Multithreading in JAVA

1. **Czym jest wątek?**

Wątek jest niezależnym podprocesem, pozwalającym na wykrozystanie wielowątkowej architektury procesorów.

1. **Jaka jest rożnica między wątkiem, a procesem?**

Wątek jest „podzbiorem” procesu. Jeden proces może zawierać wiele wątków.

Dwa procesy mają przydzielony oddzielny obszar w pamięci, a wątki tego samego procesu współdzielą zasoby, (pliki, stos itd.), ale każdy z nich ma swój własny kawałek sterty i Exception handler.

Wątki są również często określane lekkimi procesami.

Wątki mogą się w łatwy sposób ze sobą komunikować poprzez konstrukcje takie jak *wait* lub *notify* w Javie.

Stworzenie wątku jest bardzo proste w porównaniu do tworzenia procesu.

1. **Jak implementuje się wątki w Javie?**

Istnieją dwa sposoby implementacji wątków w Javie.

Instancja klasy java.lang.Thread reprezentuje wątęk, ale potrzebuje również task’u do wykonania, który jest instancją interfejsu java.lang.Runnable. Jako, że klasa Thread implementuje Runnable, wystarczy nadpisać metodę *run()* poprzez rozrzeszenie klasy Thread lub implementację interfejsu Runnable.

1. **Kiedy używać Runnable, a kiedy Thread?**

Jest to związane z brakiem możliwości wielokrotnego dziedziczenia w Javie i z możliwością implementacji wielu interfejsów. Zatem lepiej jest implementować interfejs *Runnable*, gdy chce się rozszerzyć jeszcze jakąś inną klasę.

1. **Jaka jest różnica między metodami start() i run() klasy Thread?**

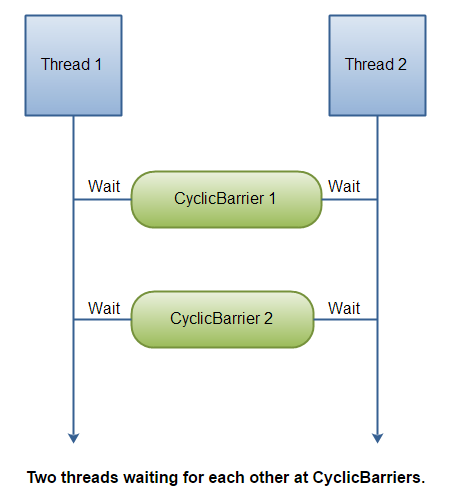
Metoda *start()* jest wykorzystywana do wystartowania nowo utworzonego wątku i *start()* wewnętrznie wywołuje metodę *run()*. Bezpośrednie wywołanie metody *run()* nie tworzy nowego wątku tylko wywołuje tę metodę w już istniejącym.

1. **Jaka jest różnica między interfejsem Runnable i Callable?**

Oba interfejsy reprezentują zadanie, która ma zostać uruchomione w oddzielnym wątku. *Runnable* istnieje od JDK 1.0, podczas gdy *Callable* zostało dodane w JDK 1.5. Główna różnica między nimi polega na tym, że metoda *call()* interfejsu *Callable* może zwrócić jakąś wartość i rzucić wyjątkiem, co nie jest możliwe z metodą *run()* interfejsu *Runnable*. *Callable* zwraca obiekt typu *Future*, ktory może przechowywać rezultat obliczeń.

1. **Jaka jest różnica między CyclicBarrtier i CountDownLatch?**

CyclicBarrier jest klasą zapewniającą mechanizm synchronizujący wątki zgodnie z zadanym algorytmem. Innymi słowy jest to bariera, przy której wszystkie wątki musza oczekiwać zanim ruszą dalej dopóki inne zadane wątki ją osiągną.



CountDownLatch jest konstrukcją pozwalającą jednemu lub wielu wątkom na oczekiwanie przez określony set operacji do zakończenia. CountDownLatch jest inicjalizowany z zadanym licznikiem, który następnie jest dekrementowany przez odwołanie do metody *countDown()*.

