Declarations

Glossary

Entity
: A distinguishable part of a program

Path
: A sequence of names which identifies a program entity

Identifiers, names and paths

TODO(Explain paths)

Introduction

TODO(Examples)

Declarations in Kotlin are used to introduce entities (values, types, etc.); most declarations are *named*, i.e. they also assign an identifier to their own entity, however, some declarations may be *anonymous*.

Every declaration is accessible in a particular *scope*, which is dependent both on where the declaration is located and on the declaration itself.

Classifier declaration

Id grammar-rule-classDeclaration not found

Id grammar-rule-objectDeclaration not found

Classifier declarations introduce new types to the program, of the forms described [here][Classifier types]. There are three kinds of classifier declarations:

- class declarations;
- interface declarations;
- object declarations.

Class declaration

A simple class declaration consists of the following parts.

- name c;
- primary constructor declaration $ptor$;

- supertype specifiers $S_1, \ldots, S_s$;
- body b, which may include the following:
  - secondary constructor declarations $stor_1, \ldots, stor_c$;
  - instance initialization block $init$;
  - property declarations $prop_1, \ldots, prop_p$;
  - function declarations $md_1, \ldots, md_m$;
  - companion object declaration $companionObj$;
  - nested classifier declarations $nested$.

and creates a simple classifier type $c : S_1, \ldots, S_s$.

Supertype specifiers are used to create inheritance relation between the declared type and the specified supertype. You can use classes and interfaces as supertypes, but not objects.

It is allowed to inherit from a single class only, i.e., multiple class inheritance is not supported. Multiple interface inheritance is allowed.

Instance initialization block describes a block of code which should be executed during object creation.

Property and function declarations in the class body introduce their respective entities in this class' scope, meaning they are available only on an entity of the corresponding class.

Companion object declaration `companion object CO { ... }` for class `C` introduces an object, which is available under this class' name or under the path `C.CO`. Companion object name may be omitted, in which case it is considered to be equal to `Companion`.

Nested classifier declarations introduce new classifiers, available under this class' path for all nested classifiers except for inner classes. Inner classes are available only on the corresponding class' entities. Further details are available [here][Inner and nested classes].

TODO(Examples)

A parameterized class declaration consists of the following parts.

- name c
- type parameter list $T_1, \ldots, T_m$
- primary constructor declaration $ptor$
- supertype specifiers $S_1, \ldots, S_s$
- body b, which may include the following
  - secondary constructor declarations $stor_1, \ldots, stor_c$
  - instance initialization block $init$
  - property declarations $prop_1, \ldots, prop_p$
  - function declarations $md_1, \ldots, md_m$
  - companion object declaration $companionObj$

   - nested classifier declarations *nested*

and extends the rules for a simple class declaration w.r.t. type parameter list. Further details are described here.

Constructor declaration

There are two types of class constructors in Kotlin: primary and secondary.

A primary constructor is a concise way of describing class properties together with constructor parameters, and has the following form

$$ptor : (p_1, \ldots, p_n)$$

where each of p_i may be one of the following:

- regular constructor parameter $name : type$;
- read-only property constructor parameter `val`$name : type$;
- mutable property constructor parameter `val`$name : type$.

Property constructor parameters, together with being regular constructor parameters, also declare class properties of the same name and type. One can consider them to have the following syntactic expansion.

```
class Foo(i: Int, val d: Double, var s: String) : Super(i, d, s) {}

class Foo(i: Int, d_: Double, s_: String) : Super(i, d_, s_) {
  val d = d_
  var s = s_
}
```

When accessing property constructor parameters inside the class body, one works with their corresponding properties; however, when accessing them in the supertype specifier list (e.g., as an argument to a superclass constructor invocation), we see them as actual parameters, which cannot be changed.

If a class declaration has a primary constructor and also includes a class supertype specifier, that specifier must represent a valid invocation of the supertype constructor.

A secondary constructor describes an alternative way of creating a class instance and has only regular constructor parameters. If a class has a primary constructor, any secondary constructor must delegate to either the primary constructor or to another secondary constructor via `this(...)`.

