Uniwersytet Wrocławski

Rafał Lewandków

# PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

WYKŁAD 7

Coroutines:

- Kanały

# Kanały

Do tej pory nasze korutyny działały głównie **niezależnie**. Co się dzieje w momencie gdy jedna korutyna musi **wysłać dane** do drugiej? Na przykład, jedna korutyna produkuje dane (np. z sensora), a druga je konsumuje (np. zapisuje do pliku).

# Kanały

Do tej pory nasze korutyny działały głównie **niezależnie**. Co się dzieje w momencie gdy jedna korutyna musi **wysłać dane** do drugiej? Na przykład, jedna korutyna produkuje dane (np. z sensora), a druga je konsumuje (np. zapisuje do pliku).

**Channel** pozwala korutynom na bezpieczne **przesyłanie między sobą danych**. Działa jak **kolejka**, do której jedne korutyny mogą wysyłać elementy, a inne je odbierać.


https://medium.com/@mortitech/mastering-kotlin-channels-from-beginner-to-pro-part-1-7368060d1391

# Kanały

Do tej pory nasze korutyny działały głównie **niezależnie**. Co się dzieje w momencie gdy jedna korutyna musi **wysłać dane** do drugiej? Na przykład, jedna korutyna produkuje dane (np. z sensora), a druga je konsumuje (np. zapisuje do pliku).

**Channel** pozwala korutynom na bezpieczne **przesyłanie między sobą danych**. Działa jak **kolejka**, do której jedne korutyny mogą wysyłać elementy, a inne je odbierać.


Channel jest jak taśmociąg łączący dwa stanowiska pracy.

- **Producent** (jedna korutyna) kładzie produkty na taśmociąg (**channel.send()**).
- **Konsument** (druga korutyna) zdejmuje produkty z taśmociągu (**channel.receive()**).
- ***Taśmociąg*** rozdziela (decouples) producenta i konsumenta. Nie muszą oni wiedzieć o swoim istnieniu – komunikują się wyłącznie przez taśmociąg.

https://medium.com/@mortitech/mastering-kotlin-channels-from-beginner-to-pro-part-1-7368060d1391

# Channel vs Flow

Flow (**zimny**) to „przepis”. Produkcja danych startuje, gdy ktoś chce je skonsumować. To zazwyczaj relacja jeden-do-jednego.

Channel (**gorący)**, konkretny obiekt, który istnieje niezależnie. Może obsługiwać wielu producentów i wielu konsumentów.

| Cecha | **Flow** | **Channel** |
|---|---|---|
| **Czym jest?** | **Strumień danych** obliczanych w ramach **tego samego kawałka kodu** | **Kolejka do komunikacji** między różnymi korutynami |
| **Jak działa?** | Funkcja **emitująca wartości** sekwencyjnie, uruchamia się dopiero przy collect (lazy) | Jedna korutyna wysyła (send), inna odbiera (receive), kanał sam **danych nie generuje** |
| **Charakter** | **Zimny** – dane powstają dopiero, gdy ktoś je zbiera (collect) | **Gorący** – dane istnieją niezależnie od odbiorcy (kanał może się zapchać) |

