**Внедрение зависимости** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Dependency injection*, DI) — процесс предоставления внешней зависимости [программному компоненту](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Является специфичной формой «[инверсии управления](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Inversion of control*, IoC), когда она применяется к управлению зависимостями. В полном соответствии с [принципом единственной обязанности](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8) объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму[[1]](https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8#cite_note-_2f16e3a9227f810c-1).

При использовании паттерна "внедрение зависимости" объект пассивен и не предпринимает вообще никаких шагов для выяснения зависимостей, а предоставляет для этого [сеттеры](https://ru.wikipedia.org/wiki/Setter) и/или принимает своим конструктором аргументы, посредством которых внедряются зависимости[[1]](https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8#cite_note-_2f16e3a9227f810c-1).

Работа фреймворка, обеспечивающая внедрение зависимости, описывается следующим образом. Приложение, независимо от оформления, исполняется внутри контейнера [IoC](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F" \o "Инверсия управления), предоставляемого фреймворком. Часть объектов в программе по-прежнему создается обычным способом языка программирования, часть создается контейнером на основе предоставленной ему конфигурации.

Условно, если [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) нужно получить доступ к определенному [сервису](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC)&action=edit&redlink=1), объект берет на себя обязанность по доступу к этому сервису: он или получает прямую [ссылку](https://ru.wikipedia.org/wiki/%D0%A1%D1%81%D1%8B%D0%BB%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) на местонахождение сервиса, или обращается к известному [«сервис-локатору»](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D1%82%D0%BE%D1%80_%D1%81%D0%BB%D1%83%D0%B6%D0%B1) и запрашивает ссылку на реализацию определенного типа сервиса. Используя же внедрение зависимости, объект просто предоставляет свойство, которое в состоянии хранить ссылку на нужный тип сервиса; и когда объект создается, ссылка на реализацию нужного типа сервиса автоматически вставляется в это свойство (поле), используя средства среды.

Внедрение зависимости более гибко, потому что становится легче создавать альтернативные реализации данного типа сервиса, а потом указывать, какая именно реализация должна быть использована в, например, [конфигурационном файле](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB_%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D1%83%D1%80%D0%B0%D1%86%D0%B8%D0%B8), без изменений в объектах, которые этот сервис используют. Это особенно полезно в [юнит-тестировании](https://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D1%82-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), потому что вставить реализацию «[заглушки](https://ru.wikipedia.org/wiki/Mock-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82)» сервиса в тестируемый объект очень просто.

С другой стороны, излишнее использование внедрения зависимостей может сделать приложения более сложными и трудными в сопровождении: так как для понимания поведения программы программисту необходимо смотреть не только в исходный код, а еще и в конфигурацию, а конфигурация, как правило, невидима для [IDE](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), которые поддерживают анализ ссылок и [рефакторинг](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), если явно не указана поддержка [фреймворков](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) с внедрениями зависимостей.

При использовании внедрения зависимостей, как правило, существует конфигурационный механизм или архитектура, которая определяет целесообразность выбора той или иной реализации в зависимости от поставленных целей.

**<service-point** id="CarBuilderService"**>**

**<invoke-factory>**

**<construct** class="Car"**>**

**<service>**DefaultCarImpl**</service>**

**<service>**DefaultEngineImpl**</service>**

**</construct>**

**</invoke-factory>**

**</service-point>**

*/\*\* Неявная реализация \*\*/*

**public** **class** **MyApplication** {

**public** **static** void main(String[] args) {

Service service = (Service)DependencyManager.get("CarBuilderService");

ICar car = (ICar)service.getService(Car.class);

car.setPedalPressure(5);

float speed = car.getSpeed();

System.out.println("Speed of the car is " + speed);}}

**Внедрение зависимостей в Android**

### Что такое внедрение зависимостей?

Классы часто требуют ссылок на другие классы. Например, Car классу может потребоваться ссылка на Engine класс. Эти обязательные классы называются зависимостями, и в этом примере Car класс зависит от наличия экземпляра Engine класса для запуска.

Жесткая зависимость от Engine затрудняет тестирование. Car использует реальный экземпляр Engine, что предотвращает использование [тестового двойника](https://en.wikipedia.org/wiki/Test_double) для изменения Engine для разных тестовых случаев.

Существуют библиотеки, которые решают эту проблему, автоматизируя процесс создания и предоставления зависимостей. Их можно разделить на две категории:

* Решения на основе отражения, которые связывают зависимости во время выполнения.
* Статические решения, которые генерируют код для подключения зависимостей во время компиляции.

[Dagger](https://dagger.dev/) - это популярная библиотека внедрения зависимостей для Java, Kotlin и Android, поддерживаемая Google. Dagger упрощает использование DI в вашем приложении, создавая и управляя графом зависимостей для вас. Он обеспечивает полностью статические зависимости и зависимости во время компиляции, решая многие проблемы разработки и производительности решений на основе отражения, таких как [Guice](https://en.wikipedia.org/wiki/Google_Guice) .