If a class does not have a primary constructor, its secondary constructors must delegate to either the superclass constructor via `super(...)` (if the superclass is present in the supertype specifier list) or to another secondary constructor via `this(...)`. If the only superclass is `kotlin.Any`, delegation is optional.

In all cases, it is forbidden if two or more secondary constructors form a delegation loop.

TODO(elaborate this `this(...)` and `super(...)` business)

TODO(default values in constructors???)

Nested and inner classifiers

If a classifier declaration ND is *nested* in another classifier declaration PD, it creates a nested classifier type — a classifier type available under the path $PD.ND$. In all other aspects, nested classifiers are equivalent to regular ones.

Inner classes are a special kind of nested classifiers, which introduce types of objects associated (linked) with other (parent) objects. An inner class declaration ID nested in another classifier declaration PD may reference an *object* of type ID associated with it.

This association happens when instantiating an object of type ID, as its constructor may be invoked only when a receiver of type PD is available, and this receiver becomes associated with the new instantiated object of type ID.

TODO(...)

Inheritance delegation

In a classifier (an object or a class) C declaration any supertype I inheritance may be *delegated to* an arbitrary value v if:

- The supertype I is an interface type;
- v has type T such that $T <: I$.

The inheritance delegation uses a syntax similar to [property delegation][Property delegation] using the `by` keyword, but is specified in the classifier declaration header and is a very different concept. If inherited using delegation, each method M of I (whether they have a default implementation or not) is delegated to the corresponding method of v as if it was overriden in C with all the parameter values directly passed to the corresponding method in v, unless the body of C itself has a suitable override of M (see the method overriding section).

(TODO: link)

The particular means on how v is stored inside the classifier object is platform-defined.

Due to the initialization order of a classifier object, the expression used to construct v can not access any of the classifier object properties or methods excluding the parameters of the primary constructor.

TODO(...)

Data class declaration

A data class *dataClass* is a special kind of class, which represents a product type constructed from a number of data properties $(dp_1, \ldots, dp_m)$, described in its primary constructor. As such, it allows Kotlin to reduce the boilerplate and generate a number of additional data-relevant functions. Each one of these functions is generated if and only if a matching signature function is not present in the class body.

- `equals() / hashCode() / toString()` functions compliant with their contracts:
  - `equals(that)` returns true iff:
    - `that` has the same runtime type as `this`;
    - `this.prop.equals(that.prop)` returns `true` for every data property `prop`;
  - `hashCode()` returns different numbers for objects `A` and `B` if they do not equal by the generated `equals`;
  - `toString` returns a string representations which is guaranteed to include the class name along with all the data properties' string representations.
  - TODO(Be more specific?).
- A `copy()` function for shallow object copying with the following properties:
  - It has the same number of parameters as the primary constructor with the same names and types;
  - It calls the primary constructor with the corresponding parameters at the corresponding positions;
  - It has defaults for all the parameters defaulting to the value of the corresponding property in `this` object.
- A number of `componentN()` functions for destructive declaration:
  - For the data property at position N (**starting with 1**), the generated `component`N function has the same type as this property and returns the value of this property;
  - It has an `operator` modifier, allowing it to be used in [destructuring declarations][Destructuring declaration];
  - The number of these functions is the same as the number of data properties.

These generated declarations of `equals`, `hashCode` and `toString` may be overriden the same way they may be overriden in normal classes. The overriding version is preferred, as normally. In addition, for every other function, if any of the base types provide an open function with a matching signature, it is automatically overriden by the generated function as if it was generated with an `override` modifier.

> Note: base classes may also have functions that are either not open or have a conflicting signature with the same function name. As expected, these cases result in override or overload conflicts the same way they would do with a normal class declaration.

All these functions consider only data properties $\{dp_i\}$; e.g., your data class may include regular property declarations in its body, however, they will *not* be considered in the `equals()` implementation or have a `componentN()` generated for them.

Data classes have the following restrictions:

- Data classes are final and cannot be inherited from;
- Data classes must have a primary constructor with only property constructor parameters, which become data properties for the data class;
- There must be at least one data property in the primary constructor.

Data class generation

TODO(Do we really need this?)