Uniwersytet Wrocławski

# Channels

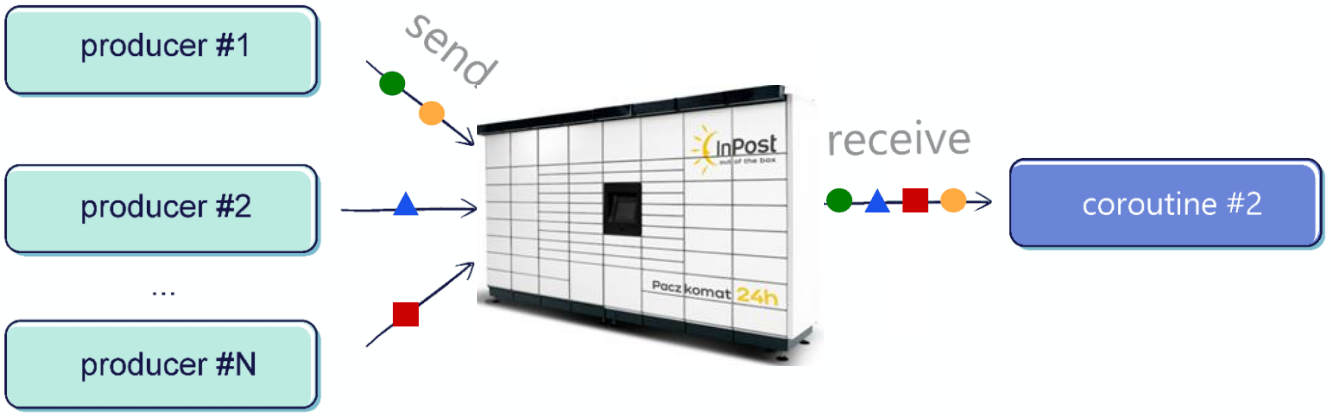
Wykład 7: Dema Channels

Wprowadzenie | Send/Receive | Buforowanie | Zamykanie

## 1. Wprowadzenie i Koncepcja

Channel działa jak taśmociąg między korutynami. Producent kładzie element (`send`), a Konsument go zdejmuje (`receive`).

Producent: Wyślij | Konsument: Odbierz



Uniwersytet Wrocławski

# Send, receive

Channel który może **transportować tylko i wyłącznie** obiekty typu String.

```
@Composable
fun IntroductionDemo() {
    val logs = remember { mutableStateListOf<String>() }
    val channel = remember { Channel<String>() }
    val scope = rememberCoroutineScope()

    DemoScreen( title = "1. Wprowadzenie i Koncepcja") {
        Text(...)
        Row(horizontalArrangement = Arrangement.SpaceEvenly,
            modifier = Modifier.fillMaxWidth()) {
            Button(onClick = {
                scope.launch {
                    logs.add("Producent: Kładę 'Produkt A' na taśmociąg...")
                    channel.send( element = "Produkt A")
                    logs.add("Producent: 'Produkt A' został odebrany.")
                }
            }) { Text( text = "Producent: Wyślij") }

            Button(onClick = {
                scope.launch {
                    logs.add("Konsument: Czekam na produkt...")
                    val product = channel.receive()
                    logs.add("Konsument: Odebrałem '$product'!")
                }
            }) { Text( text = "Konsument: Odbierz") }
        }
        LogDisplay(logs)
    }
}
```

Uniwersytet Wrocławski

# Send, receive

Channel który może **transportować tylko i wyłącznie** obiekty typu String.

channel.send(element): Funkcja suspend, która **wysyła** element do kanału. Jeśli kanał jest **pełny**, korutyna zostanie **zawieszona**, aż **zwolni się miejsce**.

```
@Composable
fun IntroductionDemo() {
    val logs = remember { mutableStateListOf<String>() }
    val channel = remember { Channel<String>() }
    val scope = rememberCoroutineScope()

    DemoScreen( title = "1. Wprowadzenie i Koncepcja") {
        Text(...)
        Row(horizontalArrangement = Arrangement.SpaceEvenly,
            modifier = Modifier.fillMaxWidth()) {
            Button(onClick = {
                scope.launch {
                    logs.add("Producent: Kładę 'Produkt A' na taśmociąg...")
                    channel.send( element = "Produkt A")
                    logs.add("Producent: 'Produkt A' został odebrany.")
                }
            }) { Text( text = "Producent: Wyślij") }

            Button(onClick = {
                scope.launch {
                    logs.add("Konsument: Czekam na produkt...")
                    val product = channel.receive()
                    logs.add("Konsument: Odebrałem '$product'!")
                }
            }) { Text( text = "Konsument: Odbierz") }
        }
        LogDisplay(logs)
    }
}
```

Uniwersytet Wrocławski

# Send, receive

Channel który może **transportować tylko i wyłącznie** obiekty typu String.

channel.send(element): Funkcja suspend, która **wysyła** element do kanału. Jeśli kanał jest **pełny**, korutyna zostanie **zawieszona**, aż **zwolni się miejsce**.

**channel.receive():** Funkcja suspend, która **odbiera** element z kanału. Jeśli kanał jest **pusty**, korutyna zostanie **zawieszona**, aż **pojawi się nowy element**.

```
@Composable
fun IntroductionDemo() {
    val logs = remember { mutableStateListOf<String>() }
    val channel = remember { Channel<String>() }
    val scope = rememberCoroutineScope()

    DemoScreen( title = "1. Wprowadzenie i Koncepcja") {
        Text(...)
        Row(horizontalArrangement = Arrangement.SpaceEvenly,
            modifier = Modifier.fillMaxWidth()) {
            Button(onClick = {
                scope.launch {
                    logs.add("Producent: Kładę 'Produkt A' na taśmociąg...")
                    channel.send( element = "Produkt A")
                    logs.add("Producent: 'Produkt A' został odebrany.")
                }
            }) { Text( text = "Producent: Wyślij") }

            Button(onClick = {
                scope.launch {
                    logs.add("Konsument: Czekam na produkt...")
                    val product = channel.receive()
                    logs.add("Konsument: Odebrałem '$product'!")
                }
            }) { Text( text = "Konsument: Odbierz") }
        }
        LogDisplay(logs)
    }
}
```

# Buforowanie

**Bufor** to **pamięć pośrednia** na elementy, które już zostały wyprodukowane, ale jeszcze nie zostały odebrane/przetworzone.

