7부 (파워툴)

1. 확장 람다

- va, vb 확장 함수, 확장 람다

```
val va: (String, Int) -> String = { str, n -> str.repeat(n) + str.repeat(n
val vb: String.(Int) -> String = { this.repeat(it) + repeat(it) }

fun main() {
    va("Vanbo", 2) eq "VanboVanboVanboVanbo"
    "Vanbo".vb(2) eq "VanboVanboVanboVanbo"
    vb("Vanbo", 2) eq "VanboVanboVanboVanbo"
    // "Vanbo".va(2) // 컴파일되지 않음
}
```

  - `va` 는 일반 람다
    - 두 파라미터로 `String` 과 `Int` 를 받고, `String` 을 반환한다.
    - 람다 본문에서도 `str`, `n` → 처럼 화살표 앞에 파라미터가 두 개 있다.
  - `vb`는 `String` 파라미터를 괄호 밖으로 옮겨서 `String.(Int)` 처럼 확장 함수 구문을 사용한다.
    - 확장 함수와 마찬가지로 확장 대상 객체가 수신 객체(여기선 `String`)이 되고, `this` 를 통해 수신 객체에 접근 할 수 있다.
    - `this.repeat(it)` 와 `repeat(it)` 형태로 호출이 가능하다. (다른 람다와 마찬가지로 파라미터가 하나뿐이면, 확장 함수에서도 it으로 그 유일한 파람미터를 가르킬 수 있다.)
    - 확장임으로 `“Vanvo”.vb(2)` 처럼 사용이 가능하고 함수 호출처럼도 가능하다.
      코

- 확장 함수와 마찬가지로 여러 파라미터를 받을 수 있는 확장 람다

```
val zero: Int.() -> Boolean = { this == 0 }
val one: Int.(Int) -> Boolean = { this $ it == 0 }
val two: Int,(Int, Int) -> Boolean = { arg1, arg2 -> this % (arg1 + arg2)

fun main() {
    0.zero() eq true
    10.one(10) eq true
    20.two(10,10) eq true
}
```

- 함수의 파라미터로 확장 람다가 사용되는 경우 (일반적으로 자주 사용되는 방식)

```
class A {
    fun af() = 1
}

class B {
    fun bf() = 2
}

fun f1(lambda: (A, B) -> Int) = lambda(A(), B())
fun f2(lambda: A.(B) -> Int) = A().lambda(B())

fun main() {
    f1 { aa, bb -> aa.af() + bb.bf() }
    f2 { af() + it.bf() }
}
```

  - `f2()`를 사용할 때 람다가 더 간결하다.

- 확장 람다의 반환 타입이 `Unit` 이면, 람다 본문이 만들어낸 결과는 무시된다.

```
fun unitReturn(lambda: A.() -> Unit) = A().lambda()
fun nonUnitReturn(lambda: A.() -> String) = A().lambda()

fun main() {
    unitReturn {
        "Unit ignores the return value" + "So it can be anything ..."
    }
    unitReturn { 1 } // ... 임의의 타입 ...
    unitReturn { } // ... 아무 값도 만들지 않는 경우
    nonUnitReturn { "Must return the proper type" }
    // nonUnitReturn { } // 이렇게 쓸 수 없음
}
```

- 일반 람다를 파라미터로 받는 위치에 확장 람다로 전달할 수 있다.
  두 람다의 파라미터 목록이 서로 호환되어야 한다.

```
fun String.transform1(n: Int, lambda: (String, Int) -> String) = lambda(th
fun String.transform2(n: Int, lambda: String.(Int) -> String) = lambda(thi
val duplicate: String.(Int) -> String = { repeat(it) }
val alternate: String.(Int) -> String = {
    toCharArray()
    .filterIndexed { i, _ -> i % it == 0 }
    .joinToString("")
}

fun main() {
    "hello".transform1(5, duplicate)
        .transform2(3, alternate) eq "hleolhleo"
```

```
    "hello".transform2(5, duplicate)
        .transform1(3, alternate) eq "hleolhleo"
}
```

  - `transform1()` 일반 람다를 인자로 받지만, `transform2()` 는 확장 람다를 인자로 받는다.

- `::` 을 사용하면 확장 람다가 필요한 곳에 함수 참조로 넘길 수 있다.

```
fun Int.d1(f: (Int) -> Int) = f(this) * 10
fun Int.d2(f: Int.() -> Int) = f() * 10
fun f1(n: Int) = n + 3
fun Int.f2() = this + 3

fun main() {
    74.d1(::f1) eq 770
    74.d2(::f2) eq 770
    74.d1(Int::f2) eq 770
    74.d2(Int::f2) eq 770
}
```

  - 확장 함수에 대한 참조는 확장 람다와 타입이 같다. (`Int::f2` 는 `Int.() → Int` 다)
  - `74.d1(Int:f2)` 호출에서는 일반 람다 파라미터를 요구하는 `d1()` 에 확장 함수를 넘긴다.