## Альтернативы внедрению зависимостей

Альтернативой внедрению зависимостей является использование [локатора служб](https://en.wikipedia.org/wiki/Service_locator_pattern) .  Шаблон проектирования локатора сервисов также улучшает отделение классов от конкретных зависимостей. Вы создаете класс, известный как локатор сервисов, который создает и хранит зависимости, а затем предоставляет эти зависимости по запросу.

Набор зависимостей, требуемых локатором сервисов, затрудняет тестирование кода, поскольку все тесты должны взаимодействовать с одним и тем же глобальным локатором сервисов.

## Используйте Hilt в своем приложении для Android

[Hilt](https://developer.android.com/training/dependency-injection/hilt-android) - это рекомендуемая библиотека Jetpack для внедрения зависимостей в Android. Hilt определяет стандартный способ выполнения DI в вашем приложении, предоставляя контейнеры для каждого класса Android в вашем проекте и автоматически управляя их жизненными циклами.

Hilt построен на основе популярной библиотеки DI [Dagger,](https://developer.android.com/training/dependency-injection/dagger-basics) чтобы извлечь выгоду из корректности времени компиляции, производительности во время выполнения, масштабируемости и поддержки Android Studio, которые предоставляет Dagger.

## Вывод

Внедрение зависимостей дает вашему приложению следующие преимущества:

* Возможность повторного использования классов и разделение зависимостей: проще поменять местами реализации зависимости. Повторное использование кода улучшено за счет инверсии управления, и классы больше не контролируют создание своих зависимостей, а вместо этого работают с любой конфигурацией.
* Легкость рефакторинга: зависимости становятся проверяемой частью поверхности API, поэтому их можно проверять во время создания объекта или во время компиляции, а не скрывать как детали реализации.
* Легкость тестирования: класс не управляет своими зависимостями, поэтому, когда вы его тестируете, вы можете передавать разные реализации для тестирования всех ваших различных случаев.

Чтобы полностью понять преимущества внедрения зависимостей, вы должны попробовать его вручную в своем приложении, как показано в [разделе Внедрение зависимостей вручную](https://developer.android.com/training/dependency-injection/manual) .

## Вывод

Внедрение зависимостей - хороший метод создания масштабируемых и тестируемых приложений для Android. Используйте контейнеры как способ совместного использования экземпляров классов в разных частях вашего приложения и как централизованное место для создания экземпляров классов с помощью фабрик.

Когда ваше приложение станет больше, вы начнете замечать, что пишете много шаблонного кода (например, фабрики), что может быть подвержено ошибкам. Вы также должны самостоятельно управлять областью действия и жизненным циклом контейнеров, оптимизируя и удаляя контейнеры, которые больше не нужны, чтобы освободить память. Неправильное выполнение этого может привести к незначительным ошибкам и утечкам памяти в вашем приложении.

В разделе [Dagger](https://developer.android.com/training/dependency-injection/dagger-basics) вы узнаете, как с помощью Dagger автоматизировать этот процесс и сгенерировать тот же код, который вы в противном случае написали бы вручную.

# Внедрение зависимостей с помощью Hilt

Hilt предоставляет стандартный способ использования DI в вашем приложении, предоставляя контейнеры для каждого класса Android в вашем проекте и автоматически управляя их жизненными циклами. Hilt построен на основе популярной библиотеки DI [Dagger,](https://developer.android.com/training/dependency-injection/dagger-basics) чтобы извлечь выгоду из корректности времени компиляции, производительности во время выполнения, масштабируемости и [поддержки Android Studio,](https://medium.com/androiddevelopers/dagger-navigation-support-in-android-studio-49aa5d149ec9) которые предоставляет Dagger. Для получения дополнительной информации см. [Рукоять и кинжал](https://developer.android.com/training/dependency-injection/hilt-android#hilt-and-dagger) .

В этом руководстве объясняются основные концепции Hilt и его созданных контейнеров. Он также включает демонстрацию того, как настроить существующее приложение для использования Hilt.

classpath 'com.google.dagger:hilt-android-gradle-plugin:2.28-alpha'

apply plugin: 'kotlin-kapt'  
apply plugin: 'dagger.hilt.android.plugin'  
implementation "com.google.dagger:hilt-android:2.28-alpha"  
kapt "com.google.dagger:hilt-android-compiler:2.28-alpha"

Все приложения, использующие Hilt, должны содержать Application класс с аннотацией @HiltAndroidApp.

