```
import scala.collection.mutable
mutable.Set
```

Or rename one of them.

It will use `+=` operator if possible. Otherwise it will try to expand to

```
x = x + y
```

It applies to **any** assignment method to make it easy to switch from mutable to immutable collections and vice versa.

Create an empty one and then add (++) to the existing one.

```
val treeSet = TreeSet[String]() ++ list
val mutaSet = mutable.Set.empty ++ treeSet
val immu = Map.empty ++ muta
```

The tedium of defining simplistic data-heavy classes.

`HashSet` and `HashMap` for all mutable sets, maps, and large immutable ones.

Small (0-4 elements) immutable `Sets` and `Maps` get specialized implementations like `Set4`.

It will multiply assign the `Tuple2` to both first and second, since there are no parens.

- Mix in `SynchronizedSet` or `SynchronizedMap`.
- Use `java.util.concurrent`.
- Use actors and unsychronized collections.
- Or, use immutable ones.

They can hide clashing simple names of the older ones

```
Map()  //immutable
import scala.commection.mutable.Map
Map()  //mutable
import scala.collection.immutable.Map
Map()  //immutable
```

Scala's implicitly converts to `RichString` which is `Seq[Char]`, hence you can iterate over it or use `Seq` methods on it.

```
val stuff = Set[Any] (42)
```

Nicer than

```
val stuff: Set[Any] = Set(42)
```

`TreeSet` and `TreeMap`.

... immutable.

They use red-black trees on items (or keys).

- It defines the type alias of `Set[T]`.
- It makes a val `Set` the singleton `scala.collection.immutable.Set`.

```
typeSet[T] = scala.collection.immutable.Set[T]
val Set = scala.collection.immutable.Set
```