TODO(A more detailed explaination)

Enum class declaration

TODO(grammar reference)

TODO(Use “enumeration” instead of “enum”?)

Enum class is a special kind of class with the following properties:

- It has a number of predefined values that are declared in the class itself (*enum entries*);
- No other values of this class can be constructed;
- It implicitly inherits the built-in class `kotlin.Enum` (and cannot have any other base classes);
- It it implicitly final and cannot be inherited from;
- It has special syntax to accommodate for the properties described above.

Enum class body uses special kind of syntax (see grammar) to declare enum entries in addition to all other declarations inside the class body. Enum entries have their own bodies that may contain their own declarations, similar to object declarations.

> Note: an enum class can have zero enum entries. This makes objects of this class impossible to construct.

In addition to this, every enum class has an implicit companion object declaration with the following member functions (in addition to the ones the object declaration specified explicitly has):

- `valueOf(value: String)` returning an object corresponding to the entry with the name equal to `value` parameter of the call;
- `values()` returning an [array][Array types] of all the possible enum values. Every invocation of this function returns a new array to disallow changing its contents.

> Note: Kotlin standard library introduces another function to access all enum values for a specific enum class called `kotlin.enumValues<T>`. Please refer to the standard library documentation for details.

TODO(`kotlin.Comparable` generation?)

TODO(. . .)

Annotation class declaration

Annotations class is a special kind of class that is used to declare [annotations][Annotations]. Annotation classes have the following properties:

- They cannot have any secondary constructors;
- All the primary constructor parameters must use the property syntax;
- They implicitly inherit `kotlin.Annotation` class (and cannot have any other base classes);
- They cannot implement interfaces;
- They are implicitly final and cannot be inherited from;
- They may not have any member functions, properties not declared in the primary constructor or any overriding declarations;
- They cannot have companion objects;
- They cannot have nested classes;
- The types of primary constructor parameters are limited to:
  - `kotlin.String`;
  - `kotlin.KClass`;
  - [Built-in number types][Built-in types];
  - Other annotation types;
  - Arrays of any other allowed type.

Annotation classes cannot be constructed directly, but their primary constructors are used when specifying [code annotations][Annotations] for other entities.

TODO(...)

Interface declaration

Interfaces differ from classes in that they cannot be directly instantiated in the program, they are meant as a way of describing a contract which should be satisfied by the interface's subtypes. In other aspects they are similar to classes, therefore we shall specify their declarations by specifying their differences from class declarations.

- An interface cannot have a class as its supertype;
- An interface cannot have a constructor;
- Interface properties cannot have initializers or backing fields;
- An interface cannot have inner classes (but can have nested classes and companion objects);
- An interface and all its members are implicitly open;
- All interface member properties and functions are implicitly public;
    - Trying to declare a non-public member property or function in an interface is an error.

TODO(Something else?)

Object declaration

Object declarations are used to support a singleton pattern and, thus, do two things at the same time. One, they (just like class declarations) introduce a new type to the program. Two, they create a singleton-like object of that type.

TODO(do we really need this ironic-ish statement about doing two things at the same time?)

Similarly to interfaces, we shall specify object declarations by highlighting their differences from class declarations.

- An object type cannot be used as a supertype for other types;
- An object cannot have a constructor;
- An object cannot have a companion object;
- An object may not have inner classes;
- An object cannot be parameterized, i.e., cannot have type parameters.

TODO(Something else?)

> Note: this section is about declaration of *named* objects. Kotlin also has a concept of *anonymous* objects, or object literals, which are similar to their named counterparts, but are expressions rather than declarations and, as such, are described in the [corresponding section][Object literals].

Classifier initialization

When creating a class or object instance via one of its constructors *ctor*, it is initialized in a particular order, which we describe here.

First, a supertype constructor corresponding to *ctor* is called with its respective parameters.

- If *ctor* is a primary constructor, a corresponding supertype constructor is the one from the supertype specifier list;
- If *ctor* is a secondary constructor, a corresponding supertype constructor is the one ending the constructor delegation chain of *ctor*;
- If an explicit supertype constructor is not available, `Any()` is implicitly used.