**Channel(capacity = N)**

•Gdy N = 0 (RENDEZVOUS): brak bufora → send i receive muszą się „zgrać” w czasie.

•Gdy N > 0: kanał może **przechować do N elementów**. send **nie wstrzyma się**, dopóki bufor się nie zapełni.

•Gdy bufor pełny:

- domyślnie send **zawiesza** korutynę (back-pressure),
- możesz zmienić zachowanie:
  **Channel(capacity = N, onBufferOverflow = DROP_OLDEST | DROP_LATEST | SUSPEND).**

•Specjalne warianty:

- **BUFFERED** – domyślny „rozsądny” rozmiar (zwykle kilkadziesiąt elementów),
- **CONFLATED** – trzyma **tylko najnowszą** wartość, starsze nadpisuje,
- **UNLIMITED** – bez górnego limitu.

# Zamykanie

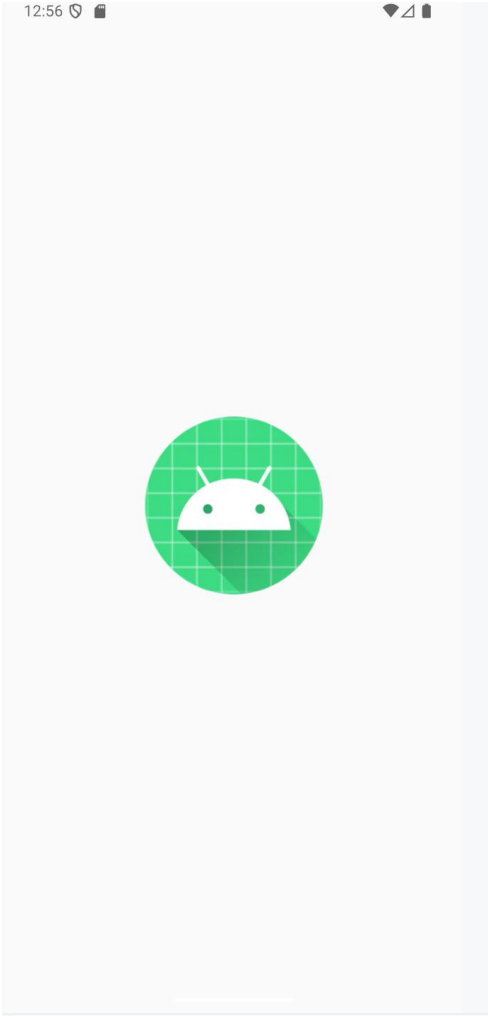
**Zamknięcie kanału** (channel.close()) to **sygnał** dla odbiorników o zakończeniu działania. **Buforowane** elementy nadal **można odebrać**. Gdy **ostatni** zostanie odczytany, kanał jest też **zamknięty dla odbioru**.

| Operacja | Co robi | Co z buforem |
|---|---|---|
| **close** | Kończy przyjmowanie nowych elementów | **Dostarcza** wszystko, co już w buforze |
| **cancel** | Natychmiastowa anulacja | **Porzuca** pozostałe elementy |

Uniwersytet Wrocławski

# select

select to wyrażenie, które pozwala korutynie **czekać** na **wiele operacji jednocześnie** i **wybrać** tę, która **zakończy** się jako **pierwsza**.



# select

Kanał na oferty od użytkowników

Kanał, który wyśle sygnał, gdy czas się skończy

```
class AuctionViewModel : ViewModel() {
    3 Usages
    var status by mutableStateOf( value = "Aukcja trwa...")
        private set

    2 Usages
    private val highBidChannel = Channel<String>()
    2 Usages
    private val timeoutChannel = Channel<Unit>()
    2 Usages
    fun startAuction() {...}
    1 Usage
    fun placeBid(bid: String) {...}
}
```

# select

Kanał na oferty od użytkowników

Kanał, który wyśle sygnał, gdy czas się skończy

Uruchamia w tle timer, który po 5 sekundach wyśle sygnał o końcu czasu.

```
class AuctionViewModel : ViewModel() {
    3 Usages
    var status by mutableStateOf( value = "Aukcja trwa...")
        private set

    2 Usages
    private val highBidChannel = Channel<String>()
    2 Usages
    private val timeoutChannel = Channel<Unit>()
    2 Usages
    fun startAuction() {...}
    1 Usage
    fun placeBid(bid: String) {...}
}
```

