <- 0 until 4 by 2; c <- "ab") yield (c + i).toChar Nie znam scali, więc nie wytłumaczę dlaczego, ale: Vector (a , b , c , d) 7. Podaj po jednym przykładzie kolekcji mutowalnej i niemutowalnej z biblioteki standardowej scali Mutowalne: ● MutableList ● Arraybuffer ● Listbuffer ● Stringbuilder ● Linkedlist ● Double linked list ● mutable.Queue ● Nie chce mi się pisać więcej https://www.scala-lang.org/api/2.12.2/scala/collection/mutable/index.html Niemutowalne: ● List ● Stream ● Vector ● Stack ● Queue ● Range ● Nie chce mi się pisać więcej https://www.scala-lang.org/api/2.12.2/scala/collection/immutable/index.html w Javie List NIE JEST podtypem List. W Kotlinie JEST ! Dodatkowe info Scala - same ciekawostki, lub rzeczy które mogą się przydać val - value - immutable(niezmienny) var - variable - muttable(zmienny) Scala od wersji 5 domyśla się typu. jednak możliwe jest określenie typu odgórnie np: val x:Int = 6 val y: Double = -3.2 Vector - takie samo jak w Javie ArrayList ale "niezmienny" brevity(zwięzłość) ● return może zostać pominięty, wynikiem jest ostatnie wyrażenie złożone -- do sprawdzenia ● średniki mogą zostać pominięte ● "parens"(chyba chodzi o nawiasy ()) w wywołaniach mogą zostać pominięte jeśli metoda ma 0 lub 1 argumentów - tylko w tak zwanej notacji operatora ● tak samo z kropką po nazwie obiektu, a przed metodą - w takim samym przypadku jak wyżej(notacji operatora?) Metody bez parametrów dobrym stylem jest używać () dla mutatorów(getów) i nie używać () dla accesorów(setter) Typy Scali Klasy Scala Any, AnyRef i AnyVal nie pojawiają się jako klasy w kodzie bajtowym - z powodu ograniczeń JVM. (Java Virtual Machine) W Scali wszystko jest obiektem(w javie nie), JVM nie może ogarnąć tego i wygenerowany kod bajtowy nie pokazuje tego. W Scali wszystkie obiekty są podklasą Any. AnyRef w Scala jest odpowiednikiem java.lang. Object (przynajmniej na JVM). AnyVal w Scali są odpowiednikiem typów prymitywnych w javie Typ Nothing nie dziedziczy z żadnego innego typu Lists jak i String są niezmienne(immutable) dostęp do początku/konca z O(1) reszta operacji O(n) Listy są konstruowane od prawej do lewej Indexing: list(0) Slicing: list.slice(1, 3); list.slice(2, list.last) Reversing: list.reverse Sorting: list.sorted => nieformalnie operator rakiety pętla for: for (y <- List(1, 2, 3)) { println(y) } for (x<- List(1,2,3); y<-List(4,5)) yield x * y wynik: List(4, 5, 8, 10, 12, 15) for{ x <- 1 to 7 // generator y = x%2; // definicja? if(y == 0) // filtr } yield { println(x) x } Wynik: 2 4 6 dla pętli for możliwość używania wielu filtrów generowana kolekcja jest kompatibilna z pierwszym podanym generatorem Funkcja operatora(operator function) def ~=(x: Double, y: Double, precision: Double) = { if ((x - y).abs < precision) true else false } wywołanie: ~=(a, b, 0.0001) wynik: true Tablice są zmienne(mutable) Sortowanie tablicy(ale nie arrayBuffer) val a = Array(1, 7, 2, 9) scala.util.Sorting.quickSort(a) // Array(1, 2, 7, 9) // generalnie działa ze wszystkimi typami z definiowanym comparison op ArrayBuffer("Mary", "had", "a", "little", "lamb").max // "little" ++ łączy(concatenates) dwie kolekcje Sealed classes równoważne z klasą finalna Scala ## prawie takie samo jak w Javie hashCode Scala == odpowiednik w Javie equals aby sprawdzić referencje dwóch obiektów trzeba użyć eq –- Gdyby pojawiło się pytanie z tego tematu z Kotlina, to tam: == jest odpowiednikiem funkcji equals() (equality). Zanegowane: != === porównuje referencje, tak jak w JS (identity). Zanegowane: !== I nie ma operatorów >>, >>>, <<, &, |, ^ . Jest kolejno: shr, ushr, shl, and, or, xor I zmienne deklaruje się dokładnie tak jak w Scali. Dodatkowe info Kotlin Standardowe info o tym na co komu kotlin: - Niby szybko się kompiluje - Możesz używać ; ale nie musisz jak java scripcie - Jest na kompy, serwery i androida - null-pointer safety - Brak typów prymitywnych tylko obiekty - fun main() może być bez parametrów - a: Int w ten sposób deklarujemy - val n = BigInteger(12345678) są automatyczne typy (działa val i var) a nazywa się to Inferencja typów czyli, że sam se wylicza na podstawie tego co przyrównujesz - var a = 10L to jest long - val ulong = 42uL to jest long bez paru znaków - toByte() toShort() toInt() toLong() toFloat() toDouble() toChar() - == oraz != działa po ludzku nie jak w jave nie trzeba używać equals() - mamy za to === oraz !== do porównywania referencji - są < , > , <= , >= - są || i && - ale nie ma operatorów logicznych | i & zamiast tego jest or i and - są shr i shl czyli przesunięcia bitowe odpowiednio >> i << Hello world w kotlinie fun main(args: Array < String >) { println ("Hello, World!") } Kompilacja w kotlinie kotlinc nazwa.kt Odpalanie programu kotlin Nazwakt Pętle - for - while - do-while - foreach - repeat sth.forEach { print (it.toString()) } repeat (10) { i -> println ("Jesteśmy w ${i+1}-wszej linii.") } Tablice val arr = arrayOf (1, 2, 5) val squares = Array (10, { i -> i * i }) tablica w kotlinie to nie element języka tylko klasa kolekcji niemutowalne oznacza read-only Lambda val allowedUsers = users.filter { it.age > MINIMUM_AGE } to jest lambda fajna nie it jest domyślnym iteratorem jak własnego nie zadeklarujesz zamiast tego możesz tak: val allowedUsers = users.filter { x -> x.age > MINIMUM_AGE } I cyk kolekcja: val persons = listOf (Person("Max", 18), Person("David", 12), Person("Peter", 23), Person("Pamela", 23)) List (interfejs) -- lista niemutowalna (zawiera size, get etc.). Podobnie Set, Map.

When using a Task as an anonymous class, the most natural way to pass parameters to the Task is by using final variables. In this example, we pass to the Task the total number of times the Task should iterate.

```
     final int totalIterations = 9000000;
     Task<Integer> task = new Task<Integer>() {
         @Override protected Integer call() throws Exception {
             int iterations;
             for (iterations = 0; iterations < totalIterations; iterations++) {
                 if (isCancelled()) {
                     updateMessage("Cancelled");
                     break;
                 }
                 updateMessage("Iteration " + iterations);
                 updateProgress(iterations, totalIterations);
             }
             return iterations;
         }
     };
```

Since `totalIterations` is final, the `call` method can safely read it and refer to it from a background thread.

