Objects and Classes

Objects are the basis of object-oriented programming. In Kotlin, every piece of data is an object.

Every object has a type, such as `String`, `Int`, `List<Int>`. The type determines what you can do with the object.

A class defines a new type of object. Think about a class as a blueprint for objects. You can create objects from the blueprint.

Objects with attributes

A simple class defines an object with various attributes:

- A point has x- and y-coordinates;
- a date has a year, month, and day;
- a student has a name, student id, and department;
- a playing card has a suit and a face value.

(There are 52 cards. Each card has a face and a suit. The suits are clubs, spades, hearts, and diamonds. The faces are 2, 3, . . . , 10, Jack, Queen, King, Ace.)

Data classes

Define a class:

```
>>> data class Point(val x: Int, val y: Int)
```

Create an object:

```
>>> var p = Point(2, 5)
>>> p
Point(x=2, y=5)
```

Use fields of the object:

```
>>> p.x
2
>>> p.y
5
```

Compare objects with `==` and `!=`.

Print objects with `println`.

More examples

Dates:

```
data class Date(val year: Int, val month: Int,
                val day: Int)
```

Students:

```
data class Student(val name: String, val id: Int,
                   val dept: String)
```

Blackjack cards:

```
data class Card(val face: String, val suit: String)
```

Mutable and immutable objects

If the state of an object cannot change after the object has been constructed, it is immutable. In Kotlin, `String`, `List`, pairs, and triples are immutable.

The data classes defined before are all immutable.

If the state of an object can change, it is mutable. `MutableList` objects are mutable objects.

A mutable case class for two-dimensional points:

```
data class MPoint(var x: Int, var y: Int)
```

```
>>> val p = MPoint(5, 3)
>>> p
MPoint(x=5, y=3)
>>> p.x = 7
>>> p
MPoint(x=7, y=3)
```


The dangers of mutable objects

What is the value of `p` after the following code?

```
>>> val p = MPoint(3, 5)
>>> val q = p
>>> q.x = 7
>>> q
MPoint(x=7, y=5)
```

```
>>> p
MPoint(x=7, y=5)
```


More dangers

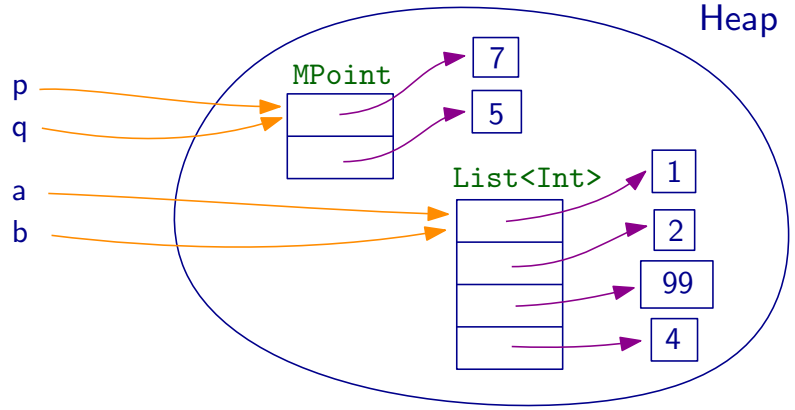
`MutableList` objects are (of course) mutable:

```
>>> val a = mutableListOf(1, 2, 3, 4)
>>> a
[1, 2, 3, 4]
>>> val b = a
>>> b
[1, 2, 3, 4]
>>> a[2] = 99
>>> a
[1, 2, 99, 4]
>>> b
[1, 2, 99, 4]
```


References

All objects live on the heap, a memory area of the JVM.

A variable stores a reference to the object. A reference uniquely identifies one object on the heap.
(Similar to a pointer in C.)



Immutable objects are safer—use them if you can!

Local variables

So where do variables live?

If it is a field of an object, it lives inside the object on the heap. In particular, the elements of a list live inside the `List` object.

The local variables of a method live inside the method's activation record (also called stack frame).

Four local variables:

```
fun test(m: Int) {
  val k = m + 27
  val s = "Hello World"
  val a = listOf( s.length, k, m )
}
```


Garbage collection

Many objects are used only briefly, and not needed afterwards. So after some time, the heap of the JVM will become full.

At that point, the JVM performs garbage collection: It checks all objects on the heap, and determines if there is any reference from a variable on some stack frame leading directly or indirectly to this object. If not, the object is destroyed.

You cannot easily predict when garbage collection happens. Modern systems may perform it incrementally.

Python, Java, and Kotlin programs do not have to worry about memory leaks.