# select

```
class AuctionViewModel : ViewModel() {
    3 Usages
    var status by mutableStateOf( value = "Aukcja trwa...")
        private set

    2 Usages
    private val highBidChannel = Channel<String>()
    2 Usages
    private val timeoutChannel = Channel<Unit>()
    2 Usages
    fun startAuction() {...}
    1 Usage
    fun placeBid(bid: String) {...}
}
```

Kanał na oferty od użytkowników

Kanał, który wyśle sygnał, gdy czas się skończy

Uruchamia w tle timer, który po 5 sekundach wyśle sygnał o końcu czasu.

Ta funkcja pozwala na złożenie oferty. Jej jedynym zadaniem jest próba wysłania wiadomości (oferty) do kanału highBidChannel. Jeśli korutyna w startAuction wciąż nasłuchuje w bloku select, odbierze tę wiadomość, co natychmiast zakończy aukcję wygraną

Uniwersytet Wrocławski

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

**Wewnątrz** korutyny **możesz uruchamiać** kolejne, **zagnieżdżone** korutyny. One Wewnątrz korutyny możesz uruchamiać kolejne, zagnieżdżone korutyny. A**utomatycznie dziedziczą** scope po swoim rodzicu.

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

Uniwersytet Wrocławski

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

**Wewnątrz** korutyny **możesz uruchamiać** kolejne, **zagnieżdżone** korutyny. One Wewnątrz korutyny możesz uruchamiać kolejne, zagnieżdżone korutyny. A**utomatycznie dziedziczą** scope po swoim rodzicu.

Używamy dwóch korutyn, aby osiągnąć **współbieżność**. Chcemy, aby **dwie rzeczy** działy się **naraz**: Odliczanie czasu w tle, Czekanie na ofertę **lub** na koniec czasu

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

Uniwersytet Wrocławski

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

**Wewnątrz** korutyny **możesz uruchamiać** kolejne, **zagnieżdżone** korutyny. One Wewnątrz korutyny możesz uruchamiać kolejne, zagnieżdżone korutyny. A**utomatycznie dziedziczą** scope po swoim rodzicu.

Używamy dwóch korutyn, aby osiągnąć **współbieżność**. Chcemy, aby **dwie rzeczy** działy się **naraz**: Odliczanie czasu w tle, Czekanie na ofertę **lub** na koniec czasu

Funkcja suspend. Jeśli kanał **nie może natychmiast** przyjąć elementu (np. w kanale Rendezvous nie ma odbiorcy), send **zawiesza korutynę** i **czeka**, aż zwolni się miejsce.

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

**Wewnątrz** korutyny **możesz uruchamiać** kolejne, **zagnieżdżone** korutyny. One Wewnątrz korutyny możesz uruchamiać kolejne, zagnieżdżone korutyny. A**utomatycznie dziedziczą** scope po swoim rodzicu.

Używamy dwóch korutyn, aby osiągnąć **współbieżność**. Chcemy, aby **dwie rzeczy** działy się **naraz**: Odliczanie czasu w tle, Czekanie na ofertę **lub** na koniec czasu

Funkcja suspend. Jeśli kanał **nie może natychmiast** przyjąć elementu (np. w kanale Rendezvous nie ma odbiorcy), send **zawiesza korutynę** i **czeka**, aż zwolni się miejsce.

pozwala korutynie **czekać** na **wiele** operacji **jednocześnie** i **kontynuować** pracę z **wynikiem** tej, która zakończy się jako **pierwsza**

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

Uniwersytet Wrocławski

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

**Wewnątrz** korutyny **możesz uruchamiać** kolejne, **zagnieżdżone** korutyny. One Wewnątrz korutyny możesz uruchamiać kolejne, zagnieżdżone korutyny. A**utomatycznie dziedziczą** scope po swoim rodzicu.