- 일반 확장 함수와 확장 람다 모두에서 다형성

```
open class Base {
  open fun f() = 1
}

class Derived : Base() {
  override fun f() = 99
}

fun Base.g() = f()

fun Base.h(xl: Base.() -> Int) = xl()

fun main() {
  val b: Base = Derived() // Upcast
  b.g() eq 99
  b.h { f() } eq 99
}
```

- 익명 확장 함수

```
fun exec(
  arg1: Int, arg2: Int,
  f: Int.(Int) -> Boolean
) = arg1.f(arg2)
```

```
fun main() {
  exec(10, 2, fun Int.(d: Int): Boolean {
    return this % d == 0
  }) eq true
}
```

  - `main()` 에서 `exec()` 호출은 익명 확장 함수를 익명 람다 위치에 써도 된다

- 코틀린 표준 라이브러리에서 확장 람다와 사용되는 경우

```
// String Builder 사용
private fun messy(): String {
  val built = StringBuilder()      // [1]
  built.append("ABCs: ")
  ('a'..'x').forEach { built.append(it) }
  return built.toString()          // [2]
}
// buildString()은 확장 람다를 인자로 받는다. 자체적으로 StringBuilder 객체를 생성하
// 확장 람다를 생성한 StringBuilder 객체에 적용한 다음, toString() 을 호출해 문자열을
private fun clean() = buildString {
  append("ABCs: ")
  ('a'..'x').forEach { append(it) }
}

private fun cleaner() =
  ('a'..'x').joinToString("", "ABCs: ")

fun main() {
  messy() eq "ABCs: abcdefghijklmnopqrstuvwx"
  messy() eq clean()
  clean() eq cleaner()
}
```

  - `messy()` 안에서는 `StringBuilder` 를 생성하고 결과를 얻어내야 한다.
  - `clean()` 에서 `buildString()` 을 사용할 때는 `append()` 호출 수신 객체를 직접 만들고 관리할 필요가 없다. 따라서 더 간결 해진다.

- `List, Map` 을 만들애는 코틀린 표준 라이브러리 함수

```
val characters: List<String> = buildList {
  add("Chars:")
  ('a'..'d').forEach { add("$it") }
}

val charmap: Map<Char, Int> = buildMap {
  ('A'..'F').forEachIndexed { n, ch ->
    put(ch, n)
  }
```

```
}

fun main() {
  characters eq "[Chars:, a, b, c, d]"
  //  characters eq characters2
  charmap eq "{A=0, B=1, C=2, D=3, E=4, F=5}"
}
```

- 빌더 패턴은 lombok 을 이용할 것이다. 그래도 장점을 나열하자면
  - 여러 단계를 거쳐 객체를 생성한다.
    객체 생성을 여러 단계로 나누면 객체 생성이 복잡할 때 유용할 수 있다.
  - 동일한 기본 생성 코드를 사용해 다양한 조합의 객체를 생성할 수 있다.
  - 공통 생성 코드와 특화된 코드를 분리할 수 있다.
    이를 통해 객체들의 변종 유형에 따른 코드를 더 쉽게 작성, 더 쉽게 읽을 수 있다.
  - 확장 람다를 사용해 빌더를 구현하면 DSL(도메인 특화 언어)를 만들기 쉽다.
  - 빌더 예시

```
open class Recipe : ArrayList<RecipeUnit>()

open class RecipeUnit {
  override fun toString() =
    "${this::class.simpleName}"
}

open class Operation : RecipeUnit()
class Toast : Operation()
class Grill : Operation()
class Cut : Operation()

open class Ingredient : RecipeUnit()
class Bread : Ingredient()
class PeanutButter : Ingredient()
class GrapeJelly : Ingredient()
class Ham : Ingredient()
class Swiss : Ingredient()
class Mustard : Ingredient()

open class Sandwich : Recipe() {
  fun action(op: Operation): Sandwich {
    add(op)
    return this
  }
  fun grill() = action(Grill())
  fun toast() = action(Toast())
  fun cut() = action(Cut())
}
```

```
fun sandwich(
  fillings: Sandwich.() -> Unit
): Sandwich {
  val sandwich = Sandwich()
  sandwich.add(Bread())
  sandwich.toast()
  sandwich.fillings()
  sandwich.cut()
  return sandwich
}

fun main() {
  val pbj = sandwich {
    add(PeanutButter())
    add(GrapeJelly())
  }
  val hamAndSwiss = sandwich {
    add(Ham())
    add(Swiss())
    add(Mustard())
    grill()
  }
  pbj eq "[Bread, Toast, PeanutButter, " +
    "GrapeJelly, Cut]"
  hamAndSwiss eq "[Bread, Toast, Ham, " +
    "Swiss, Mustard, Grill, Cut]"
}
```