Obie klasy umożliwiają oczekiwanie przez określiną liczbę wątków na jeden lub więcej event’ów przed kontynuacją wykonania, a główna różnica między nimi jest taka, że *CountDownLatch* nie może być ponownie użyty po osiągnięciu przez licznik 0, a *CyclicBarrier* może być reużyty nawet po wykorzystaniu.

1. **Czym jest Java Memory Model?**

Jest to zestaw zasad i wytycznych, które pozwalają programom Javov’ym zachowywać się w sposób deterministyczny na różnych architekturach sprzętowych i systemowych. Szczególne znaczenie Java Memory Model ma w kontekście wielowątkowości. JMM daje nam gwarancję, jakie zmiany zrobione w jednym wątku będą widoczne dla innych wątków. Jedną z takich gwarancji jest zasada *happens-before relationship*. Ten związek definiuje kilka zasad, które pozwalają programistom przewidywać i wnioskować zachowanie wielowątkowych programów Javo’wych. Przykładowo związek *happens-before* zapewnia, że:

* **Program orfer rule -** Każda akcja w wątku wykonuje się przed akcją w tym wątku, który został zdefiniowany późnij w porządku programowym.
* **Monitor lock rule** – Odblokowanie blokady monitora dzieje się przed każdą wynikowa blokadą na tym samym monitorze.
* **Volatile variable rule** – zapis danych do pola oznaczonego jako *volatile* odbywa się przed każdym następnym odczytem wartości tego pola.
* **Thread start rule** – Odwołanie do *Thread.start()* na wątku wykonuje się zanim jakikolwiek inny wątek wykryje, że ten wątek się zakończył z sukcesem z *Thread.join()* lub *Thread.isAlive()* zwracając false.
* **Thread interruption rule** – Wywołanie metody *interrupt()* przez wątek na innym wątku odbywa się przed wykryciem przez przerywany wątek sygnału przerwania.
* **Finalizer rule** – Zakończenie wywołania konstruktora obiektu dzieje się przez rozpoczęciem wywołania *finalizer’a* tego obiektu.
* **Transivity rule** – Jeśli zdarzenie A dzieje się przed B i B dzieje się przed C to A dzieje się przed C.

1. **Co oznacza słowo kluczowe *volatile*?**

*Volatile* jest specjalnym modyfikatorem, który może być używany wyłącznie z *instance variable* (polami klas).

Normalnie w programach wielowątkowych zmiany wykonane na polach obiektów przez różne wątki nie są widoczne w innych wątkach, przy braku jakiegoś mechanizmu synchronizacji.

Słowo kluczowe *volatile* gwarantuje, że każdy zapis wartości związany z tą zmienną zostanie wykonany przed jakimkolwiek kolejnym odczytem. Wprowadza obowiązek synchronizacji wartości zmiennej z wartością z pamięci przy każdym odczycie/zapisie.

**11. Czym jest thread-safety? Czy wektor jest klasą thread-safe?**

Thread-safety jest własnością obiektu lub kodu, która gwarantuje, że jeśli zostanie on wykonany lub użyty przez wiele wątków jednocześnie jego zachowanie będzie przewidywalne. Dla przykładu thread-safe obiekt typu licznik będzie liczył w sposób przewidywalny, bez gubienia żadnej wartości jeśli ta sama instancja licznika będzie współdzielona z innym wątkiem.

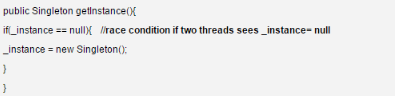
Kolekcje w javie również można podzielić na thread+safe i non-thread-safe. Vector jest rzeczywiście klasą thread-safe, a to bezpieczeństwo jest uzyskiwane przez synchronizację metod modyfikujących jego stan.

ArrayList nie jest thread-safe.