## Внедрить зависимости в классы Android

После того, как Hilt настроен в вашем Application классе и доступен компонент уровня приложения, Hilt может предоставить зависимости другим классам Android, имеющим @AndroidEntryPoint аннотацию:

В настоящее время Hilt поддерживает следующие классы Android:

* Application(с помощью @HiltAndroidApp)
* Activity
* Fragment
* View
* Service
* BroadcastReceiver

Если вы аннотируете класс Android с помощью @AndroidEntryPoint, вы также должны аннотировать классы Android, которые зависят от него. Например, если вы аннотируете фрагмент, вы также должны аннотировать любые действия, в которых вы используете этот фрагмент.

@AndroidEntryPointгенерирует отдельный компонент Hilt для каждого класса Android в вашем проекте. Эти компоненты могут получать зависимости от своих соответствующих родительских классов, как описано в [разделе «Иерархия компонентов»](https://developer.android.com/training/dependency-injection/hilt-android#component-hierarchy) .

Чтобы получить зависимости от компонента, используйте @Injectаннотацию для внедрения поля:

@AndroidEntryPoint  
class ExampleActivity : AppCompatActivity() {  
  @Inject lateinit var analytics: AnalyticsAdapter  
  ...  
}

class AnalyticsAdapter @Inject constructor(  
  private val service: AnalyticsService  
) { ... }

### Внедрить экземпляры интерфейса с помощью @Binds

### Внедрить экземпляры с помощью @Provides

# Интеграция рукояти и Jetpack

Hilt включает расширения для предоставления классов из других библиотек Jetpack. В настоящее время Hilt поддерживает следующие компоненты Jetpack:

* ViewModel
* WorkManager

## Внедрение объектов ViewModel с помощью Hilt

implementation 'androidx.hilt:hilt-lifecycle-viewmodel:1.0.0-alpha02'  
  // When using Kotlin.  
  kapt 'androidx.hilt:hilt-compiler:1.0.0-alpha02'

class ExampleViewModel @ViewModelInject constructor(  
  private val repository: ExampleRepository,  
  @Assisted private val savedStateHandle: SavedStateHandle  
) : ViewModel() {  
  ...  
}

@AndroidEntryPoint  
class ExampleActivity : AppCompatActivity() {  
  private val exampleViewModel: ExampleViewModel by viewModels()  
  ...  
}

### Интеграция с навигационной библиотекой Jetpack

val viewModel: ExampleViewModel by navGraphViewModels(R.id.my\_graph) {  
  defaultViewModelProviderFactory  
}

## Ввести WorkManager с рукояткой

implementation 'androidx.hilt:hilt-work:1.0.0-alpha01'  
  // When using Kotlin.  
  kapt 'androidx.hilt:hilt-compiler:1.0.0-alpha01'

class ExampleWorker @WorkerInject constructor(  
  @Assisted appContext: Context,  
  @Assisted workerParams: WorkerParameters,  
  workerDependency: WorkerDependency  
) : Worker(appContext, workerParams) { ... }

@HiltAndroidApp  
class ExampleApplication : Application(), Configuration.Provider {  
  @Inject lateinit var workerFactory: HiltWorkerFactory  
  override fun getWorkManagerConfiguration() =  
      Configuration.Builder()  
            .setWorkerFactory(workerFactory)  
            .build()  
}

# Руководство по тестированию рукояти

## Модульные тесты

Hilt не требуется для модульных тестов, поскольку при тестировании класса, использующего внедрение конструктора, вам не нужно использовать Hilt для создания экземпляра этого класса. Вместо этого вы можете напрямую вызвать конструктор класса, передав поддельные или фиктивные зависимости, как если бы конструктор не был аннотирован:

@ActivityScoped  
class AnalyticsAdapter @Inject constructor(  
  private val service: AnalyticsService  
) { ... }  
  
class AnalyticsAdapterTest {  
  @Test  
  fun `Happy path`() {  
    // You don't need Hilt to create an instance of AnalyticsAdapter.  
    // You can pass a fake or mock AnalyticsService.  
    val adapter = AnalyticsAdapter(fakeAnalyticsService)  
    assertEquals(...)  
  }  
}

## Сквозные тесты

Для интеграционных тестов Hilt внедряет зависимости так же, как в ваш производственный код. Тестирование с Hilt не требует обслуживания, поскольку Hilt автоматически генерирует новый набор компонентов для каждого теста.

dependencies {  
    // For Robolectric tests.  
    testImplementation 'com.google.dagger:hilt-android-testing:2.28-alpha'  
    // ...with Kotlin.  
    kaptTest 'com.google.dagger:hilt-android-compiler:2.28-alpha'  
       // For instrumented tests.  
    androidTestImplementation 'com.google.dagger:hilt-android-testing:2.28-alpha'  
    // ...with Kotlin.  
    kaptAndroidTest 'com.google.dagger:hilt-android-compiler:2.28-alpha'  
   }

### Настройка теста пользовательского интерфейса

@HiltAndroidTest  
class SettingsActivityTest {  
  @get:Rule  
  var hiltRule = HiltAndroidRule(this)  
  // UI tests here.  
}

………………………………………………..