Asserts (założenia/asercje)

Składnia:

//warunek powinien być spełniony

```
assert i % 3 == 2;

assert i % 3 == 2 : i; //dodatkowo zwraca i
```

Głównie służą do wykrywania ewentualnych bugów w kodzie. Używa się ich przy pewnych (tzn. które powinny wystąpić) założeniach, które mogą zostać niespełnione tylko przez bugi w oprogramowaniu. Jest tak dlatego, ponieważ *assert* z założenia uruchamiany jest w wersji testowej oprogramowania.

Aby włączyć(domyślnie są wyłączone) *assertion* należy uruchomić program z parametrem -ea (enable assertions)

Np.

java -ea program

asserty można włączyć także tylko w wybranych pakietach(zaaw. nawet w subpakietach) stosując zamiast

-ea -> -ea:nazwa_pakietu

Asserty można także wyłączyć(disableassertions -da) na takiej samej zasadzie np.:

Java -ea -da:nazwa_pakietu program //włączenie wszystkich *assertion* poza wybranym pakietem

Niezgodność twierdzenia z assertion skutkuje wywołaniem AssertionError!

Dlaczego assert, a nie try/catch?

Bloki try/catch powinny być używane do wyłapywania możliwych błędów spowodowanych przez przewidziane wyjątki

- np. jeżeli użytkownik podaje liczbę przez którą ma być podzielona liczba 2 i poda 0 – taki przypadek może wystąpić- możemy przewidzieć i opisać postepowanie programu w tym przypadku.

Assertion powinien być używany podczas testowania oprogramowania w ramach założeń, które program powinien spełniać, ale z powodu wewnętrznych bugów mogą nie wystąpić. W wersji finalnej(release) bloki typu *assertion* nie są włączane – nie powinny one mieć wpływu na działanie programu/obsługę wyjątków.

- np. kiedy w programie posiadamy instrukcje sprawdzające w kolejnych blokach *if* sprawdzają prawdziwość wyrażenia- wydaje się, że uwzględniamy każdy przypadek.

```
void cośzwracanego funkcja() {

    for (...) {
      if (...)
        return coś;
    }
         //teoretycznie tutaj kod nigdy nie powinien dojść
}
```

W powyższej funkcji powinno zostać użyte wyrażenie *assert*

```
void cośzwracanego funkcja() {
    for (...) {
      if (...)
        return coś;
    }
         assert false; //jeżeli program tutaj dojdzie zostaniemy poinformowani
}
```

W ten sposób, jeżeli wystąpi bug – funkcja przejdzie do bloku do którego nie powinna zajść, będziemy mogli namierzyć problem (program się zarumieni - AssertionError).

Używanie tego ma sens w dużych programach gdzie parametry przechodzą przez wiele funkcji i gdzie jest sens dzielić tworzenie oprogramowania na fazy *test i release* (przy własnych mniejszych projektach, gdzie przyczyny błędów są bardziej szczegółowe lub łatwe do zlokalizowania bloki *assertion* są mało przydatne – subiektywna opinia).

Dzienniki (Logger)

Służą do zapisu w pliku lub konsoli, aktualnego zachowania programu (każdy używał println(), aby zobaczyć co się dzieje w środku programu)

Konieczne jest zadeklarowanie loggera jako static final, ponieważ nie ma on powiązania z żadnymi obiektami dlatego w innej deklaracji zostanie on usunięty przez śmieciokolekcjonera.

```
    private static final Logger myLogger = Logger.getLogger("nazwa");
```

(…)

```
myLogger.info("utworzono okno");
```

(console)

```
maj 26, 2018 2:45:15 PM ruchharmoniczny.RHmain main
INFO: utworzono okno
```

7 typów komunikatów :

1. SEVERE
2. WARNING
3. INFO
4. CONFIG
5. FINE
6. FINER
7. FINEST

Każdy z poziomów posiada własne metody(dla każdego przypadku jest analogicznie) np.:

```
myLogger.info("utworzono okno");
```

=

```
myLogger.log(Level.INFO, "utworzono okno");
```

Domyślnie wszystkie poziomy poniżej info są tłumione(4-7) – Aby zmienić tłumienie należy zedytować plik *java.util.logging.config.file ->.level=INFO/FINE/…*(co tam się chce)- można przy wywołaniu lub specjalną komendą (w małych programach lepiej działać na domyślnych ustawieniach).

Dodatkowo istnieją metody logp(log precise) oraz logrb(log with resource bundle), które szczegółowiej opisują lokalizację wpisu. Istnieją także metody entering/exiting/throwing – wywoływane w szczegółowych miejscach.

Zwykli śmiertelnicy używają prostych wpisów.