After the supertype initialization is done, we continue the initialization by processing each inner declaration in its body, *in the order of their inclusion in the body.* If any initialization step creates a loop, it is considered an undefined behavior.

When a classifier type is initialized using a particular secondary constructor *ctor* delegated to primary constructor *pctor* which, in turn, is delegated to the superclass constructor *sctor*, the following happens, in this order:

- *pctor* is invoked using the specified parameters, initializing all the properties declared by its property parameters in the order they appear in the constructor declaration;
- The superclass object (if any) is initialized as if created by invoking *sctor* with the specified parameters;
- Interface delegation expressions (if any) are invoked and the result of each is stored in the object to allow for interface delegation, in the order of appearance of delegation declarations in the classifier header;
- All the properties' initialization code as well as all the initialization blocks in the class body get initialized in the order of appearance in the class body;
- *ctor* body is invoked using the specified parameters.

> Note: this means that if an `init`-block appears between two property declarations in the class body, its body is invoked between the initialization code of these two properties.

This order stays the same if any of the entities involved are omitted, omitting the corresponding step (e.g. if there is no primary constructor, it is not invoked, and if the object is created using primary constructor, the body of the secondary one is not invoked, etc.), but performing all others. If any of the properties of the object are accessed before they are initialized in this order (for example, if a method called in an initialization block accesses a property that is mention after the block), the value of the property is undefined.

> Note: this can happen if a property is captured in a lambda expression that is used in some way during other initialization phases

TODO(This needs thorough testing)

Function declaration

Id grammar-rule-functionDeclaration not found

Id grammar-rule-functionBody not found

Function declarations assign names to functions — blocks of code which may be called by passing them a number of arguments. Functions have special *function types* which are covered in more detail [here][Function types].

A simple function declaration consists of four main parts:

- name f
- parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$
- return type R
- body b

and creates a function type $f : (P_1, \ldots, P_n) \rightarrow R$.

Parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$ describes function parameters — inputs needed to execute the declared function. Each parameter $p_i : P_i = v_i$ introduces p_i as a name of value with type P_i available inside function body b; therefore, parameters are final and cannot be changed inside the function. A function may have zero or more parameters.

A parameter may include a default value v_i, which is used if the corresponding argument is not specified in function invocation; v_i should be an expression which evaluates to type $V <: P_i$.

Return type R is optional, if function body b is present and may be inferred to have a valid type $B : B \not\equiv kotlin.Nothing$, in which case $R \equiv B$. In other cases return type R must be specified explicitly.

> As type $kotlin.Nothing$ has a [special meaning][`kotlin.Nothing`] in Kotlin type system, it must be specified explicitly, to avoid spurious $kotlin.Nothing$ function return types.

Function body b is optional; if it is ommited, a function declaration creates an *abstract* function, which does not have an implementation. This is allowed only inside an abstract classifier declaration. If a function body b is present, it should evaluate to type B which should satisfy $B <: R$.

TODO: `expect` and `external` functions also do not have implementations

A parameterized function declaration consists of five main parts.

- name f
- type parameter list $T_1, \ldots, T_m$
- parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$
- return type R
- body b

and extends the rules for a simple function declaration w.r.t. type parameter list. Further details are described here.

Named, positional and default parameters

Kotlin supports *named* parameters out-of-the-box, meaning one can bind an argument to a parameter in function invocation not by its position, but by its name, which is equal to the argument name.

```
fun bar(a: Int, b: Double, s: String): Double = a + b + s.toDouble()

fun main(args: Array<String>) {
    println(bar(b = 42.0, a = 5, s = "13"))
}
```

TODO(Argument names are resolved in compile time)

If one wants to mix named and positional arguments, the argument list must conform to the following form: $P_1, \ldots, P_M, N_1, \ldots, N_Q$, where P_i is a positional argument, N_j is a named argument; i.e., positional arguments must precede all of the named ones.