2. 영역 함수

- 객체의 이름을 사용하지 않아도 그 객체에 접근할 수 있는 임시 영역을 만들어주는 함수
- 오로지 코드를 더 간결하고 읽기 좋게 만들기 위해 존재한다.
- `let()`, `run()`, `with()`, `apply()`, `also()`
  - 각각 람다와 함께 쓰이고 import 필요 없음.
  - 각 영역 함수는 **문맥 객체**를 `it` 으로 다루는지 혹은 `this` 로 다루는지와 각 함수가 어떤 값을 반환하는지에 따라 달라진다.
  - `with` 만 나머지 4 함수와 다르게 다른 호출 문법을 쓴다.

```
data class Tag(var n: Int = 0) {
    var s: String = ""
    fun increment() = ++n
}

fun main() {
```

```
    // let(): 객체를 'it' 로 접근
    // 람다의 마지막 식의 값을 반환
    Tag(1).let {
        it.s = "let: ${it.n}"
        it.increment()
    } eq 2
    // let() 을 사용하면서 람다 인자에 이름을 붙인 경우
    Tag(2).let { tag ->
        tag.s = "let: ${tag.n}"
        tag.increment()
    } eq 3
    // run(): 객체를 'this'로 접근하고
    // 람다의 마지막 식의 값을 반환
    Tag(3).run {
        s = "run: $n"
        increment()
    } eq 4
    // with(): 객체를 'this'로 접근하고
    // 람다의 마지막 식을 반환한다.
    with(Tag(4)) {
        s = "with: $n"
        increment()
    } eq 5
    // apply(): 객체를 'this'로 접근하고
    // 변경된 객체를 다시 반환한다.
    Tag(5).apply {
        s = "apply: $n"
        increment()
    } eq "Tag(n=6)"
    // also(): 객체를 'it'로 접근하고
    // 변경된 객체를 다시 반환한다.
    Tag(6).also {
        it.n = "also: ${it.n}"
        increment()
    } eq "Tag(n=7)"
    // also() 을 사용하면서 람다 인자에 이름을 붙인 경우
    Tag(7).also { tag ->
        tag.n = "also: ${tag.n}"
        increment()
    } eq "Tag(n=8)"
}
```

	this 문맥 객체	it 문맥 객체
마지막 식의 값을 돌려줌	with(일반함수), run(확장함수)	let
수신 객체를 돌려줌	apply	also

- 문맥 객체를 `this`로 접근할 수 있는 영역 함수 (`run`, `with`, `apply`)

  - 문맥 객체를 `it`로 접근할 수 있는 영역 함수(`let`, `also`)
    람다에 이름을 붙일 수 있다.
  - 결과를 만드는 경우, 람다의 마지막 식의 값을 돌려주는 영역 함수(`let`, `run`, `with`)
  - (객체에 대한 호출)식을 연쇄적 사용하는 경우, 객체를 돌려주는 영역 함수(`apply`, `also`)

- 안전한 접근 연산자 ?. 을 사용하면 영역 함수를 널이 될 수 있는 수신 객체에도 적용할 수 있다.
  안전한 접근 연산자를 사용하면 수신 객체가
  null 이 아닌 경우에만 영역 함수가 호출된다.

```
fun gets(): String? = it (Random.nextBoolean()) "str!" else null

fun main() {
    gets()?.let {
        it.removeSuffix("!") + it.length
    }?.eq("str4")
}
```

  - `gets()`가 `null` 이 아닌 값을 반환하면 `let` 이 호출된다.

    `let` 에서 널이 될 수 없는 수신 객체는 람다 내부에서 널이 될 수 없는 `it` 이 된다.

- 문맥 객체에 대해 **안전한 접근 연산**을 적용하면 영역 함수 전 `null` 검사를 수행

  안전한 접근 연산을 사용하지 않으면 영역 함수 안에서 개별적으로 `null` 검사해야함

```
class Gnome(val name: String) {
    fun who() = "Gnome: $name"
}

fun whatGnome(gnome: Gnome?) {
    grome?.let { it.who() }
    grome.let { it?.who() }
    grome?.run { who() }
    grome.run { this?.who() }
    grome?.apply { who() }
    grome.apply { this?.who() }
    grome?.also { it.who() }
    grome.also { it?.who() }
    // with는 null 검사인 안전한 접근 연산을 적용할 수 없음
    with(gnome) { this?.who() }
}
```

- 엘비스 연산자 (?:) 로 왼 값이 null 이면 오른쪽 return 실행

```
data class Plumbus(var id: Int)

fun display(map: Map<String, Plumbus>) {
  trace("displaying $map")
```

```
  val pb1: Plumbus = map["main"]?.let {
    it.id += 10
    it
  } ?: return
  trace(pb1)

  val pb2: Plumbus? = map["main"]?.run {
    id += 9
    this
  }
  trace(pb2)

  val pb3: Plumbus? = map["main"]?.apply {
    id += 8
  }
  trace(pb3)

  val pb4: Plumbus? = map["main"]?.also {
    it.id += 7
  }
  trace(pb4)
}

fun main() {
  display(mapOf("main" to Plumbus(1)))
  display(mapOf("none" to Plumbus(2)))
  trace eq """
    displaying {main=Plumbus(id=1)}
    Plumbus(id=11)
    Plumbus(id=20)
    Plumbus(id=28)
    Plumbus(id=35)
    displaying {none=Plumbus(id=2)}
  """
}
```

- 영역 함수는 `use()` 처럼 자원 해제를 제공하지 못한다.

```
data class Blob(val id: Int) : AutoCloseable {
  override fun toString() = "Blob($id)"
  fun show() { trace("$this")}
  override fun close() = trace("Close $this")
}

fun main() {
  Blob(1).let { it.show() }
  Blob(2).run { show() }
  with(Blob(3)) { show() }
```

```
  Blob(4).apply { show() }
  Blob(5).also { it.show() }
  Blob(6).use { it.show() }
  Blob(7).use { it.run { show() } }
  // 명시적으로 close를 호출.
  Blob(8).apply { show() }.also { it.close() }
  Blob(9).also { it.show() }.apply { close() }
  Blob(10).apply { show() }.use {  }
  trace eq """
    Blob(1)
    Blob(2)
    Blob(3)
    Blob(4)
    Blob(5)
    Blob(6)
    Close Blob(6)
    Blob(7)
    Close Blob(7)
    Blob(8)
    Close Blob(8)
    Blob(9)
    Close Blob(9)
    Blob(10)
    Close Blob(10)
  """
}
```

- `use()` 는 `let()`, `also()` 와 비슷하게 람다 안에 `it` 를 써야한다.
  하지만,
  `use()` 는 람다에서 반환을 허용하지 않는다.
- 영역 함수를 사용하면서 자원해제를 보장하고 싶다면 `Blob(7)`의 경우처럼 영역 함수를 `use()` 람다 안에서 써라.