When writing Task libraries (as opposed to specific-use implementations), we need to use a different technique. In this case, I will create an IteratingTask which performs the same work as above. This time, since the IteratingTask is defined in its own file, it will need to have parameters passed to it in its constructor. These parameters are assigned to final variables.

```
     public class IteratingTask extends Task<Integer> {
         private final int totalIterations;

         public IteratingTask(int totalIterations) {
             this.totalIterations = totalIterations;
         }

         @Override protected Integer call() throws Exception {
             int iterations = 0;
             for (iterations = 0; iterations < totalIterations; iterations++) {
                 if (isCancelled()) {
                     updateMessage("Cancelled");
                     break;
                 }
                 updateMessage("Iteration " + iterations);
                 updateProgress(iterations, totalIterations);
             }
             return iterations;
         }
     }
```

And then when used:

```
     IteratingTask task = new IteratingTask(8000000);
```

In this way, parameters are passed to the IteratingTask in a safe manner, and again, are final. Thus, the `call` method can safely read this state from a background thread.

WARNING: Do not pass mutable state to a Task and then operate on it from a background thread. Doing so may introduce race conditions. In particular, suppose you had a SaveCustomerTask which took a Customer in its constructor. Although the SaveCustomerTask may have a final reference to the Customer, if the Customer object is mutable, then it is possible that both the SaveCustomerTask and some other application code will be reading or modifying the state of the Customer from different threads. Be very careful in such cases, that while a mutable object such as this Customer is being used from a background thread, that it is not being used also from another thread. In particular, if the background thread is reading data from the database and updating the Customer object, and the Customer object is bound to scene graph nodes (such as UI controls), then there could be a violation of threading rules! For such cases, modify the Customer object from the FX Application Thread rather than from the background thread.

```
     public class UpdateCustomerTask extends Task<Customer> {
         private final Customer customer;

         public UpdateCustomerTask(Customer customer) {
             this.customer = customer;
         }

         @Override protected Customer call() throws Exception {
             // pseudo-code:
             //   query the database
             //   read the values

             // Now update the customer
             Platform.runLater(new Runnable() {
                 @Override public void run() {
                     customer.setF setFirstName(rs.getString("FirstName"));
                     // etc
                 }
             });

             return customer;
         }
     }
```

A Task Which Returns No Value

Many, if not most, Tasks should return a value upon completion. For CRUD Tasks, one would expect that a "Create" Task would return the newly created object or primary key, a "Read" Task would return the read object, an "Update" task would return the number of records updated, and a "Delete" task would return the number of records deleted.

However sometimes there just isn't anything truly useful to return. For example, I might have a Task which writes to a file. Task has built into it a mechanism for indicating whether it has succeeded or failed along with the number of bytes written (the progress), and thus there is nothing really for me to return. In such a case, you can use the Void type. This is a special type in the Java language which can only be assigned the value of `null`. You would use it as follows:

```
     final String filePath = "/foo.txt";
```

```
final String contents = "Some contents";
Task<Void> task = new Task<Void>() {
    @Override protected Void call() throws Exception {
        File file = new File(filePath);
        FileOutputStream out = new FileOutputStream(file);
        // ... and other code to write the contents ...

        // Return null at the end of a Task of type Void
        return null;
    }
};
```

A Task Which Returns An ObservableList

Because the ListView, TableView, and other UI controls and scene graph nodes make use of ObservableList, it is common to want to create and return an ObservableList from a Task. When you do not care to display intermediate values, the easiest way to correctly write such a Task is simply to construct an ObservableList within the `call` method, and then return it at the conclusion of the Task.

25. Operacje terminalne na strumieniach Javy 8 zawsze zwracają void Fałsz

" Terminal operations produces a non-stream, result such as primitive value, a collection or no value at all." Przykłady: toArray(), collect(), count(), reduce(), forEach(), forEachOrdered(), min(), max(), anyMatch(), allMatch(), noneMatch(), findAny(), findFirst()

Operacje pośrednie (intermediate) z kolei zwracają strumień

26. W języku Java, jeden plik źródłowy może zawierać co najwyżej jeden publiczny interfejs. Odpowiedź: Prawda

27. W Javie Stałe pola statyczne możemy zdefiniować dla: Klasy abstrakcyjnej: Tak Klasy konkretnej: Tak Interfejsu: Tak

28. Java 8 wspiera wielodziedziczenie typów (dopuszcza następujący kod): class C extends A, B { /* */} Odpowiedź: Fałsz Tylko wiele interfejsów możemy rozszerzać

29. Jaką metodą wybudza się jeden uśpiony wątek? Odpowiedź: notify

31. Napisz funkcję przyjmującą jako argument tablicę stringów (zakładamy że argument ten jest != null oraz ma długość >0) i zwracającą string posiadający co najmniej 2 wystąpienia Jeśli takiego nie ma, to niech funkcja zwraca null. Jeśli więcej niż jeden string ma >= wystąpienia, to niech funkcja zwróci dowolny z nich.
Przykłady: -argumentem jest tablica {"kot","pies","mysz","pies"}, funkcja zwraca "pies", -argumentem jest tablica {"kot","pies","mysz","koza"}, funkcja zwraca null.

Odpowiedź: public String func(String[] tab) { LinkedList list = new LinkedList<>(); for (String s:tab) { if (list.contains(s)) { return s; } else list.add(s); } return null; }

32. Całkowite typy prymitywne Javy to: char, int, short, long, byte Dodatkowo zmiennoprzecinkowe, to: float, double jeszcze jest ………………………..:   boolean

33. W języku java jeden plik źródłowy może zawierać co najwyżej jedną klasę publiczną. prawda

34. W języku java jeden plik źródłowy może zawierać co najwyżej jeden publiczny interfejs. prawda

35. Przyporządkuj nazwy strumieni wejściowych do odpowiednich kategorii. Każdy użyj tylko raz, pewne mogą pozostać niewykorzystane

Binarny odczyt liczb float: DataInputStream Strumień bajtów: ? Strumień znaków: ?

36. Jeżeli w definicji typu adnotacyjnego umieścimy metaadnotację @Retention(RetentionPolicy.RUNTIME) oznaczać to, że będzie ona przetwarzana dopiero w czasie wykonania kodu Odpowiedź: Prawda

38. W Javie Klasę wewnętrzną możemy zdefiniować dla: Klasy konkretnej: Tak Interfejsu: Nie Klasy abstrakcyjnej: Tak

39. Napisz w Javie kod serwera dla aplikacji Klient Serwer działającej na Socketach.
Serwer powinien przyłączyć klienta, wysłać klientowi pewien String, rozłączyć go i zakończyć działanie. Wykorzystaj słowa kluczowe: accept, close, flush, getOutputStream, IOexception, java.io, java.net, println, PrintWriter, ServerSocket, Socket

Odpowiedź: //kod ze zdj, ocenione na 3 z 3 try{ // utworzenie gniazda String serverHost = …; int serverPort = …;

Socket socket = new Socket(serverHost, serverPort)

OutputStream sockOut = socket.getOutputStream(); InputStream sockIn = socket.getInputStream();

socktOut.write(...); socktIn.read();

sockOut.close(); sockIn.close(); socket.close(); } catch(UnknownHostException exc) { // nieznany host } catch(SocketException exc) { // wyjątki związane z komunikacją przez gniazda } catch(IOException exc) { // inne wyjątki we/wy }