**12. Czym jest race condition w Javie? Podaj przykład.**

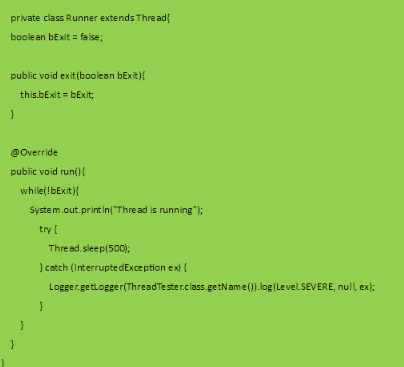
Race condition występuje jako efekt wyścigu między kilkoma wątkami. Jeśli wątek, który powinien wykonać się jako pierwszy przegra taki wyścig i wykona się jako drugi może to wpłynąć na wykonanie programu skutkując niedeterministycznym jego zachowaniem. Tzw.niedeterministyczne bugi.

Przykładem może być metoda getInstance() klasy reprezentującej Singleton.



**13. Jak zatrzymać wątek w Javie?**

W Javie nie ma skutecznego API do zatrzymywania wątków jednak wątek zostaje zatrzymany gdy zakończy się wykonanie metody *run()* zatem wystarczy w metodzie *run()* wykorzystać jakąś zewnętrzną flagę i uzależnić zakończenie wykonania metody *run()* od tej flagi.



**14. Co się dzieje gdy wewnątrz wątku zostaje rzucony wyjątek?**

Nieprzechwycony wyjątek umrze oraz jeśli uncaught exception handler jest zarejestrowany to wyjątek zostanie mu przekazany. *Thread.UncaughtExceptionHandler* jest interfejsem do obsługi nieobsłużonych wyjątków wewnątrz wątków. Gdy jakiś wątek ma zostać zakończony z powodu wystąpienia nieobslużonego wyjątku VMJ sprawdzi, czy ten wątek posiada zadeklarowany *UncaughtExceptionhandler* wykorzystując metodę *Thread.getUncaughtExceptionHandler()* i jeśli tak to wywoła metodę handlera *uncaughtException()* przekazując do niej dany wątek i parametry wyjątku.

**15. Jak należy współdzielić dane między dwoma wątkami?**

Jest to możliwe poprzez użycie wspólnego obiektu lub wielowątkowych struktur danych jak *BlockingQueue*. Przy czym należy pamiętać o prawidłowej synchronizacji takiego dostępu do zasobów.

**16. Jaka jest różnica między *notify()* i *notifyAll()*?**

Wiele wątków może oczekiwać na zwolnienie się jednej blokady na jakimś zasobie i dlatego API Javy zapewnia metodę do informowania tylko jednego lub wszystkich dostępnych wątków o zmianie warunku oczekiwania. Jednak metody *notify()* nie daje możliwości wyboru konkretnego wątku, który ma być poinformowany i dlatego jest użyteczna tylko, gdy jesteśmy pewni, że tylko jeden wątek oczekuje na zwolnienie blokady. Z drugiej strony metoda *notifyAll()* wysyła powiadomienie do wszystkich oczekujących wątków i pozwala im na konkurowanie o dostęp do zasobu, co daje gwarancje, żę przynajmniej jeden z wątku będzie mógł dalej kontynuować wykonanie.

**17. Dlaczego metody *wait()*, *notify()* i *notifyAll()* nie są wewnątrz klasy Thread?**

Jest to kwestia związana z projektowaniem klas i takim powodem może być fakt iż blokady są nakładane na obiekty, a nie na wątki. Każdy obiekt ma blokadę, która jest pozyskiwana przez wątek. Jeśli jakiś wątek ma oczekiwać na zwolnienie blokady jakiegoś obiektu, wywołanie metody *wait()* na tym obiekcie, a nie na wątku ma sens. Gdyby metoda *wait()* była zadeklarowana w klasie Thread, a nie *Object* mogłoby to być mało czytelne, którego obiektu dotyczy oczekiwanie. W skrócie w związku z tym, że wszystkie te metody operują na blokadzie, ich definicja na obiekcie ma sens bo blokada należy do obiektu.