Kotlin also supports *default* parameters — parameters which have a default value used in function invocation, if the corresponding argument is missing. Note that default parameters cannot be used to provide a value for positional argument *in the middle* of the positional argument list; allowing this would create an ambiguity of which argument for position i is the correct one: explicit one provided by the developer or implicit one from the default value.

```
fun bar(a: Int = 1, b: Double = 42.0, s: String = "Hello"): Double =
    a + b + s.toDouble()
```

```
fun main(args: Array<String>) {
    // Valid call, all default parameters used
    println(bar())
    // Valid call, defaults for `b` and `s` used
    println(bar(2))
    // Valid call, default for `b` used
    println(bar(2, s = "Me"))

    // Invalid call, default for `b` cannot be used
    println(bar(2, "Me"))
}
```

In summary, argument list should have the following form:

- Zero or more positional arguments;
- Zero or more named arguments.

Missing arguments are bound to their default values, if they exist.

Variable length parameters

One of the parameters may be designated as being variable length (aka *vararg*). A parameter list $(p_1, \ldots, \text{vararg } p_i : P_i = v_i, \ldots, p_n)$ means a function may be called with any number of arguments in the i-th position. These arguments are represented inside function body b as a value p_i of type, which is the result of [*array type specialization*][Array types] of type `Array<out`P_i`>`.

If a variable length parameter is not last in the parameter list, all subsequent arguments in the function invocation should be specified as named arguments.

If a variable length parameter has a default value, it should be an expression which evaluates to a value of type, which is the result of [*array type specialization*][Array types] of type `Array<out`P_i`>`.

An array of type `Array<Q>`$<:$`ATS(Array<out`P_i`>)` may be *unpacked* to a variable length parameter in function invocation using [spread operator][Spread operator]; in this case array elements are considered to be separate arguments in the variable length parameter position.

> Note: this means that, for variable length parameters corresponding to specialized array types, unpacking is possible only for these specialized versions; for a variable length parameter of type `Int`, for example, unpacking is valid only for `IntArray`, and not for `Array<Int>`.

A function invocation may include several spread operator expressions corresponding to the vararg parameter.

Extension function declaration

An *extension function declaration* is similar to a standard function declaration, but introduces an additional special function parameter, the *receiver parameter*. This parameter is designated by specifying the receiver type (the type before `.` in function name), which becomes the type of this receiver parameter. This parameter is not named and must always be supplied (either explicitly or implicitly), e.g. it cannot be a variable-argument parameter, have a default value, etc.

Calling such a function is special because the receiver parameter is not supplied as an argument of the call, but as the [*receiver*][Receivers] of the call, be it implicit or explicit. This parameter is available inside the scope of the function as the implicit receiver or `this`-expression, while nested scopes may introduce additional receivers that take precedence over this one. See [the receiver section][Receivers] for details. This receiver is also available (as usual) in nested scope using labeled `this` syntax using the name of the declared function as the label.

For more information on how a particular receiver for each call is chosen, please refer to the [overloading section][Overload resolution].

> Note: when declaring extension functions inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested functions

For all other purposes, extension functions are not different from non-extension functions.

Examples:

```
fun Int.foo() { println(this + 1) } // this has type Int

fun main(args: Array<String>) {
    2.foo() // prints "3"
}

class Bar {
    fun foo() { println(this) } // this has type Bar
    fun Int.foo() { println(this) } // this has type Int
}
```

Property declaration

Id grammar-rule-propertyDeclaration not found

Property declarations are used to create read-only (`val`) or mutable (`var`) entities in their respective scope. Properties may also have custom getter or setter — functions which are used to read or write the property value.

Read-only property declaration

A read-only property declaration `val x: T = e` introduces `x` as a name of the result of `e`.

A read-only property declaration may include a custom getter in the form of

```
val x: T = e
    get() { ... }
```

in which case `x` is used as a synonym to the getter invocation. Both the right-hand value `e`, the type `T` and the getter are optional, however, at least one of them must be specified. More so, if both the type of `e` and the return type of the getter cannot be [inferred][Type inference] (or, in case of the getter, specified explicitely), the type `T` must be specified explicitly. In case both `e` and `T` are specified, the type of `e` must be a subtype of `T` (see [subtyping][Subtyping] for more details).

TODO: we never actually say how getters are similar/different to normal functions and, henceworth, how the inference works

The initializer expression `e`, if given, serves as the starting value for the property backing field (see getters and setters section for details) and is evaluated when the property is created. Properties that are not allowed to have backing fields (see getters and setters section for details) are also not allowed to have initializer expressions.