40. Napisz funkcję przyjmującą tablicę stringów (zakładamy że ten argument != null i ma długość > 0) i zwracającą string który występuje w tej tablicy co najmniej dwa razy. Jeśli takiego nie ma zwróć null public String foo(String args[]) { HashSet set = new HashSet<>(); for(String s : args) if(set.contains(s)) return s; else set.add(s); return null; }

41. Słowo kluczowe "this" może być użyte w ciele konstruktora.
Odpowiedź: Prawda

42. Jaki napis wypisze poniższy fragment kodu? Integer i = 0; System.out.println(i != null ? 1<<2 : 6 >> 1);

Odpowiedź: 4 Objaśnienie: 0b100=2^2 = 4 0b1000=2^3 = 8 0b10000=2^4 = 16

44. Słowa kluczowe używane przy wyjątkach to: try, catch, ***, throw, *** Odpowiedź: throws, finally

46. Można stworzyć konstruktor klasy abstrakcyjnej. Odpowiedź: Prawda

47. Obiekt w Javie może dziedziczyć stan z kilku KLAS bazowych. Odpowiedź: Fałsz

48. Przyporządkuj nazwy strumieni wejściowych do odpowiednich kategorii. Każdy użyj tylko raz, pewne mogą pozostać niewykorzystane.

Konwersja char - byte: OutputStreamWriter -- do sprawdzenia!! Strumień obiektów, np Date: ObjectInputStream Odczyt linii pliku tekstowego: BufferedReader

49. Metodę domyślną interfejsu w javie oznaczamy słowem Odpowiedź: default

Dodatkowe info Java Jeżeli w definicji typu adnotacyjnego umieścimy metadnotację @Retention(RetentionPolicy.RUNTIME) oznacza to że będzie ona przetwarzana dopiero w momencie wykonywania kodu

W języku java plik źródłowy musi mieć tę samą nazwę co zawarta w nim klasa publiczna, natomiast klasa niepubliczna może mieć inną.

W javie 8 adnotacją możemy opatrzeć argument metody

W javie klazulę extends możemy umieścić w nagłówku dla: klasy konkretnej, klasy abstrakcyjnej, interfejsu.

Strumienie wejsciowe: Binarny odczyt liczby float: DataInputStream Strumień bajtów: FileInputStream Strumien znaków: BufferedReader

W javie klasę wewnętrzną można zdefiniować dla: klasy konkretnej, klasy abstrakcyjnej, interfejsu

@Override - adnotacja metody, wykorzystywana podczas dziedziczenia, na przykład klasa Object ma metodę hashCode(), i w naszym obiekcie można ją nadpisać. @Deprecated - porzucona funkcjonalność, może się popsuć w kolejnej wersji @Test - przy testach JUnit @SupressWarnings - ignoruje ostrzeżenia

Tablice nie mogą mieć generycznych typów: Pair[] table = new Pair<>[10]; // !!!

Collection - kolekcja zawierająca typy implementujące interfejs Andrzej / dziedziczące z klasy Andrzej

Klasy strumieni wejścia/wejścia - bazowe klasy abstrakcyjne: InputStream, OutputStream - sekwencje bajtów: FileInputStream, FileOutputStream - sekwencje dowolnych typów danych: DataInputStream, DataOutputStream - obiekty: ObjectInputStream, ObjectOutputStream

Klasy operacji na tekście: FileReader, FileWriter Obsługa niestandardowego kodowania: opakowanie FileReader w InputStreamReader Odczyt z konsoli: Scanner

Serializacja: implementacja interfejsu Serializable , metody readObject() i writeObject() w obiekcie strumienia

Programowanie sieciowe Adresowanie localhosta: InetAddress a0 = InetAddress.getLocalHost(); Host odległy: InetAddress address = InetAddress.getByName(" www.oreilly.com ");

Nie uwierzytelniony aplet nie może wykonywać operacji getByName i getAllByName !

Strumienie filtrujące oraz transformujące są przykładem dekoratorów

PrintWriter jest właściwą klasą do zapisu plików tekstowych! Klasy InputStreamReader oraz OutputStreamWriter pozwalają na przezroczystą dla programisty konwersję źródłowego strumienia bajtowego na znaki Unicode i na odwrót.

Gniazda umożliwiają niskopoziomową komunikację sieciową. Gniazdo to jeden z końców dwukierunkowego połączenia pomiędzy klientem a serwerem.

Pakiet datagramowy (zwykle UDP) jest reprezentowany przez klasę DatagramPacket.

Gniazda bezpołączeniowe (ang. datagram sockets) umożliwiają przesyłanie i odbieranie datagramów. Najczęściej są to pakiety UDP. Przy tym połączeniu nie rozróżniamy klienta i serwera, ponieważ połączenie nie jest zestawiane

(RPC) Remote Procedure Call -   Zdalne wywołanie procedur (RMI) Remote Method Invocation - Zdalne wywoływanie metod System RMI pozwala na to, aby obiekt działający na jednej maszynie wirtualnej wywoływał metody obiektu działającego na innej maszynie wirtualnej Javy

Protokół RMI zapewnia,że obiekty zdalne mogą być zniszczone dopiero, kiedy nie ma do nich ani zdalnych, ani lokalnych referencji. RMI do przekazywania obiektów przez sieć wykorzystuje mechanizm serializacji.

Głównym problemem przy RMI i zarządzaniu pamięcią jest rozproszone zwalnianie pamięci (distributed garbage collection) . Rozwiązaniem tego jest licznik referencji dla każdego obiektu zdalnego.

RMI pozwala na szybkie i łatwe tworzenie aplikacji rozproszonych. Ze względu na silny związek z Javą RMI jest w stanie współdziałać z dziedziczeniem. Wywołania RMI są blokujące, jak zwykle w RPC, jednak wbudowana wielowątkowość pozwala na uruchomienie innych funkcji programu w trakcie obliczeń na maszynie zdalnej

Bazy danych

Rodzaje sterowników JDBC (Java Database Connectivity) 1) mostek JDBC-ODBC 2) mieszany kod rodzimy klienta bazy danych i Javy 3) czysty kod Javy komunikujący się niezależnym od bazy danych protokołem sieciowym, tłumaczonym przez pośredni serwer 4) czysty kod Javy komunikujący się zależnym od bazy danych protokołem sieciowym

Metody obiektu typu ResultSet do pobierania wartości pobranych rekordów
first() - przejście do pierwszego rekordu last() - przejście do ostatniego rekordu next() - przejście do następnego rekordu previous() - przejście do poprzedniego rekordu getFloat("nazwa kolumny") - pobranie wartości float getInt("nazwa kolumny") - pobranie wartości integer getDate("nazwa kolumny") - pobranie wartości typu Date getString("nazwa kolumny") - pobranie wartości String Zadan

1. Co zwróci następująca konstrukcja (proszę zaznaczyć też typ): for(i <- 0 untril 4 by 2; c <- "ab") yield (c+i).toChar

Napisz w scali wyrażenie tworzące zmienną x o wartości -1 jeśli y > 0, "!!!" w przeciwnym wypadku. Jaki będzie typ x? def manOf[T: Manifest](t: T): Manifest[T] = manifest[T] var y = -20 var x = if (y > 0) -1 else "!!!" println(manOf(x)) Typ: Any

2. W języku Scala 1. Wartość(value) b jest ArrayBufferem. Co oznacza kod: b.sortWith(_>_)?   2.   Mamy następujące dwie klasy w Scali: class Osoba1 (var wiek = 20) oraz class Osoba2 (val wiek = 20) Czy można ustawić wiek obiektu klasy Osoba1 i Osoba , a jeśli tak to w jakis sposób?