**18. Co oznacza zmienna ThreadLocal?**

ThreadLocal variable jest specjalnym typem zmiennej. Zmienna ThreadLocal analogicznie do instance variable przynależącej do danej instancji przynależy do konkretnego wątku. Zmienna ta wymaga dużo pamięci dlatego nie nalezy jej stosować jako zmiennej lokalnej, a raczej jako zmienną przypisaną do instancji wątku. Dobrym przykładem wykoszytania może być np. *SimpleDateFormatter* zadeklarowany jako ThreadLocal.

**19. Co oznacza FutureTask?**

*FutureTask* reprezentuje możliwe do anulowania, asynchroniczne przetwarzanie danych w wielowątkowej aplikacji. Klasa ta zapewnia podstawową implementację interfejsu *Future* z metodami do rozpoczęcia, zatrzymania obliczeń, czy sprawdzenie czy przetwarzanie zostało ukończone. Jako, iż *FutureTask* implementuje również interfejs *Runnable* może być rownież przekazana do obiektu *Executor*.

**20. Jaka jest różnica między metodami *interrupted()* i *isInterrupted()?***

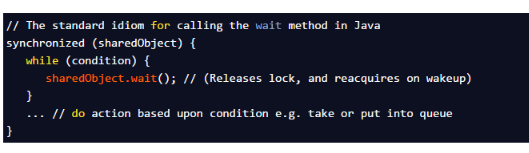
Mechanizm przerywania w wielowątkowych aplikacjach Javovych jest zaimplementowany poprzez wykorzystanie wewnętrznej flagi okreslanej jako interrupt status. Przerwanie obsługi wątku poprzez wywołanie metody *interrupt()* ustawia te flagę. Statyczna metoda *interrupted()* sprawdza status przerwania, jednocześnie czyszcząc flagę przerwania. Metoda niestatyczna *isInterrupted()* jest wykorzystywana przez inny wątek do sprawdzenia statusu przerwania innego wątku, bez zmiany stanu flagi przerwania. Zgodnie z konwencją każda metoda rzucająca wyjątkiem *InterruptedException* automatycznie czyści status przerwania.

**21. Dlaczego metody *wait()* oraz *notify()* są wywoływane wewnątrz bloku *synchronized*?**

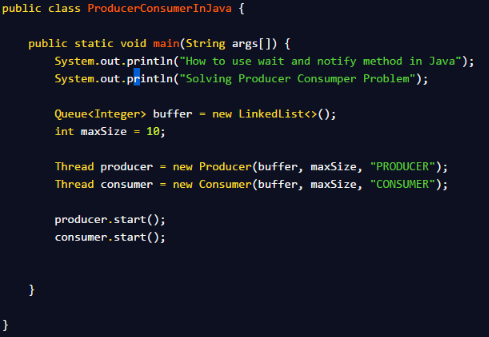
Jest to obowiązek nałożony przez API Javy. Próba wywołania tych metod spoza bloku *synchronized* zakończy się wyrzuceniem *IllegalMonitorStateException*. Innym powodem jest kwestia uniknięcia wyścigu *race condition* między metodami *wait()* i *notify().*

**22. Dlaczego powinno się sprawdzać warunek oczekiwania wewnątrz pętli?**

Metoda *wait()* powinna być wywoływana na wspoldzielonym obiekcie, wewnątrz bloku *synchronized* oraz wewnątrz pętli. Wywołanie tej metody wewnątrz pętli jest zalecane ponieważ istnieje możliwość nieoczekiwanego obudzenia się wątku nawet w przypadku niespełnienia warunku *if*. Dlatego należy powtarzać sprawdzanie warunku poprzez wykorzystanie pętli, aby niwelować efekt takiego nieoczekiwanego pobudzenia wątku.