> Note: although a property with an initializer expression looks similar to an [assignment][Assignments], it is different in several key ways: first, a read-only property cannot be assigned, but may have an initializer expression; second, the initializer expression never invokes the property setter, but assigns the property backing field value directly.

Mutable property declaration

A mutable property declaration `var x: T = e` introduces `x` as a name of a mutable variable with type `T` and initial value equals to the result of `e`. The rules regarding the right-hand value `e` and the type `T` match those of a read-only property declaration.

A mutable property declaration may include a custom getter and/or custom setter in the form of

```
var x: T = e
    get(): TG { ... }
    set(value: TS) { ... }
```

in which case `x` is used as a synonym to the getter invocation when read from and to the setter invocation when written to.

Delegated property declaration

A delegated read-only property declaration `val x: T by e` introduces `x` as a name for the *delegation* result of property `x` to the entity `e`. One may view these properties as regular properties with a special *delegating* getters.

In case of a delegated read-only property, every access to such property (`x` in this case) becomes an [overloadable][Operator overloading] form which is expanded into the following:

```
e.getValue(thisRef, property)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `getValue` is a suitable operator function available on `e`;
- `thisRef` is the [receiver][Receivers] object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details).

A delegated mutable property declaration `var x: T by e` introduces `x` as a name of a mutable entity with type `T`, access to which is *delegated* to the entity `e`. As before, one may view these properties as regular properties with special *delegating* getters and setters.

Read access is handled the same way as for a delegated read-only property. Any write access to `x` (using, for example, an assignment operator `x = y`) becomes an overloadable form with the following expansion:

```
e.setValue(thisRef, property, y)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `getValue` is a suitable operator function available on `e`;
- `thisRef` is the [receiver][Receivers] object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details);
- `y` is the value `x` is assigned to. In case of complex assignments (see the [assignment][Assignments] section), as they are all overloadable forms, first

the assignment expansion is performed, and after that, the expansion of the delegated property using normal assignment.

An example on how the delegation expansion may be actually implemented by the compiler is as follows.

```
/*
 * Actual code
 */
class C {
    var prop: Type by DelegateExpression
}

/*
 * Expanded code
 */
class C {
    private val prop$delegate = DelegateExpression
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

The type of a delegated property may be omitted at the declaration site, meaning that it may be [inferred][Type inference] from the delegating function itself. If this type is omitted, it is inferred as if it was assigned the value of its expansion. If this inference fails, it is a compile-time error.

TODO(provideDelegate)

Local property declaration

If a property declaration is local, it creates a local entity which follows most of the same rules as the ones for regular property declarations. However, local property declarations cannot have custom getters or setters.

Local property declarations also support *destructive* declaration in the form of

```
val (a: T, b: U, c: V, ...) = e
```

which is a syntactic sugar for the following expansion

```
val a: T = e.component1()
val b: U = e.component2()
val c: V = e.component3()
...
```

where `componentN()` should be a valid operator function available on the result of `e`. Each individual component property follows the rules for regular local property declaration.

Getters and setters

As mentioned before, a property declaration may include a custom getter and/or custom setter (together called *accessors*) in the form of

```
var x: T = e
    get(): TG { ... }
    set(anyValidArgumentName: TS): RT { ... }
```

These functions have the following requirements

- $TG \equiv T$;
- $TS \equiv T$;
- $RT \equiv$ `kotlin.Unit`;
- Types TG, TS and RT are optional and may be omitted from the declaration;
- Read-only properties may have a custom getter, but not a custom setter;
- Mutable properties may have any combination of a custom getter and a custom setter
- Setter argument may have any valid identifier as argument name.

  Note: Regular coding convention recommends `value` as the name for the setter argument

One can also ommit the accessor body, in which case a *default* implementation is used (also known as default accessor).