- **영역 함수는 인라인된다**
  - 일반적으로 람다를 인자로 전달하면, 람다 코드를 외부 객체에 넣기 때문에 일반 함수 호출에 비해 실행 시점의 부가 비용이 좀 더 발생한다.
  - 그래서, 권장 사항으로 영역 함수를 inline 으로 만들면 모든 실행 시점 부가 비용을 없앨 수 있다.
  - 컴파일러는 inline 함수 호출을 보면 함수 호출 식을 함수의 본문으로 치환하며, 이때 함수의 모든 파라미터를 실제 제공된 인자로 바꿔준다.

```
/* inline 예제 */

inline fun doSomething() {
   println("인라인")
   println("예제")
}
```

```
fun example() {
   doSomething()
}

/* 인라인 된 디컴파일 결과 */
fun example() {
   println("인라인")
   println("예제")
}


/* inline 예제 */

inline fun doSomething(body: () -> Unit) {
   body()
}

fun example() {
   doSomething {
      println("인라인 함수 예제")
   }
}

/* 인라인 된 디컴파일 결과 */
fun example() {
   println("인라인 함수 예제") //doSomething()이 인라인 됨
}
```

- 인라인 함수의 작동 방식
  - 일반적인 함수 호출은 함수의 코드로 점프하고, 함수의 실행이 끝나면 호출한 지점으로 돌아오는 과정을 포함합니다. 반면, 인라인 함수는 컴파일 시에 함수 호출 지점에 함수의 바디가 직접 복사되어 삽입됩니다. 이로 인해 함수 호출 과정이 사라져, 실행 시간이 단축될 수 있습니다.
- 인라인 함수의 장점
  - 성능 향상: 함수 호출에 따른 오버헤드가 줄어들어, 특히 루프 내부나 고차 함수에서 성능 이점을 볼 수 있습니다.타입 소거(Type Erasure) 회피: 제네릭을 사용하는 고차 함수에서 인라인 함수를 사용하면, 런타임에 타입 정보가 소거되는 것을 방지할 수 있습니다. 이는 제네릭 타입을 정확히 알고 있을 때 유용할 수 있습니다.

- 함수 호출 비용이 함수 전체 비용에서 큰 비중을 차지하는 작은 함수의 경우 인라인이 잘 작동한다.
  함수가 커질수록 전체 호출을 실행하는 데 걸리는 시간에서 함수 호출이 차지하는 비중이 줄어들기 때문에 인라인의 가치도 하락한다.
  게다가 함수가 크면 모든 함수 지점에 함수 본문이 삽입되므로 컴파일된 전체 바이트코드의 크기도 늘어난다.
- 인라인 함수가 람다를 인자로 받으면, 컴파일러는 인라인 함수의 본문과 함께 람다 본문을 인라인 해준다.
  따라서 인라인 함수에 람다를 전달하는 경우 클래스나 객체가 추가로 생기지 않는다 (이런 동작은 인라인 함수에 람다 리터널을 바로 전달하는 경우에만 성립한다. 람다를 변수에 담아서 전달하거나 다른 함수가 반환하는 람다를 전달하면 람다를 저장하기 위한 객체가 생긴다.)

3. 제네릭스 만들기

- 제네릭스는 나중에 지정할 타입에 대해 작동하는 코드를 말한다.
- 자바 제네릭스
- `Any`
  - 코틀린 클래스 계층의 root 뿌리 - Java 에서 Object
  - 모든 코틀린 클래스는 Any 를 상위 클래스로 가진다.
  - 미리 정해지지 않는 타입을 다루는 방법중 하나로 Any 타입을 전달하는 방법이 있고,
    이를 제네릭스를 사용해야 하는 경우와 혼동하기도 한다.
  - 사용하는 방법 2가지
    - `Any` 에 대해서만 연산 수행하고, 다른 어느 타입도 요구하지 않는 경우
      - 극히 제한적이다.

        `Any` 멤버함수는 `equlas()`, `hashCode()`, `toString()` 세 가지.
        확장 함수도 있지만, 이런 확장 함수는 Any 타입 객체에 대해 직접 연산을 적용할 수는 없다.
    - 어떤 `Any` 타입 객체의 실제 타입을 안다면 타입을 변환해서 구체적인 타입에 따른 연산을 수행할 수 있다.
      - 이 과정에서 런타임 타입 정보가 필요하므로, 여러분이 타입을 변환할 때 잘못된 타입을 지정하면 런타임 오류가 발생할 가능성이 있다.

```
class Person {
    fun speak() = "Hi!"
}

class Dog {
    fun bark() = "Ruff!"
}

class Robot {
    fun communicate() = "Beep!"
}

fun talk(speaker: Any) =    when (speaker) {
    is Person -> speaker.speak()
    is Dog -> speaker.bark()
    is Robot  -> speaker.communicate()
    else -> "Not a talker"
}

fun main() {
    talk(Person()) eq "Hi!"
    talk(Dog()) eq "Ruff!"
    talk(Robot()) eq "Beep!"
```

```
    talk(11) eq "Not a talker"
}
```

  - when 식으로 타입을 찾고 적절한 함수를 호출한다.
  - 하지만, 다른 type 이 추가되면 talk 함수를 수정해야한다.

