В Kotlin существует разнообразие потоков данных.

Категоризация
1. Flow
2. Channel
3. Select

Channel

Интерфейс взаимодействия с данными в Kotlin, похожий на очередь в Java.
Так, один или множество источников информации могут отсылать одному или множеству потребителей данные через канал связи.
Channel построен через корутины, поэтому, при попытке отослать или получить данные не блокируют поток выполнения.
Каналы также могут быть разными (в зависимости от их настроек)

1. RENDEZVOUS - канал без буфера. Это означает что каналу некуда сохранять информацию и поэтому при отправке данных через метод send мы приостановим выполнение операции, пока на стороне потребителя эти данные не будут приняты через receive()
2. Буферизированный канал - этот канал обладает буфером, в котором хранится информация, еще не полученная потребителем. Размер буфера мы можем задавать сами (главное не меньше 0). Благодаря нему при отправке данных через send мы не приостанавливаем выполнение операции. Однако, если буфер будет заполнен целиком, то поведение будет схожим с RENDEZVOUS (приостановка до получения потребителем информации)
3. UNLIMITED - Канал с неограниченным буфером - канал обладает таким же поведением, как и второй, только его размер буфер ничем не ограничен (на самом деле ограничен, Int32.Max), поэтому отправка через send не будет приостанавливать выполнение
4. CONFLATED - канал, где старое значение, если оно хранится в буфере, будет перезаписано новым от нашего источника, поэтому операция send никогда не будет приостанавливать наш код

Поведение нашего канала определяется через int значение в конструкторе канала
RENDEZVOUS = 0
UNLIMITED = -1
CONFLATED = -2
Все что выше числа 0 это длина нашего буффера и мы задаем его сами (канал тип 2)
Все что ниже -2 будет выдавать ошибку

При работе с каналами мы можем использовать extension функции в CoroutineScope, которые являются функциями буилдерами каналов, предназначенных для отдельных задач

1. produce - канал-источник данных, имеет внутри себя функцию send, которая используется для отсылки данных, возвращает ReceiveChannel с методом receive (то есть внутри он как-то работает с данными и сохраняет в буфер через send, а мы потребляем через receive)
2. actor - канал-потребитель данных, обратен produce, функция строитель возвращает SendChannel (то есть внутри он берет данные через receive и как-то с ними работает, мы же ему посылаем данные через send)
3. ticker - аналогичен каналу-источнику данных, только здесь мы не контролируем отправку данных, задавая лишь настройки (сколько раз во времени мы будем получать событие, при этом стоит учесть что тут идет речь об внутреннем таймере канала, если мы не успеваем выполнить что-то до новой итерации, то при достижении receive() мы ждать не будем, операция продолжится сразу)

Select

Позволяет выбирать первое появившееся значение среди каналов. Можно представить в виде глобального обработчика, который принимает первое попавшееся ему значение (Например, двое сотрудников стремятся сдать свои отчеты первыми начальнику. Кто это сделает первым, тому будет счастье и хвала).
Примеры из области IT

1. Конкуренция между несколькими асинхронными операциями
   Например, у тебя есть два источника данных (API, БД, сенсоры), и ты хочешь взять результат того, который ответит первым.
2. Таймауты и отмена
   Можно ждать результат, но с ограничением по времени (если операция не успела — выполнить fallback).
3. Комбинирование каналов (Channels)
   Позволяет слушать несколько Channel или Flow и реагировать на первый пришедший элемент.

Select API делится на несколько интерфейсов и методов. Главный метод входа в API это метод select, который представляет собой suspend блок работающий в контексте с SelectBuilder. SelectBuilder содержит внутри себя extension функции, завязанные на интерфейсы типа:

- SelectClause0 - не выбирает никакое значение (не нашел применения или реализации, возможно что предполагалось как что-то где нам не важен результат, важно лишь то что выполнилось)

- SelectClause1 - работает с такими параметрами каналов как onReceive (то что принимает и обрабатывает)
- SelectClause2 - работает с такими параметрами каналов как onSend, где принимается отправляемый параметр и кэллбэк успешной отправки.

Для принятия значения существует два метода-конструктора:

1. select - выбирает первый полученный результат, при этом выбор сиещен к первым зарегистрированным (чем ближе к первому тем вероятней что он будет принят в качестве результата)
2. selectUnbiased - аналогично select выбирает первый полученный результат, но при этом выбор падает не на первые зарегистрированные, а случайно (в порядке рандома)

Также существуют экспериментальные конструкторы, которые под капотом содержат определенную логику

1. onTimeout - по истечении времени выбирается результат, если установленное время является нулевым или отрицательным, то результат выбирается сразу, без ожидания
2. whileSelect - значение выбирается каждый раз пока возвращает true

Если один из каналов в Select будет закрыт во время выбора, то нам придет ошибка ClosedReceiveChannelException
Аналогично методам onReceive, onSend существует onReceiveCatching/onSendCatching

Flows

Kotlin Flow API - специальное апи, основанное на последовательностях (sequence) и корутинах
Основная логика Flow состоит в том, что у нас есть определенная последовательность данных, которые мы либо не можем получить сразу (так как она может быть условно бесконечной), либо обработка всех этих данных сразу может занимать огромное кол-во времени.
Для работы с подобными вещами в Kotlin существует два вида последовательностей.
Sequence и Flow (еще Stream, но он больше завязан на Java).

Sequence в своей основе содержит тот же набор операция, что и Flow.
Он оперирует такими понятиями:

1. Промежуточные (intermediate) и терминальные (terminal) операции, где терминальными выступают функции, которые добавляют эффект к цепочке (например, фильтрация по условию, приведение к другому типу и т.д.), а

терминальные в свою очередь вызывают все эти операции вместе и предоставляют результат

2. Upstream и Downstream - где речь идет об влиянии операции. Например, если нам необходимо, чтобы наша операция влияла на данные, которые будут использоваться дальше (то есть в нижнем downstream потоке), то мы можем вызвать промежуточную операцию filter (отсечение элементов по предикату). Если же мы хотим, чтобы наши операции выполнялись в каком-то конкретном потоке (например, если речь идет об Flow API), то нам важно учесть что flowOn (операция, задающая Dispatcher выполнения) будет влиять только на операции сверху (upstream поток), нижние же операции (downstream) будут работать в вызывающем их потоке
3. Также последовательности можно поделить на горячие и холодные
    a. Холодные - операции в потоках будут запущены только при исполнении терминальной операции, то есть пока нет условного подписчика, то последовательность и не будет выполняться. Kotln Flow по-умолчанию является холодным, точно также речь идет и об Sequence
    b. Горячие - отличаются от холодных тем что им не надо ждать терминальной операции, они выполняются просто на том основании что существуют. Обычно в Kotlin Flow такие последовательности подразделяются на StateFlow и SharedFlow