Odpowiedź: 1. Oznacza to, że należy posortować wartości malejąco. 2. Nie można ustawić wieku osoby klasy Osoba2 gdyż val jest niemutowalną zmienną. Można natomiast ustawić wiek Osoba1; val os: Osoba1 =new Osoba1(); os.wiek = 30;

3. Co zwróci następująca konstrukcja: for(c<- "abc"; i <- 1 to 2) yield (c+i).toChar Odpowiedź: bccdde

4. Napisz w Scali kod zapisujący w kolekcji listowej (np. Vector) wszystkie pary liczb
(i,j), 1<= i, j <= 5 których suma jest nie nieparzysta, Użyj for: Odpowiedź: var e = for(i <- Range(0, 2); j <- Range(0, 5) if (i + j) % 2 == 1) yield (i, j)

5. Jak jest różnica między val a = myArray(10) a val a = new myArray(10) Zakładając że myArray to jakiś typ tablicowy

Odpowiedź: Pierwsza wersja: używa metody apply z obiektu towarzyszącego klasy myArray. Drugie wywołuje konstruktor

6. Co zwróci następująca konstrukcja (proszę zaznaczyć też typ): for(i <- 0 until 4 by 2; c <- "ab") yield (c + i).toChar Nie znam scali, więc nie wytłumaczę dlaczego, ale: Vector (a , b , c , d)

7. Podaj po jednym przykładzie kolekcji mutowalnej i niemutowalnej z biblioteki standardowej scali Mutowalne: ● MutableList ● Arraybuffer ● Listbuffer ● Stringbuilder ● Linkedlist ● Double linked list ● mutable.Queue ● Nie chce mi się pisać więcej https://www.scala-lang.org/api/2.12.2/scala/collection/mutable/index.html Niemutowalne: ● List ● Stream ● Vector ● Stack ● Queue ● Range ● Nie chce m

w Javie List NIE JEST podtypem List.
W Kotlinie JEST !

Dodatkowe info Scala - same ciekawostki, lub rzeczy które mogą się przydać val - value - immutable(niezmienny) var - variable - muttable(zmienny) Scala od wersji 5 domyśla się typu. jednak możliwe jest określenie typu odgórnie np: val x:Int = 6 val y: Double = -3.2

Vector - takie samo jak w Javie ArrayList ale "niezmienny"

brevity(zwięzłość) ● return może zostać pominięty, wynikiem jest ostatnie wyrażenie złożone -- do sprawdzenia ● średniki mogą zostać pominięte ● "parens"(chyba chodzi o nawiasy ()) w wywołaniach mogą zostać pominięte jeśli metoda ma 0 lub 1 argumentów - tylko w tak zwanej notacji operatora ● tak samo z kropką po nazwie obiektu, a przed metodą - w takim samym przypadku jak wyżej(notacji operato

Metody bez parametrów dobrym stylem jest używać () dla mutatorów(getów) i nie używać () dla accesorów(setter)

Typy Scali

Klasy Scala Any, AnyRef i AnyVal nie pojawiają się jako klasy w kodzie bajtowym - z powodu ograniczeń JVM. (Java Virtual Machine)

W Scali wszystko jest obiektem(w javie nie), JVM nie może ogarnąć tego i wygenerowany kod bajtowy nie pokazuje tego.

W Scali wszystkie obiekty są podklasą Any. AnyRef w Scala jest odpowiednikiem java.lang. Object (przynajmniej na JVM). AnyVal w Scali są odpowiednikiem typów prymitywnych w javie

Typ Nothing nie dziedziczy z żadnego innego typu

Lists jak i String są niezmienne(immutable) dostęp do początku/konca z O(1) reszta operacji O(n)

Listy są konstruowane od prawej do lewej Indexing: list(0) Slicing: list.slice(1, 3); list.slice(2, list.last) Reversing: list.reverse Sorting: list.sorted
=>   nieformalnie operator rakiety pętla for: for (y <- List(1, 2, 3)) { println(y) }

for (x<- List(1,2,3); y<-List(4,5)) yield x * y

wynik: List(4, 5, 8, 10, 12, 15) for(x <- 1 to 7 // generator y = x%2; // definicja? if(y == 0) // filtr   } yield { println(x) x } Wynik: 2 4 6

dla pętli for możliwość używania wielu filtrów generowana kolekcja jest kompatibilna z pierwszym podanym generatorem

Funkcja operatora(operator function) def ~=(x: Double, y: Double, precision: Double) = {   if ((x - y).abs < precision) true else false }

wywołanie: ~=(a, b, 0.0001) wynik: true

Tablice są zmienne(mutable)

Sortowanie tablicy(ale nie arrayBuffer) val a = Array(1, 7, 2, 9) scala.util.Sorting.quickSort(a) // Array(1, 2, 7, 9) // generalnie działa ze wszystkimi typami z definiowanym comparison op   ArrayBuffer("Mary", "had", "a", "little", "lamb").max // "little"

++ łączy(concatenates) dwie kolekcje
Sealed classes równoważne z klasą finalna Scala ## prawie takie samo jak w Javie hashCode Scala == odpowiednik w Javie equals aby sprawdzić referencje dwóch obiektów trzeba użyć eq

--- Gdyby pojawiło się pytanie z tego tematu z Kotlina, to tam: == jest odpowiednikiem funkcji equals() (equality
). Zanegowane: != === porównuje referencje, tak jak w JS (identity
). Zanegowane: !== I nie ma operatorów >>, >>>, <<, &, |, ^ . Jest kolejno: shr, ushr, shl, and, or, xor I zmienne deklaruje się dokładnie tak jak w Scali.

Dodatkowe info Kotlin Standardowe info o tym na co komu kotlin: - Niby szybko się kompiluje - Możesz używać ; ale nie musisz jak java scripcie - Jest na kompy, serwery i androida - null-pointer safety - Brak typów prymitywnych tylko obiekty - fun main() może być bez parametrów - a: Int w ten sposób deklarujemy - val n = BigInteger(12345678) są automatyczne typy (działa val i var) a nazywa

Hello world w kotlinie fun main(args: Array < String >) { println ("Hello, World!") }

Kompilacja w kotlinie kotlinc nazwa.kt

Odpalanie programu kotlin Nazwakt

Pętle - for - while - do-while - foreach - repeat

sth.forEach { print (it.toString()) } repeat (10) { i -> println ("Jesteśmy w ${i+1}-wszej linii.") }

Tablice val arr = arrayOf (1, 2, 5) val squares = Array (10, { i -> i * i })

tablica w kotlinie to nie element języka tylko klasa kolekcji

niemutowalne oznacza read-only

Lambda val allowedUsers = users.filter { it.age > MINIMUM_AGE } to jest lambda fajna nie it jest domyślnym iteratorem jak własnego nie zadeklarujesz

zamiast tego możesz tak: val allowedUsers = users.filter { x -> x.age > MINIMUM_AGE }

I cyk kolekcja: val persons = listOf (Person("Max", 18), Person("David", 12), Person("Peter", 23), Person("Pamela", 23))

List (interfejs) -- lista niemutowalna (zawiera size, get etc.). Podobnie Set, Map.

In the above example, we are going to create 100 rectangles and return them from this task. An ObservableList is created within the `call` method, populated, and then returned.