- 제네릭스 정의하기

```
fun <T> gFunction(arg: T): T = arg

class GClass<T>(val x: T) {
  fun f(): T = x
}

class GMemberFunction {
  fun <T> f(arg: T): T = arg
}

interface GInterface<T> {
  val x: T
  fun f(): T
}

class GImplementation<T>(
  override val x: T
) : GInterface<T> {
  override fun f(): T = x
}

class ConcreteImplementation
  : GInterface<String> {
  override val x: String
    get() = "x"
  override fun f() = "f()"
}

fun basicGenerics() {
  gFunction("Yellow")
  gFunction(1)
  gFunction(Dog()).bark()            // [1]
  gFunction<Dog>(Dog()).bark()       // [2]

  GClass("Cyan").f()
  GClass(11).f()
  GClass(Dog()).f().bark()           // [1]
  GClass<Dog>(Dog()).f().bark()      // [2]

  GMemberFunction().f("Amber")
```

```
  GMemberFunction().f(111)
  GMemberFunction().f(Dog()).bark()  // [1]
  GMemberFunction().f<Dog>(Dog()).bark() // [2]

  GImplementation("Cyan").f()
  GImplementation(11).f()
  GImplementation(Dog()).f().bark()

  ConcreteImplementation().f()
  ConcreteImplementation().x
}
```

  - [1] 은 결과 타입이 Dog 라는 타입으로 결정된다. (컴파일러 추론 오류 가능성 有)
  - [2] 는 구체적인 타입을 지정해서 타입 추론을 결정해준다.

- 타입 정보 보존하기
  - 제네릭 클래스나 제네릭 함수의 내부 코드 T 타입에 대해 알 수 있는 방법은 없다. (타입 소거)
    - `Car` 타입을 객체를 담는 `CarCreate`

```
class Car {
  override fun toString() = "Car"
}

class CarCrate(private var c: Car) {
  fun put(car: Car) { c = car }
  fun get(): Car = c
}

fun main() {
  val cc = CarCrate(Car())
  val car: Car = cc.get()
  car eq "Car"
}
```

    - 다른 타입도 담을 수 있는 `CarCreate` 에서 일반화된 클래스 `Create<T>`

```
open class Crate<T>(private var contents: T) {
  fun put(item: T) { contents = item }
  fun get(): T = contents
}

fun main() {
  val cc = Crate(Car())
  val car: Car = cc.get()
  car eq "Car"
}
```

    - 제네릭 확장 함수

```
fun <T, R> Crate<T>.map(f:(T) -> R): List<R> =
  listOf(f(get()))

fun main() {
  Crate(Car()).map { it.toString() + "x" } eq
    "[Carx]"
}
```

- 타입 파라미터 제약
    - 제네릭 타입 인자가 다른 클래스를 상속해야 한다고 지정한다.
    - `<T: Base>` 는 `T` 가 `Base` 타입이나 `Base` 에서 파생된 타입이어야 한다는 뜻
    - 타입 계층 모델링

```
interface Disposable {
  val name: String
  fun action(): String
}

class Compost(override val name: String) :
  Disposable {
  override fun action() = "Add to composter"
}

interface Transport : Disposable

class Donation(override val name: String) :
  Transport {
  override fun action() = "Call for pickup"
}

class Recyclable(override val name: String) :
  Transport {
  override fun action() = "Put in bin"
}

class Landfill(override val name: String) :
  Transport {
  override fun action() = "Put in dumpster"
}

val items = listOf(
  Compost("Orange Peel"),
  Compost("Apple Core"),
  Donation("Couch"),
  Donation("Clothing"),
```

```
  Recyclable("Plastic"),
  Recyclable("Metal"),
  Recyclable("Cardboard"),
  Landfill("Trash"),
)

val recyclables =
  items.filterIsInstance<Recyclable>()
```

- 제약을 사용하면 제네릭 함수 안에서 제약이 이뤄진 타입의 프로퍼티와 함수에 접근 가능

```
fun <T: Disposable> nameOf(disposable: T) =
  disposable.name

// As an extension:
fun <T: Disposable> T.name() = name

fun main() {
  recyclables.map { nameOf(it) } eq
    "[Plastic, Metal, Cardboard]"
  recyclables.map { it.name() } eq
    "[Plastic, Metal, Cardboard]"
}
```

  - 제약을 사용하지 않으면 name 을 사용할 수 없다.

