	Wydział Politechniczny **Katedra Informatyki**
Prowadzący	Tomasz Gądek
Kurs	Narzędzia i środowiska programistyczne
Rok / Semestr	1 / Letni
Temat	Lab09: Git, praca zespołowa, rozwiązywanie konfliktów.

Data ostatniej modyfikacji: **01-10-2024**

Symulacja konfliktu

Twoje zadanie będzie polegało na utworzeniu nowego repozytorium **nisp_09** na platformie **Bitbucket**. Następnie utwórz nowy projekt w języku **Kotlin** oraz synchronizuj zmiany. Nie zapomnij o pliku .gitignore.

Zawartość pliku **Main.kt**

```
fun main() {
   println("Hello World!")
}
```

Zawartość pliku **.gitignore**

```
*.iml

.idea/

out/
```

Aby przyśpieszyć proces synchronizacji kodu z platformą podaje gotowe komendy (z opisem).

Tworzymy lokalne repozytorium.

```
git init
```

Tworzy nową gałąź develop i "przenosimy się" na nią.

```
git checkout -b develop
```

Dodanie pliku do poczekalni.

```
git add .
```

Zatwierdzanie zmian.

```
git commit -m "init"
```

Połącz lokalne i zdalne repozytorium.

```
git remote add...
```

Wyślij zmiany do zdalnego repozytorium.

```
git push origin develop
```

Przejdź do pliku **Main.kt** poprzez serwis **Bitbucket** i zmień ciąg tekstowy **Hello World!** na **Hello ANS!** Symulujemy przypadek, gdy programista dokona zmian na zdalnym repozytorium. Zatwierdź i zapisz zmiany.

Następnie przejdź do swojego lokalnego pliku **Main.kt** i zmień ciąg tekstowy **Hello World!** na **Hello AT!**.

Wykonaj polecenia:

```
git add .
git commit -m "Zmiana tresci komunikatu"
```

W tym momencie wersja na serwerze różni się od Twojej lokalnej wersji. Teraz spróbujemy pobrać zmiany z **Bitbucket** i sprawdzimy co takiego się wydarzy. Wykonaj polecenie.

```
git pull origin develop --no-rebase
```

Efekt jest następujący. Mój edytor kodu generuje taki rezultat.

```
fun main() {
<<<<<<< HEAD
    println("Hello AT!")
=======
    println("Hello ANS!")
>>>>>>> 7f011aaded96c3569b8e9a1f557be332024cb3f5
```

Nie przeraź się. Jest to normalne. **Mamy konflikt!** Gdy system kontroli wersji **Git** sobie nie poradzi to programista ma obowiązek zdecydować, które zmiany przyjąć.

Kolorem zielonym oznaczyłem zmiany w **HEAD**, czyli są to zmiany, które wprowadziłeś(aś) na swoim komputerze. Kolorem czerwonym oznaczyłem zmiany na zdalnym repozytorium. Ktoś przed Tobą dokonał zmian w tym samym fragmencie kodu, więc mamy konflikt.

Musisz zdecydować, który fragment zatwierdzić i usunąć wersję, która według ciebie jest niepotrzebna.

Moja propozycja jest następująca:

```
fun main() {
   println("Hello AT!")
}
```

Gdy już zmiany zostaną wprowadzone należy jeszcze raz dodać zmiany do poczekalni...

```
git add .
```

zatwierdź zmiany...

```
git commit –m "rozwiazanie konfliktu"
```

Następnie wypchnij zmiany do zdalnego repozytorium (push). Konflikt rozwiązujemy na branchu **develop**.