Używamy dwóch korutyn, aby osiągnąć **współbieżność**. Chcemy, aby **dwie rzeczy** działy się **naraz**: Odliczanie czasu w tle, Czekanie na ofertę **lub** na koniec czasu

Funkcja suspend. Jeśli kanał **nie może natychmiast** przyjąć elementu (np. w kanale Rendezvous nie ma odbiorcy), send **zawiesza korutynę** i **czeka**, aż zwolni się miejsce.

pozwala korutynie **czekać** na **wiele** operacji **jednocześnie** i **kontynuować** pracę z **wynikiem** tej, która zakończy się jako **pierwsza**

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

Służy do zdefiniowania **jednej z możliwych** *gałęzi*, na którą select ma **czekać.**

Uniwersytet Wrocławski

# select

Uruchamia **główną** korutynę, która zarządza **całą logiką** aukcji. Jest ona naszym "głównym pracownikiem"

**Wewnątrz** korutyny **możesz uruchamiać** kolejne, **zagnieżdżone** korutyny. One Wewnątrz korutyny możesz uruchamiać kolejne, zagnieżdżone korutyny. A**utomatycznie dziedziczą** scope po swoim rodzicu.

Używamy dwóch korutyn, aby osiągnąć **współbieżność**. Chcemy, aby **dwie rzeczy** działy się **naraz**: Odliczanie czasu w tle, Czekanie na ofertę **lub** na koniec czasu

Funkcja suspend. Jeśli kanał **nie może natychmiast** przyjąć elementu (np. w kanale Rendezvous nie ma odbiorcy), send **zawiesza korutynę** i **czeka**, aż zwolni się miejsce.

pozwala korutynie **czekać** na **wiele** operacji **jednocześnie** i **kontynuować** pracę z **wynikiem** tej, która zakończy się jako **pierwsza**

Jeśli w kanale **jest miejsce** (lub czeka na niego **odbiorca** w select), trySend wysyła element i zwraca true. **Nie czeka** jeżeli nie ma odbiorcy – zwraca false

```
fun startAuction() {
    status = "Aukcja trwa... Oczekiwanie na oferty " +
            "lub koniec czasu."
    viewModelScope.launch {
        launch {
            delay( timeMillis = 5000)
            timeoutChannel.send( element = Unit)
        }

        val result = select<String> {
            highBidChannel.onReceive { offer ->
                "Aukcja zakończona! Wygrywa: $offer"
            }
            timeoutChannel.onReceive {
                "Aukcja zakończona! Czas upłynął."
            }
        }
        status = result
    }
}
1 Usage
fun placeBid(bid: String) {
    viewModelScope.launch {
        highBidChannel.trySend( element = bid)
    }
}
```

Służy do zdefiniowania **jednej z możliwych** *gałęzi*, na którą select ma **czekać.**

# select

select to wyrażenie w korutynach – pozwala jednej korutynie **czekać na wiele różnych operacji asynchronicznych jednocześnie** i kontynuuje działanie z wynikiem tej, która zakończy się jako **pierwsza**.

- **Współbieżne Oczekiwanie**: select nasłuchuje na wielu operacjach naraz (np. na kilku kanałach lub zadaniach async) bez potrzeby tworzenia wielu korutyn.
- **Wykonanie**: Gdy tylko jedna z monitorowanych operacji jest gotowa, select wykonuje jej gałąź kodu.
- **Automatyczne Anulowanie**: Po wybraniu *zwycięzcy*, wszystkie pozostałe, konkurujące operacje w bloku select są natychmiast **anulowane**. Gwarantuje to, że tylko jeden blok kodu zostanie wykonany.
- **Zwraca Wartość**: Jest to wyrażenie, co oznacza, że zwraca wartość, która jest wynikiem wykonania bloku.

# Podsumowanie

Kanały (Channels) to element, służący do **bezpiecznej komunikacji i przesyłania danych** między współbieżnymi zadaniami.

- **Komunikacja Między Korutynami**: Umożliwiają jednej korutynie wysyłanie (send) danych, a innej ich odbieranie (receive).
- **Synchronizacja**: Operacje send i receive są funkcjami suspend. Oznacza to, że:
  - Nadawca **czeka**, jeśli kanał jest pełny.
  - Odbiorca **czeka**, jeśli kanał jest pusty. To w naturalny sposób **synchronizuje** pracę obu stron.
- **Buforowanie**: Zachowanie kanału zależy od jego bufora:
  - **Rendezvous** (domyślny, bufor 0): Pełna synchronizacja. Nadawca i odbiorca muszą się *spotkać*, aby przekazać dane.
  - **Buffered**: Posiada bufor o określonej pojemności. Pozwala producentowi *wyprzedzić* konsumenta, dopóki bufor się nie zapełni.
  - **Conflated**: Bufor na jeden element, gdzie nowa wartość **zastępuje** starą, jeśli nie została jeszcze odebrana.
- **Zamykanie**: Producent może zamknąć kanał za pomocą channel.close(), aby zasygnalizować, że nie będzie już więcej wysyłał danych. Pozwala to konsumentowi na zakończenie pracy, np. poprzez pętlę for (element in channel).