When passing a tuple to a function that takes a single argument.

```
(Int,Int) => String
```

or

```
Function2[Int,Int, String]
```

- They're easier to set up and define.
- They're faster because the compiler can't optimize extractors, since their unapply and unapplySeq can do anything.
- If inheriting a sealed class, the compiler can check for pattern match exhaustiveness.

Patterns to match objects of different types.

If a match of a selector s succeeds against case C(...), you know the selector expression is an instance of C.

Extraction method (unapply)

Injenction method (apply)

An n-element tuple wrapped in a Some.

You can then use the object's name for both a constructor and a pattern, which simulates the convention for pattern matching with case classes.

An element wrapped in a Some, because there is no 1-tuple.

A Boolean indicating match success or failure.

It is named `unapplySeq` and returns `Option[Seq[T]]` for any type T.

- To allow constructor pattern matching without a case class. For example, for pre-existing types like String:

```
s match { case EMail(...
```

- To allow the definition of new patterns that don't follow the internal representation of the scrutinee.

If X is a regex with three capturing groups,

```
valX(a, b, c) = someString
```

a, b, c, will be bound to String matches or null for no match.

Changing case class (e.g. changing its name, moving it to another package) will break client code that matches on it.