- 제네릭스를 쓰지 않고 같은 결과를 내는 예제

```
fun nameOf2(disposable: Disposable) =
  disposable.name

fun Disposable.name2() = name

fun main() {
  recyclables.map { nameOf2(it) } eq
    "[Plastic, Metal, Cardboard]"
  recyclables.map { it.name2() } eq
    "[Plastic, Metal, Cardboard]"
}
```

- 같은 결과를 내지만 타입 파라미터 제약을 써야 한다.
  - 제네릭스를 쓰지 않고 다형성을 쓰는 경우 반환 타입을 기반 타입으로 업캐스트해 반환하지만, 제네릭을 쓰면 정확한 타입을 지정할 수 있다.

```
private val rnd = Random(47)

fun List<Disposable>.aRandom(): Disposable =
  this[rnd.nextInt(size)]
```

```
fun <T: Disposable> List<T>.bRandom(): T =
  this[rnd.nextInt(size)]

fun <T> List<T>.cRandom(): T =
  this[rnd.nextInt(size)]

fun sameReturnType() {
  val a: Disposable = recyclables.aRandom()
  val b: Recyclable = recyclables.bRandom()
  val c: Recyclable = recyclables.cRandom()
}
```

- 제네릭스를 쓰지 않은 `aRandom()`은 기반 클래스인 `Disposable` 만 만들어 낼 수 있다.
  반면,
  `bRandom`, `cRandom`은 `Recyclable` 을 만든다.
- `bRandom()` 은 `Disposable`의 멤버를 전혀 사용하지 않아 `T`에 걸린 `:Disposable` 제약은 의미 없어서, 제약을 걸지 않은 `cRandom()` 과 같아진다.

- 타입 파라미터 제약이 필요한 경우는 아래 두 가지가 모두 필요할 때입니다.
  - 타입 파라미터 안에 선언된 함수나 프로퍼티에 접근해야 한다.
  - 결과를 반환할 때 타입을 유지해야 한다.

```
private val rnd = Random(47)

// action()에 접근 할 수 있지만 정확한 타입을 반환할 수 없다.
fun List<Disposable>.inexact(): Disposable {
  val d: Disposable = this[rnd.nextInt(size)]
  d.action()
  return d
}

// 제약이 없어서 action()에 접근할 수 없다.
fun <T> List<T>.noAccess(): T {
  val d: T = this[rnd.nextInt(size)]
  // d.action()
  return d
}

// action()에 접근하고 정확한 타입을 반환한다.
fun <T: Disposable> List<T>.both(): T {
  val d: T = this[rnd.nextInt(size)]
  d.action()
  return d
}

fun constraints() {
```

```
  val i: Disposable = recyclables.inexact()
  val n: Recyclable = recyclables.noAccess()
  val b: Recyclable = recyclables.both()
}
```

    - `inexact()` 는 `List<Disposable>`의 확장이므로 함수 내부에서 `action()`에 접근할 수 있다.
      하지만, 제네릭 함수가 아니므로 기반 타입인
      `Disposable`로만 값을 반환할 수 있다.
    - 제네릭 함수인 `noAccess()`는 정확히 `T` 타입을 반환할 수 있지만, 제약이 없음으로 `Disposable`에 정의된 `action()` 에 접근할 수 없다.
    - `T`에 제약을 가한 `both()` 에서만 `action()` 에 접근하면서 정확한 타입을 반환할 수 있다.

- 타입 소거
    - 제네릭 타입은 컴파일 시점에만 사용할 수 있고 런타임 바이트코드에서는 제네릭 타입 정보가 보존되지 않는다 (제네릭 타입의 파라미터 타입이 지워짐)

```
fun main() {
    val strings = listOf("a", "b", "c")
    val all: List<Any> = listOf(1, 2, "x")
    useList(strings)
    useList(all)
}

fun useList(list: List<Any>) {
    // 주석을 해제하면 에러가 발생한다. (타입 소거 때문에 실행 시점에 제네릭 타입의 E
    // if (list is List<String>) {}
}
```

    - 사용 이유
        - 자바 호환성을 유지한다.
        - 타입 정보를 유지하려면 부가 비용이 든다.

- 함수 타입 인자에 대한 실체화
    - 제네릭 함수를 호출할 때도 타입 정보가 소거된다.
    - 함수 인자의 타입 정보를 보전하려면 `reified` 키워드를 추가해야한다.

```
import kotlin.reflect.KClass // 코틀린 클래스를 표현하는 클래스

// a
fun <T: Any> a(kClass: KClass<T>) {
  // KClass<T> 사용
}

// b
// 타입 소거로 인해 컴파일되지 않음
// fun <T: Any> b() = a(T::class)
```

```
// c
// 자바에서는 수작업으로 타입 정보를 전달한다.
// 컴파일러가 이미 T의 타입을 알고 조용히 타입을 넘겨줄 수 있지만,
// 이렇게 명시적으로 타입 정보를 전달하는 것은 불 필요한 중복이다.
fun <T: Any> c(kClass: KClass<T>) = a(kClass)

class K

val kc = c(K::class)

// d
// reified 키워드는 불필요한 중복을 제거해준다.
// reified 를 사용하기 위해서는 제네릭 함수를 inline 으로 선언해야 한다.
inline fun <reified T: Any> d() = a(T::class)

val kd = d<K>()
```

- `d` 와 `c` 는 같은 효과를 내지만, 클래스 참조를 인자로 요구할 필요가 없다.
- `reified` 는 `reified` 가 붙은 타입 인자의 타입 정보를 유지시키라고 컴파일러에 명령한다.
  이제 실행 시점에도 타입 정보를 사용할 수 있기 때문에 함수 본문 안에서 이를 쓸 수 있다.

```
inline fun <reified T> check(t: Any) = t is T
// fun <T> check1(t: Any) = t is T     // [1]

fun main() {
  check<String>("1") eq true
  check<Int>("1") eq false
}
```

