**Type aliases**

Type aliases provide alternative names for existing types.

If the type name is too long you can introduce a different shorter name and use the new one instead.

It's useful to **shorten** **long** **generic** types.

For instance, it's often tempting **to shrink** **collection** **types**:

typealias NodeSet = **Set**<Network.Node>

typealias FileTable<**K**> = **MutableMap**<**K**, MutableList<File>>

You can provide different aliases for function types:

typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<**T**> = (**T**) -> Boolean

You can have new names for inner and nested classes:

class A {

inner class **Inner**

}

class B {

inner class **Inner**

}

typealias AInner = A.**Inner**

typealias BInner = B.**Inner**

Type aliases do not introduce new types.

They are equivalent to the corresponding underlying types.

When you add typealias Predicate<**T**> and use Predicate<**Int**> in your code, the Kotlin compiler always expands it to (**Int**) -> Boolean.

Thus you can pass a variable of your type whenever a general function type is **required** and vice versa:

typealias Predicate<**T**> = (**T**) -> Boolean

**fun** foo(p: Predicate<Int>) = p(42)

**fun** main() {

**val** f: (Int) -> Boolean = { it > 0 }

println(foo(f)) // prints "true"

**val** p: Predicate<Int> = { it > 0 }

println(listOf(1, -2).filter(p)) // prints "[1]"

}