A Task Which Returns Partial Results

Sometimes you want to create a Task which will return partial results. Perhaps you are building a complex scene graph and want to show the scene graph as it is being constructed. Or perhaps you are reading a large amount of data over the network and want to display the entries in a TableView as the data is arriving. In such cases, there is some shared state available both to the FX Application Thread and the background thread. Great care must be taken to **never update shared state from any thread other than the FX Application Thread**.

The easiest way to do this is to expose a new property on the Task which will represent the partial result. Then make sure to use `Platform.runLater` when adding new items to the partial result.

```
public class PartialResultsTask extends Task<ObservableList<Rectangle>> {
    // Uses Java 7 diamond operator
    private ReadOnlyObjectWrapper> partialResults =
            new ReadOnlyObjectWrapper<>(this, "partialResults",
                    FXCollections.observableArrayList(new ArrayList()));

    public final ObservableList getPartialResults() { return partialResults.get(); }
    public final ReadOnlyObjectProperty> partialResultsProperty() {
        return partialResults.getReadOnlyProperty();
    }

    @Override protected ObservableList call() throws Exception {
        updateMessage("Creating Rectangles...");
        for (int i=0; i<100; i++) {
            if (isCancelled()) break;
            final Rectangle r = new Rectangle(10, 10);
            r.setX(10 * i);
            Platform.runLater(new Runnable() {
                @Override public void run() {
                    partialResults.get().add(r);
                }
            });
            updateProgress(i, 100);
        }
        return partialResults.get();
    }
}
```

A Task Which Modifies The Scene Graph

Generally, Tasks should not interact directly with the UI. Doing so creates a tight coupling between a specific Task implementation and a specific part of your UI. However, when you do want to create such a coupling, you must ensure that you use `Platform.runLater` so that any modifications of the scene graph occur on the FX Application Thread.

```
     final Group group = new Group();
     Task<Void> task = new Task<Void>() {
         @Override protected Void call() throws Exception {
             for (int i=0; i<100; i++) {
                 if (isCancelled()) break;
                 final Rectangle r = new Rectangle(10, 10);
                 r.setX(10 * i);
                 Platform.runLater(new Runnable() {
                     @Override public void run() {
                         group.getChildren().add(r);
                     }
                 });
             }
             return null;
         }
     };
```

Reacting To State Changes Generically

Sometimes you may want to write a Task which updates its progress, message, text, or in some other way reacts whenever a state change happens on the Task. For example, you may want to change the status message on the Task on Failure, Success, Running, or Cancelled state changes.

```
     Task<Integer> task = new Task<Integer>() {
         @Override protected Integer call() throws Exception {
             int iterations = 0;
             for (iterations = 0; iterations < 100000; iterations++) {
                 if (isCancelled()) {
                     break;
                 }
                 System.out.println("Iteration " + iterations);
             }
             return iterations;
         }

         @Override protected void succeeded() {
             super.succeeded();
             updateMessage("Done!");
         }

         @Override protected void cancelled() {
             super.cancelled();
             updateMessage("Cancelled!");
         }

         @Override protected void failed() {
             super.failed();
             updateMessage("Failed!");
         }
     };
```

Property Summary

Properties

Type	Property and Description
`ReadOnlyObjectProperty<java.lang.Throwable>`	`exception` Gets the ReadOnlyObjectProperty representing any exception which occurred.
`ReadOnlyStringProperty`	`message` Gets the ReadOnlyStringProperty representing the message.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onCancelled` The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onFailed` The onFailed event handler is called whenever the Task state transitions to the FAILED state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onRunning` The onRunning event handler is called whenever the Task state transitions to the RUNNING state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onScheduled` The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onSucceeded` The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.
`ReadOnlyDoubleProperty`	`progress` Gets the ReadOnlyDoubleProperty representing the progress.
`ReadOnlyBooleanProperty`	`running` Gets the ReadOnlyBooleanProperty representing whether the Worker is running.
`ReadOnlyObjectProperty<Worker.State>`	`state` Gets the ReadOnlyObjectProperty representing the current state.
`ReadOnlyStringProperty`	`title` Gets the ReadOnlyStringProperty representing the title.
`ReadOnlyDoubleProperty`	`totalWork` Gets the ReadOnlyDoubleProperty representing the maximum amount of work that needs to be done.
`ReadOnlyObjectProperty<V>`	`value` Gets the ReadOnlyObjectProperty representing the value.
`ReadOnlyDoubleProperty`	`workDone` Gets the ReadOnlyDoubleProperty representing the current progress.

Nested Class Summary

Nested classes/interfaces inherited from interface javafx.concurrent.Worker

`Worker.State`

Constructor Summary

Constructors

Constructor and Description
`Task()` Creates a new Task.

Method Summary

Methods

Modifier and Type	Method and Description
`<T extends Event> void`	`addEventFilter(EventType<T> eventType, EventHandler<? super T> eventFilter)` Registers an event filter to this task.
`<T extends Event> void`	`addEventHandler(EventType<T> eventType, EventHandler<? super T> eventHandler)` Registers an event handler to this task.
`EventDispatchChain`	`buildEventDispatchChain(EventDispatchChain tail)` Construct an event dispatch chain for this target.
`protected abstract V`	`call()` Invoked when the Task is executed, the call method must be overridden and implemented by subclasses.
`boolean`	`cancel()` Terminates execution of this Worker.
`boolean`	`cancel(boolean mayInterruptIfRunning)`
`protected void`	`cancelled()` A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the CANCELLED state.
`ReadOnlyObjectProperty<java.lang.Throwable>`	`exceptionProperty()` Gets the ReadOnlyObjectProperty representing any exception which occurred.
`protected void`	`failed()` A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the FAILED state.
`void`	`fireEvent(Event event)` Fires the specified event.
`java.lang.Throwable`	`getException()` Gets the value of the property exception.
`java.lang.String`	`getMessage()` Gets the value of the property message.
`EventHandler<WorkerStateEvent>`	`getOnCancelled()` The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.
`EventHandler<WorkerStateEvent>`	`getOnFailed()` The onFailed event handler is called whenever the Task state transitions to the FAILED state.
`EventHandler<WorkerStateEvent>`	`getOnRunning()` The onRunning event handler is called whenever the Task state transitions to the RUNNING state.
`EventHandler<WorkerStateEvent>`	`getOnScheduled()` The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.
`EventHandler<WorkerStateEvent>`	`getOnSucceeded()` The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.
`double`	`getProgress()` Gets the value of the property progress.
`Worker.State`	`getState()` Gets the value of the property state.
`java.lang.String`	`getTitle()` Gets the value of the property title.
`double`	`getTotalWork()` Gets the value of the property totalWork.
`V`	`getValue()` Gets the value of the property value.
`double`	`getWorkDone()` Gets the value of the property workDone.
`boolean`	`isRunning()` Gets the value of the property running.
`ReadOnlyStringProperty`	`messageProperty()` Gets the ReadOnlyStringProperty representing the message.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onCancelledProperty()` The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onFailedProperty()` The onFailed event handler is called whenever the Task state transitions to the FAILED state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onRunningProperty()` The onRunning event handler is called whenever the Task state transitions to the RUNNING state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onScheduledProperty()` The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.
`ObjectProperty<EventHandler<WorkerStateEvent>>`	`onSucceededProperty()` The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.
`ReadOnlyDoubleProperty`	`progressProperty()` Gets the ReadOnlyDoubleProperty representing the progress.
`<T extends Event> void`	`removeEventFilter(EventType<T> eventType, EventHandler<? super T> eventFilter)` Unregisters a previously registered event filter from this task.
`<T extends Event> void`	`removeEventHandler(EventType<T> eventType, EventHandler<? super T> eventHandler)` Unregisters a previously registered event handler from this task.
`protected void`	`running()` A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the RUNNING state.
`ReadOnlyBooleanProperty`	`runningProperty()` Gets the ReadOnlyBooleanProperty representing whether the Worker is running.
`protected void`	`scheduled()` A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the SCHEDULED state.
`protected <T extends Event> void`	`setEventHandler(EventType<T> eventType, EventHandler<? super T> eventHandler)` Sets the handler to use for this event type.
`void`	`setOnCancelled(EventHandler<WorkerStateEvent> value)` The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.
`void`	`setOnFailed(EventHandler<WorkerStateEvent> value)` The onFailed event handler is called whenever the Task state transitions to the FAILED state.
`void`	`setOnRunning(EventHandler<WorkerStateEvent> value)` The onRunning event handler is called whenever the Task state transitions to the RUNNING state.
`void`	`setOnScheduled(EventHandler<WorkerStateEvent> value)` The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.
`void`	`setOnSucceeded(EventHandler<WorkerStateEvent> value)` The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.
`ReadOnlyObjectProperty<Worker.State>`	`stateProperty()` Gets the ReadOnlyObjectProperty representing the current state.
`protected void`	`succeeded()` A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the SUCCEEDED state.
`ReadOnlyStringProperty`	`titleProperty()` Gets the ReadOnlyStringProperty representing the title.
`ReadOnlyDoubleProperty`	`totalWorkProperty()` Gets the ReadOnlyDoubleProperty representing the maximum amount of work that needs to be done.
`protected void`	`updateMessage(java.lang.String message)` Updates the `message` property.
`protected void`	`updateProgress(double workDone, double max)` Updates the `workDone`, `totalWork`, and `progress` properties.
`protected void`	`updateProgress(long workDone, long max)` Updates the `workDone`, `totalWork`, and `progress` properties.
`protected void`	`updateTitle(java.lang.String title)` Updates the `title` property.
`ReadOnlyObjectProperty<V>`	`valueProperty()` Gets the ReadOnlyObjectProperty representing the value.
`ReadOnlyDoubleProperty`	`workDoneProperty()` Gets the ReadOnlyDoubleProperty representing the current progress.