- [1] `reified` 가 없으면 타입 정보가 지워지므로 실행 시점에 어떤 객체가 T의 인스턴스인지 검사할 수 없다.
- 실제 사용 예시 (`filterIsInstance`)

```
inline fun <reified T : Disposable> select() =
  items.filterIsInstance<T>().map { it.name }

fun main() {
  select<Transport>() eq
    "[Orange Peel, Apple Core]"
  select<Donation>() eq "[Couch, Clothing]"
  select<Recyclable>() eq
    "[Plastic, Metal, Cardboard]"
  select<Landfill>() eq "[Trash]"
}
```

- 타입 변성

- `in`, `out` 변성 애너테이션을 붙여서 타입 파라미터를 적용한 Container 타입의 상하위 타입 관계를 제한해야 한다.
- 예제로 알아본다.

```
class Box<T>(private var contents: T) {
  fun put(item: T) { contents = item }
  fun get(): T = contents
}

class InBox<in T>(private var contents: T) {
  fun put(item: T) { contents = item }
}

class OutBox<out T>(private var contents: T) {
  fun get(): T = contents
}
```

  - `in T` 는 이 클래스의 멤버 함수가 `T` 타입 값을 인자로만 받고, `T` 타입 값을 반환하지 않는다는 뜻이다.
    즉,
    `T` 객체를 `InBox` 안으로 넣을 수는 있어도, `InBox`에서 `T` 객체가 나올 수는 없다.
  - `out T` 는 이 클래스의 멤버 함수가 `T` 타입 값을 반환하기만 하고 `T` 타입 값을 인자로 받지는 않는다는 뜻이다.
    즉,
    `T` 객체가 `OutBox` 밖으로 나올수는 있어도 `OutBox` 안으로 `T` 객체를 넣을 수 없다.
- 이런 제약이 왜 필요할까? 예제로 알아본다.

```
open class Pet
class Cat : Pet()
class Dog : Pet()
```

  - Cat 과 Dog 모두 Pet 의 하위 타입
  - `Box<Pet>` 타입 변수에 `Box<Cat>` 객체를 대입할 수 있어야 할 것 같다.

    `Cat`의 `Box` 를 `Pet` 의 `Box` 에 대입, `Any` 의 `Box` 에 대입하는 것이 가능해야 한다.

```
val catBox = Box<Cat>(Cat())
// val petBox: Box<Pet> = catBox
// val anyBox: Box<Any> = catBox
```

  - 코틀린이 이를 허용한다면 `Dog` 를 `catBox` 에 넣을 수 있게됨으로, 이는 `catBox` 를 위반하게 된다.
  - `anyBox` 에서도 `put` 이 있을 수 있는데, `catBox` 에 `Any` 타입 객체를 넣을 수 있음으로, `catBox`는 아무런 타입 안전성을 제공하지 못한다.

```
val outCatBox: OutBox<Cat> = OutBox(Cat())
val outPetBox: OutBox<Pet> = outCatBox
```

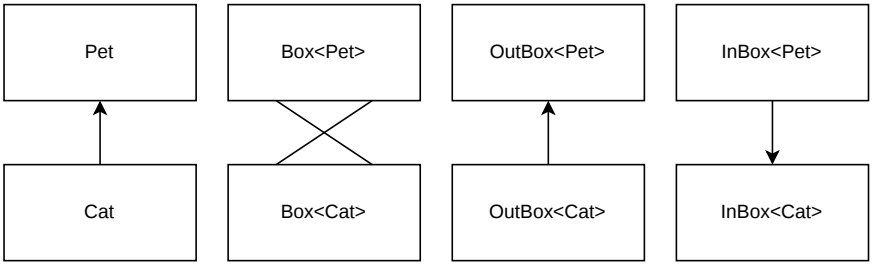
```
val outAnyBox: OutBox<Any> = outCatBox

fun getting() {
  val cat: Cat = outCatBox.get()
  val pet: Pet = outPetBox.get()
  val any: Any = outAnyBox.get()
}
```

- `OutBox`에서 `put()` 사용을 막으면, 아무도 `Dog` 를 `OutBox<Cat>`에 넣을 수 없음으로 `catBox` 를 `petBox`나 `anyBox`에 대입하는 대입문이 안전해진다.

```
val inBoxAny: InBox<Any> = InBox(Any())
val inBoxPet: InBox<Pet> = inBoxAny
val inBoxCat: InBox<Cat> = inBoxAny
val inBoxDog: InBox<Dog> = inBoxAny

fun main() {
  inBoxAny.put(Any())
  inBoxAny.put(Pet())
  inBoxAny.put(Cat())
  inBoxAny.put(Dog())

  inBoxPet.put(Pet())
  inBoxPet.put(Cat())
  inBoxPet.put(Dog())

  inBoxCat.put(Cat())
  inBoxDog.put(Dog())
}
```

- `get()` 이 없기 때문에 `InBox<Any>` 이나 `InBox<Cat>` 혹은 `InBox<Dog>`에 대입할 수 있다.

◦ 변성



- `Box<T>` 는 무공변이다.

  `Box<Cat>` 과 `Box<Pet>` 사이에 아무런 하위 타입 관계가 없다.

- `OutBox<out T>`는 공변이다.

`OutBox<Cat>`을 `OutBox<Pet>`으로 업캐스트하는 방향이 `Cat`을 `Pet` 하는 업캐스트하는 방향과 같은 방향이다.

- `InBox<in T>`는 반공변이다.