Przykład Producer-Consumer:

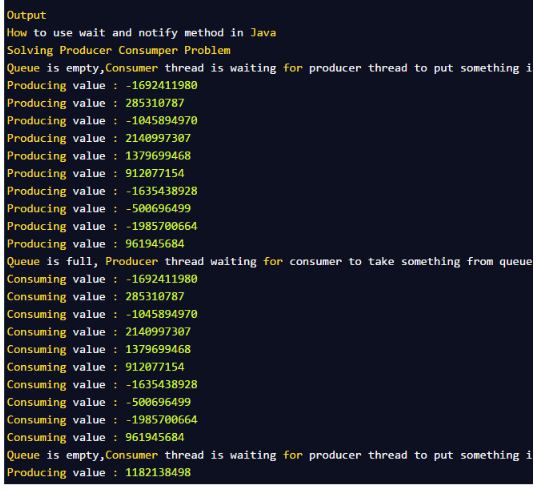






**23. Jaka jest różnica między kolekcjami *synchronized* i *concurrent*?**

Zarówno kolekcje *synchronized* jak i *concurrent* zapewniają implementacje thread-safe kolekcji, ale *synchronized* są mniej skalowalne od *synchronized*. Do Javy 1.5 programiście mieli dostępnejedynie kolekcje *synchronized* do zastosowań wielowątkowych. Java 5 wprowadziła kolekcję *concurrent* takie jak *ConcurrentHashMap*, które nie tylko zapewniają thread-safety, ale również poprawiają skalowalność poprzez wykorzystanie nowych technik takich jak *lock striping* lub partycjonowanie wewnętrznych tablic.



**24. Jaka jest różnica między stosem, a stertą w Javie?**

Stos jest wykorzystywany do przechowywania zmiennych lokalnych oraz wywołań funkcji (tak zwane ramki), podczas gdy na stercie są przechowywane wszystkie tworzone obiekty.

Każdy wątek w Javie posiada swój własny stos, którego wielkość może być ustawiona poprzez –Xss parametr JVM.

Jeśli zabraknie miejsca na stosie JVM rzuci java.lang.StackOverFlowError, a jeśli skończy się miejce na stercie java.lang.OutOfMemotyError.

Wielkość stosu jest przeważnie zdecydowanie mniejsza niż wielkość sterty.

Zmienne przechowywane na stosie są widoczne jedynie dla wątku, do którego ten stos należy, a obiekty na stercie są widoczne dla wszystkich wątków.

**25. Czym jest thread pool i dlaczego powinno się go używać w Javie?**

Tworzenie wątków jest kosztowne, zarówno pod względem czasu jak i zasobów. Tworzenie nowego wątku, zawsze gdy jest tam potrzebny może spowolnić czas między requestem, a odpowiedzią, co więcej istnieje ograniczenie dotyczące liczby wątków jakie jeden proces może stworzyć. Aby uniknąć tych problemów tworzony jest worek z wątkami podczas startu aplikacji i wątki są następnie reużywane do wykonywania konkretnych zadań. Ten worek z wątkami jest znany jako thread pool, a wątki określane są worker thread. Java API zapewnia Executor framework, pozwalający na tworzenie różnych rodzajów thread pool’i, np. Single thread pool, fixed thread pool (o określonej wielkości) oraz cached thread pool (rozszerzalny thread pool, dobry dla aplikacji z wieloma krótkimi taskami).

**26. Napisz kod rozwiązujący problem producer consumer.**

Problem ten można rozwiązać kalsycznie z wykorzystaniem metod *wait()* i *notify()* lub z wykorzystaniem struktury *BlockingQueue*, która zapewnia default’ową kontrolę poprzez metody *put()* i *take()*, które sprawdzają, czy kolejka jest pełna lub pusta i w razie czego blokują wywołanie przez odpowiedni wątek.

**http://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html**