Methods inherited from class java.util.concurrent.FutureTask

`done, get, get, isCancelled, isDone, run, runAndReset, set, setException`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Property Detail

state

`public final ReadOnlyObjectProperty<Worker.State> stateProperty`

Specified by:

`stateProperty` in interface `Worker<V>`

Returns:

The property representing the state

See Also:

`getState()`

onScheduled

`public final ObjectProperty<EventHandler<WorkerStateEvent>> onScheduledProperty`

The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.

Returns:

the onScheduled event handler property

See Also:

`getOnScheduled()`, `setOnScheduled(EventHandler)`

onRunning

`public final ObjectProperty<EventHandler<WorkerStateEvent>> onRunningProperty`

The onRunning event handler is called whenever the Task state transitions to the RUNNING state.

Returns:

the onRunning event handler property

See Also:

`getOnRunning()`, `setOnRunning(EventHandler)`

onSucceeded

```
public final ObjectProperty<EventHandler<WorkerStateEvent>> onSucceededProperty
```

The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.

Returns:

the onSucceeded event handler property

See Also:

`getOnSucceeded()`, `setOnSucceeded(EventHandler)`

onCancelled

```
public final ObjectProperty<EventHandler<WorkerStateEvent>> onCancelledProperty
```

The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.

Returns:

the onCancelled event handler property

See Also:

`getOnCancelled()`, `setOnCancelled(EventHandler)`

onFailed

```
public final ObjectProperty<EventHandler<WorkerStateEvent>> onFailedProperty
```

The onFailed event handler is called whenever the Task state transitions to the FAILED state.

Returns:

the onFailed event handler property

See Also:

`getOnFailed()`, `setOnFailed(EventHandler)`

value

```
public final ReadOnlyObjectProperty<V> valueProperty
```

Specified by:

`valueProperty` in interface `Worker<V>`

Returns:

The property representing the current value

See Also:

`getValue()`

exception

```
public final ReadOnlyObjectProperty<java.lang.Throwable> exceptionProperty
```

Specified by:

`exceptionProperty` in interface `Worker<V>`

Returns:

the property representing the exception

See Also:

`getException()`

workDone

```
public final ReadOnlyDoubleProperty workDoneProperty
```

Specified by:

`workDoneProperty` in interface `Worker<V>`

Returns:

The property representing the amount of work done

See Also:

`getWorkDone()`

totalWork

```
public final ReadOnlyDoubleProperty totalWorkProperty
```

Specified by:

`totalWorkProperty` in interface `Worker<V>`

Returns:

the property representing the total work to be done

See Also:

`getTotalWork()`

progress

```
public final ReadOnlyDoubleProperty progressProperty
```

Specified by:

`progressProperty` in interface `Worker<V>`

Returns:

the property representing the progress

See Also:

`getProgress()`

running

```
public final ReadOnlyBooleanProperty runningProperty
```

Specified by:

`runningProperty` in interface `Worker<V>`

Returns:

the property representing whether the worker is running

See Also:

`isRunning()`

message

```
public final ReadOnlyStringProperty messageProperty
```

Specified by:

`messageProperty` in interface `Worker<V>`

Returns:

a property representing the current message

See Also:

`getMessage()`

title

```
public final ReadOnlyStringProperty titleProperty
```

Specified by:

`titleProperty` in interface `Worker<V>`

Returns:

the property representing the current title

See Also:

`getTitle()`

Constructor Detail

Task

```
public Task()
```

Creates a new Task.

Method Detail

call

```
protected abstract V call()
                   throws java.lang.Exception
```

Invoked when the Task is executed, the call method must be overridden and implemented by subclasses. The call method actually performs the background thread logic. Only the updateProgress, updateMessage, and updateTitle methods of Task may be called from code within this method. Any other interaction with the Task from the background thread will result in runtime exceptions.

Returns:

The result of the background work, if any.

Throws:

`java.lang.Exception` - an unhandled exception which occurred during the background operation

getState

```
public final Worker.State getState()
```

Gets the value of the property state.

Specified by:

`getState` in interface `Worker<V>`

Returns:

The current state of this Worker

stateProperty

```
public final ReadOnlyObjectProperty<Worker.State> stateProperty()
```

Description copied from interface: `Worker`
Gets the ReadOnlyObjectProperty representing the current state.

Specified by:

`stateProperty` in interface `Worker<V>`

Returns:

The property representing the state

See Also:

`getState()`

onScheduledProperty

```
public final ObjectProperty<EventHandler<WorkerStateEvent>> onScheduledProperty()
```

The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.

Returns:

the onScheduled event handler property

See Also:

`getOnScheduled()`, `setOnScheduled(EventHandler)`

getOnScheduled

```
public final EventHandler<WorkerStateEvent> getOnScheduled()
```

The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.

Returns:

the onScheduled event handler, if any

setOnScheduled

```
public final void setOnScheduled(EventHandler<WorkerStateEvent> value)
```

The onSchedule event handler is called whenever the Task state transitions to the SCHEDULED state.