  `InBox<Pet>`이 `InBox<Cat>` 의 하위 타입이다. `InBox<Pet>`을 `InBox<Cat>`으로 업캐스트하는 방향이 `Cat` 을 `Pet` 으로 업캐스트하는 방향과 반대 방향으로 변한다.

- **함수는 공변적인 반환 타입을 가진다.**
  오버라이드하는 함수가 오버라이드 대상 함수보다 구체적인 반환 타입을 돌려줘도 된다는 뜻

```
interface Parent
interface Child : Parent

interface  X {
  fun f(): Parent
}

interface Y : X {
  override fun f(): Child
}
```

- `Y`에서 오버라이드하는 `f()`가 `Child` 반환하지만 `X`의 `f()` 는 `Parent` 를 반환한다.

4. 연산자 오버로딩

- 새로 만든 타입에 대해 + 같은 연산자에 의미를 부여하거나 기존 타입에 대해 작용하는 연산자에 추가로 의미를 부여할 수 있다.
  - 연산자를 오버로딩하려면 `fun` 앞에 `operator` 를 붙어야 한다.
  - `Num` 클래스의 + 확장 함수 연산자 오버로딩

```
data class Num(val n: Int)

operator fun Num.plus(rval: Num) = Num (n + rval.n)

fun mian() {
    Num(4) + Num(5) eq Num(9)
    Num(4).plus(Num(5)) eq Num(9)
}
```

  - 어떤 연산자를 확장 함수로 정의하면 클래스의 private 멤버를 접근 불가능, 멤버 함수는 접근 가능

```
data class Num2(private val n: Int) {
    operator fun plus(rval: Num2) = Num2(n + rval.n)
}

// Cannot acess 'n': it is private in 'Num2' 라는 오류 발생
// operator fun Num2.minus(rval: Num2) = Num2(n - rval.n)
```

```
fun main() {
    Num2(4) + Num2(5) eq Num(9)
}
```

- 동등성
  - == 과 != 은 equals() 멤버 함수를 호출한다.
    - `data` 클래스는 모든 필드를 서로 비교하는 `equals()` 를 오버라이드 해줌.
    - 일반 클래스는 참조를 비교하는 디폴트 버전이 실행됨.
  - 동일성 비교 (==, !=)

```
class A(val i: Int)

data class D(val i: Int)

fun main() {
    // 일반 클래스
    val a = A()
    val b = A()
    val c = a
    (a == b) eq false
    (a == c) eq true
    // 데이터 클래스
    val d = D(1)
    val e = D(1)
    (d == e) eq true
}
```

  - `equals()` 는 확장 함수로 정의할 수 없는 유일한 연산자. (반드시 멤버함수 오버라이드)
    클래스 안에서
    `equals()` 를 정의할 때는 디폴트 `equals(other: Any?)` 를 오버라이드 함- `other` 의 타입이 `Any?` 임으로, 타입과 다른 타입을 비교할 수 있다.
    즉, 이 연산자를 오버라이드할 때는 반드시 비교 대상 타입을 지정해야 한다.

```
class E(var v: Int) {
    override fun equals(other: Any?) = when {
        this === other -> true // [1]
        other !is E -> false   // [2]
        else -> v == other.v   // [3]
    }
    override fun hashCode(): Int = v
    override fun toString() = "E($v)"
}

fun main() {
    val a = E(1)
```

```
    val b = E(2)
    (a == b) eq false
    (a != b) eq true
    // 참조 동등성
    (E(1) ===  E(1)) eq false
}
```

- [1] `other` 가 메모리에서 `this`와 같은 객체를 가르키면 결과는 `true` 이다
  === (삼중 등호) 참조 동등성 검사
- [2] `other` 의 타입이 현재 타입(클래스) 과 같은지 결정하는 코드다.

  `E`를 `other` 의 타입과 비교해 타입이 일치한 경우에만 다음에 오는 검사를 수행
- [3] 저장된 데이터를 비교하는 코드. 이 시점에서는 컴파일러는 `other`의 타입이 E라는 사실을 알기 때문에 별도의 타입 변환 없어 `other.v`를 사용할 수 있다.
- **`equals` 를 구현한다면 `hashCode` 도 구현해야한다. (컬렉션 프레임워크 참조)**

- 널이 될 수 있는 객체를 ==로 비교하면 코틀린은 널 검사를 강제한다.
  이 경우 if 나 엘비스 연산자를 통해 널 검사한다.

```
fun equalsWithIf(a: E?, b: E?) =
  if (a === null)
    b === null
  else
    a == b

fun equalsWithElvis(a: E?, b: E?) =
  a?.equals(b) ?: (b === null)

fun main() {
  val x: E? = null
  val y = E(0)
  val z: E? = null
  (x == y) eq false
  (x == z) eq true
  equalsWithIf(x, y) eq false
  equalsWithIf(x, z) eq true
  equalsWithElvis(x, y) eq false
  equalsWithElvis(x, z) eq true
}
```

- `equalsWithIf()`, `equalsWithElvis()`는 먼저 참조가 `a` 가 `null` 인지 검사한다.

  `a` 가 `null` 인 경우 `a` 와 `b` 가 같을 수 있는 유일한 경우는 `b`도 `null` 일 때뿐이다.
- 둘 다 같은 효과지만 안전한 호출과 앨비스 연산자를 사용하는 게 간결하다.