```
var x: T = e
    get
    set
```

> This notation is usually used if you need to change some aspects of an accessor (i.e., its visibility) without changing the default implementation.

Getters and setters allow one to customize how the property is accessed, and may need access to the property's *backing field*, which is responsible for actually storing the property data. It is accessed via the special `field` property available inside accessor body, which follows these conventions

- For a property declaration of type `T`, `field` has the same type `T`
- `field` is read-only inside getter body

- `field` is mutable inside setter body

However, the backing field is created for a property only in the following cases

- A property has no custom accessors;
- A property has a default accessor;
- A property has a custom accessor, and it uses `field` property;
- A mutable property has a custom getter or setter, but not both/

In all other cases a property has no backing field. Properties without backing fields are not allowed to have initializer expressions.

Read/write access to the property is replaced with getter/setter invocation respectively.

Getters and setters allow for some modifiers available for function declarations (for example, they may be declared `inline`, see grammar for details).

Extension property declaration

An *extension property declaration* is similar to a standard property declaration, but, very much alike an extension function, introduces an additional parameter to the property called *the receiver parameter*. This is different from usual property declarations, that do not have any parameters. There are other differences from standard property declarations:

- Extension properties cannot have initializers;
- Extension properties cannot have backing fields;
- Extension properties cannot have default accessors.

> Note: informally, on can say that extension properties have no state of their own. Only properties that use other objects' storage facilities and/or uses constant data can be extension properties.

Aside from these differences, extension properties are similar to regular properties, but, when accessing such a property one always need to supply a [*receiver*][Receivers], implicit or explicit. Also, unlike regular properties, the type of the receiver must be a subtype of the receiver parameter, and the value that is supplied as the receiver is bound to the receiver parameter. For more information on how a particular receiver for each access is chosen, please refer to the [overloading section][Overload resolution].

The receiver parameter can be accessed inside getter and setter scopes of the property as the implicit receiver or `this`. It may also be accessed inside nested scopes using [labeled `this` syntax][This-expressions] using the name of the property declared as the label. For delegated properties, the value passed into the operator functions `getValue` and `setValue` as the receiver is the value of the receiver parameter, rather than the value of the outer classifier. This is also true for local extension properties: while regular local properties are passed `null` as the first argument of these operator functions, local extension properties are passed the value of the receiver argument instead.

> Note: when declaring extension properties inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested properties

For all other purposes, extension properties are not different from non-extension properties.

Examples:

```
val Int.foo: Int get() = this + 1

fun main(args: Array<String>) {
    println(2.foo.foo) // prints "4"
}

class Bar {
    val foo get() = this // returns type Bar
    val Int.foo get() = this // returns type Int
}
```

TODO(More examples (delegation, at least))

Property initialization

All non-abstract properties must be definitely initialized before their first use. To guarantee this, Kotlin compiler uses a number of analyses which are described in more detail [here][Control- and data-flow analysis].

TODO(maybe it makes more sense to write all the initialization business right here)

Constant properties

A property may be declared **constant**, meaning that its value is known during compilation, by using the special `const` modifier. In order to be declared `const`, a property must meet the following requirements:

- Its type is one of the following:
  - One of the [the built-in integral types][Built-in integer types];
  - `kotlin.Boolean`;
  - `kotlin.Char`;
  - `kotlin.String`;

- It is declared in the top-level scope or inside [an object declaration][Object declarations];
- It has an initializer expression and this initializer expression may be evaluated in the compile-time. Integer literals and string interpolation expressions without evaluated expressions, as well as builtin arithmetic/comparison operations and string concatenation operations on those are such expressions, but it is implementation-defined which other expressions qualify for this;
- It does not have getters, setters or delegation specifiers.

Type alias

Id grammar-rule-typeAlias not found

Type alias introduces an alternative name for the specified type and supports both simple and parameterized types. If type alias is parameterized, its type parameters must be [unbounded][Type parameters]. Another restriction is that recursive type aliases are forbidden — the type alias name cannot be used in its own right-hand side.

At the moment, Kotlin supports only top-level type aliases. The scope where it is accessible is defined by its [*visibility modifiers*][Visibility].

Declarations with type parameters

TODO()

Declaration modifiers

TODO(declaration scope)

TODO(`open`)

TODO(`abstract`)

TODO(`lateinit`)

TODO(`const`)

TODO(overriding vs overloading vs shadowing)

TODO(visibility)