Parameters:

`value` - the event handler, can be null to clear it

scheduled

```
protected void scheduled()
```

A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the SCHEDULED state. This method is invoked on the FX Application Thread after any listeners of the state property and after the Task has been fully transitioned to the new state.

onRunningProperty

```
public final ObjectProperty<EventHandler<WorkerStateEvent>> onRunningProperty()
```

The onRunning event handler is called whenever the Task state transitions to the RUNNING state.

Returns:

the onRunning event handler property

See Also:

`getOnRunning()`, `setOnRunning(EventHandler)`

getOnRunning

```
public final EventHandler<WorkerStateEvent> getOnRunning()
```

The onRunning event handler is called whenever the Task state transitions to the RUNNING state.

Returns:

the onRunning event handler, if any

setOnRunning

```
public final void setOnRunning(EventHandler<WorkerStateEvent> value)
```

The onRunning event handler is called whenever the Task state transitions to the RUNNING state.

Parameters:

`value` - the event handler, can be null to clear it

running

```
protected void running()
```

A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the RUNNING state. This method is invoked on the FX Application Thread after any listeners of the state property and after the Task has been fully transitioned to the new state.

onSucceededProperty

`public final ObjectProperty<EventHandler<WorkerStateEvent>> onSucceededProperty()`

The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.

Returns:

the onSucceeded event handler property

See Also:

`getOnSucceeded()`, `setOnSucceeded(EventHandler)`

getOnSucceeded

`public final EventHandler<WorkerStateEvent> getOnSucceeded()`

The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.

Returns:

the onSucceeded event handler, if any

setOnSucceeded

`public final void setOnSucceeded(EventHandler<WorkerStateEvent> value)`

The onSucceeded event handler is called whenever the Task state transitions to the SUCCEEDED state.

Parameters:

`value` - the event handler, can be null to clear it

succeeded

`protected void succeeded()`

A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the SUCCEEDED state. This method is invoked on the FX Application Thread after any listeners of the state property and after the Task has been fully transitioned to the new state.

onCancelledProperty

`public final ObjectProperty<EventHandler<WorkerStateEvent>> onCancelledProperty()`

The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.

Returns:

the onCancelled event handler property

See Also:

`getOnCancelled()`, `setOnCancelled(EventHandler)`

getOnCancelled

`public final EventHandler<WorkerStateEvent> getOnCancelled()`

The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.

Returns:

the onCancelled event handler, if any

setOnCancelled

`public final void setOnCancelled(EventHandler<WorkerStateEvent> value)`

The onCancelled event handler is called whenever the Task state transitions to the CANCELLED state.

Parameters:

`value` - the event handler, can be null to clear it

cancelled

`protected void cancelled()`

A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the CANCELLED state. This method is invoked on the FX Application Thread after any listeners of the state property and after the Task has been fully transitioned to the new state.

onFailedProperty

`public final ObjectProperty<EventHandler<WorkerStateEvent>> onFailedProperty()`

The onFailed event handler is called whenever the Task state transitions to the FAILED state.

Returns:

the onFailed event handler property

See Also:

`getOnFailed()`, `setOnFailed(EventHandler)`

getOnFailed

`public final EventHandler<WorkerStateEvent> getOnFailed()`

The onFailed event handler is called whenever the Task state transitions to the FAILED state.

Returns:

the onFailed event handler, if any

setOnFailed

`public final void setOnFailed(EventHandler<WorkerStateEvent> value)`

The onFailed event handler is called whenever the Task state transitions to the FAILED state.

Parameters:

`value` - the event handler, can be null to clear it

failed

`protected void failed()`

A protected convenience method for subclasses, called whenever the state of the Task has transitioned to the FAILED state. This method is invoked on the FX Application Thread after any listeners of the state property and after the Task has been fully transitioned to the new state.

getValue

`public final V getValue()`

Gets the value of the property value.

Specified by:

`getValue` in interface `Worker<V>`

Returns:

the current value of this Worker

valueProperty

`public final ReadOnlyObjectProperty<V> valueProperty()`

Description copied from interface: `Worker`
Gets the ReadOnlyObjectProperty representing the value.

Specified by:

`valueProperty` in interface `Worker<V>`

Returns:

The property representing the current value

See Also:

`getValue()`

getException

`public final java.lang.Throwable getException()`

Gets the value of the property exception.

Specified by:

`getException` in interface `Worker<V>`

Returns:

the exception, if one occurred

exceptionProperty

`public final ReadOnlyObjectProperty<java.lang.Throwable> exceptionProperty()`

Description copied from interface: `Worker`
Gets the ReadOnlyObjectProperty representing any exception which occurred.

Specified by:

`exceptionProperty` in interface `Worker<V>`

Returns:

the property representing the exception

See Also:

`getException()`

getWorkDone

`public final double getWorkDone()`

Gets the value of the property workDone.

Specified by:

`getWorkDone` in interface `Worker<V>`

Returns:

the amount of work done

See Also:

`Worker.totalWorkProperty()`, `Worker.progressProperty()`

workDoneProperty

`public final ReadOnlyDoubleProperty workDoneProperty()`

Description copied from interface: `Worker`
Gets the ReadOnlyDoubleProperty representing the current progress.

Specified by:

`workDoneProperty` in interface `Worker<V>`

Returns:

The property representing the amount of work done

See Also:

`getWorkDone()`

getTotalWork

`public final double getTotalWork()`

Gets the value of the property totalWork.

Specified by:

`getTotalWork` in interface `Worker<V>`

Returns:

the total work to be done

See Also:

`Worker.workDoneProperty()`, `Worker.progressProperty()`

totalWorkProperty

`public final ReadOnlyDoubleProperty totalWorkProperty()`

Description copied from interface: `Worker`
Gets the ReadOnlyDoubleProperty representing the maximum amount of work that needs to be done. These "work units" have meaning to the Worker implementation, such as the number of bytes that need to be downloaded or the number of images to process or some other such metric.

Specified by:

`totalWorkProperty` in interface `Worker<V>`

Returns:

the property representing the total work to be done

See Also:

`getTotalWork()`

getProgress

`public final double getProgress()`

Gets the value of the property progress.

Specified by:

`getProgress` in interface `Worker<V>`

Returns:

the current progress

See Also:

`Worker.workDoneProperty()`, `Worker.totalWorkProperty()`

progressProperty

`public final ReadOnlyDoubleProperty progressProperty()`

Description copied from interface: `Worker`
Gets the ReadOnlyDoubleProperty representing the progress.

Specified by:

`progressProperty` in interface `Worker<V>`

Returns:

the property representing the progress

See Also:

`getProgress()`

isRunning

```
public final boolean isRunning()
```

Gets the value of the property running.

Specified by:

`isRunning` in interface `Worker<V>`

Returns:

true if this Worker is running

runningProperty

```
public final ReadOnlyBooleanProperty runningProperty()
```

Description copied from interface: `Worker`
Gets the ReadOnlyBooleanProperty representing whether the Worker is running.

Specified by:

`runningProperty` in interface `Worker<V>`

Returns:

the property representing whether the worker is running

See Also:

`isRunning()`

getMessage

```
public final java.lang.String getMessage()
```

Gets the value of the property message.