Praca zespołowa - pola i obwody figur geometrycznych

Zanim rozpoczniecie implementację dobierzcie się w zespoły składające się z 2 osób. Waszym zadaniem będzie wspólna implementacja klas w języku **Kotlin** reprezentujących figury geometryczne oraz sporządzenie dokumentacji technicznej kodu w pliku **README.md**

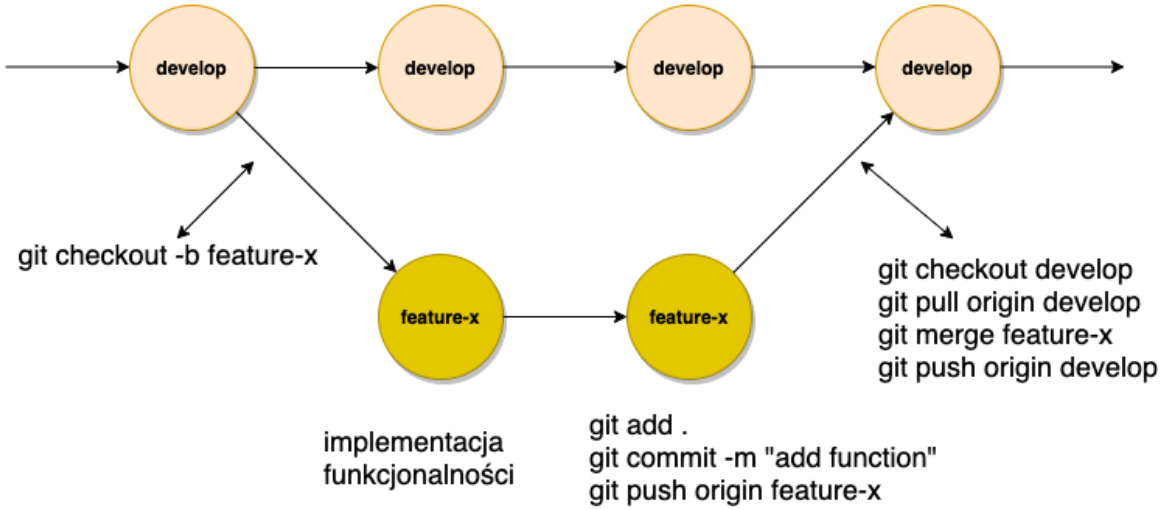
Zadania do wykonania:

- Implementacja klasy **Triangle** (konstruktor, metoda wyznaczająca pole figury, metoda wyznaczająca obwód figury).
- Implementacja klasy **Circle** (konstruktor, metoda wyznaczająca pole figury, metoda wyznaczająca obwód figury).
- Sporządzenie wspólnej dokumentacji technicznej w pliku **README.md**. Powinna zawierać opis istotnych fragmentów kodu (klasy, metody).

Dobrze przeanalizujcie schemat współpracy. Zaplanujcie swoją pracę i podzielcie się zadaniami.

Zapamiętajcie, że nowa funkcjonalność to nowa gałąź! Branch **develop** służy do synchronizacji pracy oraz rozwiązywania ewentualnych konfliktów.

x = nazwa funkcjonalności



Podczas wykonywania polecenia **git pull origin develop** proszę dodawać flagę **--no-rebase**.

Pomoc

- Film - Jak wygenerować tymczasowe hasło?
- Film - Jak uruchomić projekt w Kotlin po sklonowaniu?

Czy wiesz, że...

Pierwsze wzmianki o programie **Hello, World!** pochodzą z lat 70. XX wieku. Według niektórych źródeł, program ten po raz pierwszy został użyty w książce **A Tutorial Introduction to the Language B** autorstwa Briana Kernighana i Dennisa Ritchiego, opublikowanej w 1973 roku. Jednak najbardziej znana historia związana z **Hello, World!** dotyczy języka programowania C.

Inspiracją dla wyświetlenia akurat takiego komunikatu była dla B. Kernighana kreskówka, w której wykluwający się z jajka kurczak wypowiadał słowa **Hello, world!**

Pierwszy udokumentowany program **Hello world** opublikował Brian Kernighan w podręczniku do języka B z 1972 roku. Był to kod, który miał ilustrować wykorzystanie zmiennych globalnych.

```
main( ) {
    extrn a, b, c;
    putchar(a); putchar(b); putchar(c); putchar('!*n');
}

a 'hell';
b 'o, w';
c 'orld';
```