Specified by:

`getMessage` in interface `Worker<V>`

Returns:

the current message

messageProperty

```
public final ReadOnlyStringProperty messageProperty()
```

Description copied from interface: `Worker`
Gets the ReadOnlyStringProperty representing the message.

Specified by:

`messageProperty` in interface `Worker<V>`

Returns:

a property representing the current message

See Also:

`getMessage()`

getTitle

```
public final java.lang.String getTitle()
```

Gets the value of the property title.

Specified by:

`getTitle` in interface `Worker<V>`

Returns:

the current title

titleProperty

```
public final ReadOnlyStringProperty titleProperty()
```

Description copied from interface: `Worker`
Gets the ReadOnlyStringProperty representing the title.

Specified by:

`titleProperty` in interface `Worker<V>`

Returns:

the property representing the current title

See Also:

`getTitle()`

cancel

```
public final boolean cancel()
```

Description copied from interface: `Worker`
Terminates execution of this Worker. Calling this method will either remove this Worker from the execution queue or stop execution.

Specified by:

`cancel` in interface `Worker<V>`

Returns:

returns true if the cancel was successful

cancel

```
public boolean cancel(boolean mayInterruptIfRunning)
```

Specified by:

`cancel` in interface `java.util.concurrent.Future<V>`

Overrides:

`cancel` in class `java.util.concurrent.FutureTask<V>`

updateProgress

```
protected void updateProgress(long workDone,
                  long max)
```

Updates the `workDone`, `totalWork`, and `progress` properties. Calls to updateProgress are coalesced and run later on the FX application thread, and calls to updateProgress, even from the FX Application thread, may not necessarily result in immediate updates to these properties, and intermediate workDone values may be coalesced to save on event notifications. `max` becomes the new value for `totalWork`.

This method is safe to be called from any thread.

Parameters:

`workDone` - A value from -1 up to max. If the value is greater than max, an illegal argument exception is thrown. If the value passed is -1, then the resulting percent done will be -1 (thus, indeterminate).

`max` - A value from -1 to Long.MAX_VALUE. Any value outside this range results in an IllegalArgumentException.

See Also:

`updateProgress(double, double)`

updateProgress

```
protected void updateProgress(double workDone,
                  double max)
```

Updates the `workDone`, `totalWork`, and `progress` properties. Calls to updateProgress are coalesced and run later on the FX application thread, and calls to updateProgress, even from the FX Application thread, may not necessarily result in immediate updates to these properties, and intermediate workDone values may be coalesced to save on event notifications. `max` becomes the new value for `totalWork`.

This method is safe to be called from any thread.

Parameters:

`workDone` - A value from -1 up to max. If the value is greater than max, an illegal argument exception is thrown. If the value passed is -1, then the resulting percent done will be -1 (thus, indeterminate).

`max` - A value from -1 to Double.MAX_VALUE. Any value outside this range results in an IllegalArgumentException.

Since:

2.2

updateMessage

```
protected void updateMessage(java.lang.String message)
```

Updates the `message` property. Calls to updateMessage are coalesced and run later on the FX application thread, so calls to updateMessage, even from the FX Application thread, may not necessarily result in immediate updates to this property, and intermediate message values may be coalesced to save on event notifications.

This method is safe to be called from any thread.

Parameters:

`message` - the new message

updateTitle

```
protected void updateTitle(java.lang.String title)
```

Updates the `title` property. Calls to updateTitle are coalesced and run later on the FX application thread, so calls to updateTitle, even from the FX Application thread, may not necessarily result in immediate updates to this property, and intermediate title values may be coalesced to save on event notifications.

This method is safe to be called from any thread.

Parameters:

`title` - the new title

addEventHandler

```
public final <T extends Event> void addEventHandler(EventType<T> eventType,
                                    EventHandler<? super T> eventHandler)
```

Registers an event handler to this task. Any event filters are first processed, then the specified onFoo event handlers, and finally any event handlers registered by this method. As with other events in the scene graph, if an event is consumed, it will not continue dispatching.

Type Parameters:

`T` - the specific event class of the handler

Parameters:

`eventType` - the type of the events to receive by the handler

`eventHandler` - the handler to register

Throws:

`java.lang.NullPointerException` - if the event type or handler is null

removeEventHandler

```
public final <T extends Event> void removeEventHandler(EventType<T> eventType,
                                       EventHandler<? super T> eventHandler)
```

Unregisters a previously registered event handler from this task. One handler might have been registered for different event types, so the caller needs to specify the particular event type from which to unregister the handler.

Type Parameters:

`T` - the specific event class of the handler

Parameters:

`eventType` - the event type from which to unregister

`eventHandler` - the handler to unregister

Throws:

`java.lang.NullPointerException` - if the event type or handler is null

addEventFilter

```
public final <T extends Event> void addEventFilter(EventType<T> eventType,
                                   EventHandler<? super T> eventFilter)
```

Registers an event filter to this task. Registered event filters get an event before any associated event handlers.

Type Parameters:

`T` - the specific event class of the filter

Parameters:

`eventType` - the type of the events to receive by the filter

`eventFilter` - the filter to register

Throws:

`java.lang.NullPointerException` - if the event type or filter is null

removeEventFilter

```
public final <T extends Event> void removeEventFilter(EventType<T> eventType,
                                      EventHandler<? super T> eventFilter)
```

Unregisters a previously registered event filter from this task. One filter might have been registered for different event types, so the caller needs to specify the particular event type from which to unregister the filter.

Type Parameters:

`T` - the specific event class of the filter

Parameters:

`eventType` - the event type from which to unregister

`eventFilter` - the filter to unregister

Throws:

`java.lang.NullPointerException` - if the event type or filter is null

setEventHandler

```
protected final <T extends Event> void setEventHandler(EventType<T> eventType,
                                       EventHandler<? super T> eventHandler)
```

Sets the handler to use for this event type. There can only be one such handler specified at a time. This handler is guaranteed to be called first. This is used for registering the user-defined onFoo event handlers.

Type Parameters:

`T` - the specific event class of the handler

Parameters:

`eventType` - the event type to associate with the given eventHandler

`eventHandler` - the handler to register, or null to unregister

Throws:

`java.lang.NullPointerException` - if the event type is null

fireEvent

```
public final void fireEvent(Event event)
```

Fires the specified event. Any event filter encountered will be notified and can consume the event. If not consumed by the filters, the event handlers on this task are notified. If these don't consume the event either, then all event handlers are called and can consume the event.

This method must be called on the FX user thread.

Parameters:

`event` - the event to fire

buildEventDispatchChain

```
public EventDispatchChain buildEventDispatchChain(EventDispatchChain tail)
```

Description copied from interface: `EventTarget`
Construct an event dispatch chain for this target. The event dispatch chain contains event dispatchers which might be interested in processing of events targeted at this `EventTarget`. This event target is not automatically added to the chain, so if it wants to process events, it needs to add an `EventDispatcher` for itself to the chain.

In the case the event target is part of some hierarchy, the chain for it is usually built from event dispatchers collected from the root of the hierarchy to the event target.

The event dispatch chain is constructed by modifications to the provided initial event dispatch chain. The returned chain should have the initial chain at its end so the dispatchers should be prepended to the initial chain.

The caller shouldn't assume that the initial chain remains unchanged nor that the returned value will reference a different chain.

Specified by:

`buildEventDispatchChain` in interface `EventTarget`

Parameters:

`tail` - the initial chain to build from

Returns:

the resulting event dispatch chain for this target