

# GENERIC FUNCTIONS PART 2

# 1. Parametric functions accept ALL types

```
def map[From, To](list: List[From], update: From => To): List[To]
```

```
map[String, Int](List(...), ...)
```

```
map[User, Address](List(...), ...)
```

## 2. All types must be treated EQUALLY

```
def map[From, To](list: List[From], update: From => To): List[To] =  
  list match {  
    case ints    : List[Int]    => ...  
    case strings: List[String] => ...  
    case users   : List[User]   => ...  
    case _       => ...  
  }
```

# All types must be treated EQUALLY

```
def format[A](value: A): String =  
  value match {  
    case x: String => x.toLowerCase  
    case x: Double => truncate(2, x)  
    case _         => "N/A"  
  }
```

```
format("Blue")  
// res0: String = "blue"  
format(123.123456)  
// res1: String = "123.12"  
format(true)  
// res2: String = "N/A"
```

# Why? Type erasure

```
def format[A](value: A): String =  
  value match {  
    case x: String      => x.toLowerCase  
    case x: Double      => truncate(2, x)  
    case x: List[String] => x.map(_._toLowerCase).mkString(",")  
    case x: List[Double] => x.map(truncate(2, _)).mkString(",")  
    case _              => "N/A"  
  }
```

```
format("Blue")  
// res4: String = "blue"  
format(123.123456)  
// res5: String = "123.12"  
format(true)  
// res6: String = "N/A"  
format(List("Hello", "World"))  
// res7: String = "hello,world"
```

# Why? Type erasure

```
def format[A](value: A): String =  
  value match {  
    case x: String      => x.toLowerCase  
    case x: Double      => truncate(2, x)  
    case x: List[String] => x.map(_._toLowerCase).mkString(",")  
    case x: List[Double] => x.map(truncate(2, _)).mkString(",")  
    case _              => "N/A"  
  }
```

```
format("Blue")  
// res4: String = "blue"  
format(123.123456)  
// res5: String = "123.12"  
format(true)  
// res6: String = "N/A"  
format(List("Hello", "World"))  
// res7: String = "hello,world"
```

```
format(List(123.123456, 0.1234))  
// java.lang.ClassCastException: class java.lang.Double cannot be cast to class java.lang.String (java.l  
//   at repl.Session$App3$$anonfun$format$1.apply(generic-functions-part-2.html:58)  
//   at scala.collection.immutable.List.map(List.scala:246)  
//   at repl.Session$App3.format(generic-functions-part-2.html:58)  
//   at repl.Session$App3$$anonfun$8.apply(generic-functions-part-2.html:83)  
//   at repl.Session$App3$$anonfun$8.apply(generic-functions-part-2.html:83)
```

# Why? Type erasure

```
def format[A](value: A): String =  
  value match {  
    case x: String           => ...  
    case x: Double           => ...  
    case x: List[String]     => ...  
    case x: List[Double]    => ...  
    case _                   => ...  
  }
```

```
def format[A](value: A): String =  
    if(value.isInstanceOf[String]) ...  
  else if(value.isInstanceOf[Double]) ...  
  else if(value.isInstanceOf[List[String]]) ...  
  else if(value.isInstanceOf[List[Double]]) ...  
  else ...
```

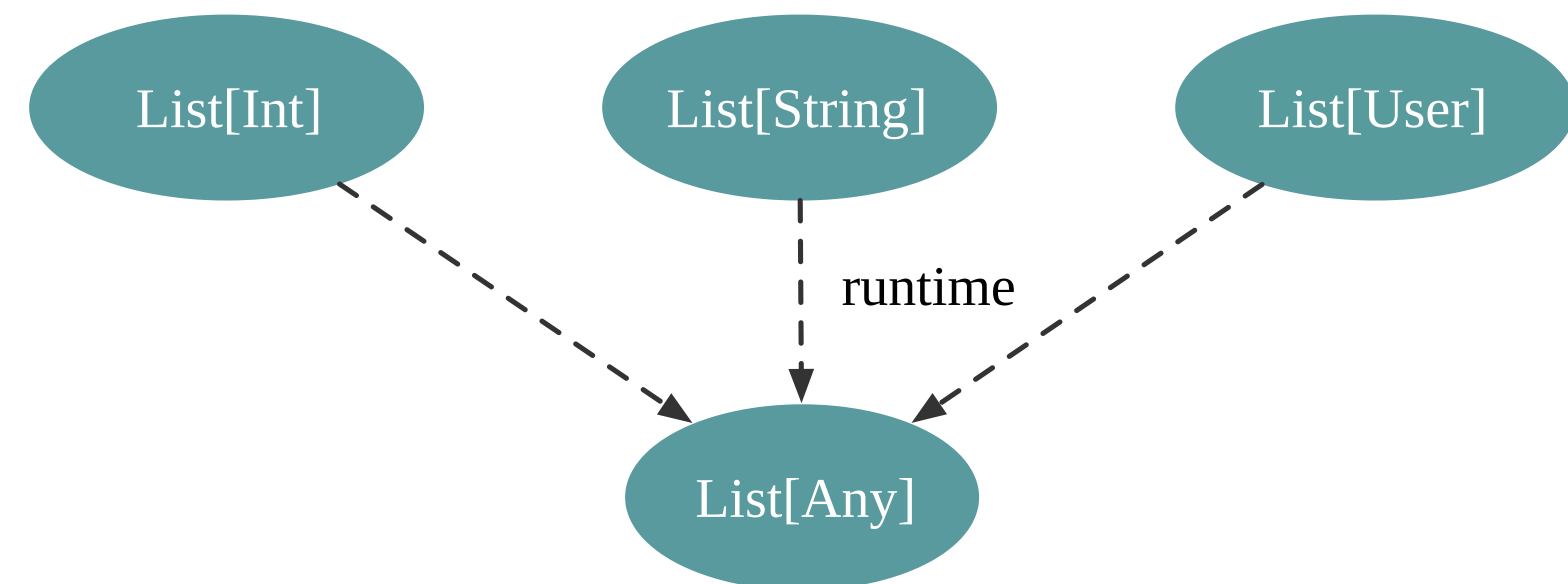
# Why? Type erasure

```
def format[A](value: A): String =  
  value match {  
    case x: String      => ...  
    case x: Double      => ...  
    case x: List[String] => ...  
    case x: List[Double] => ...  
    case _              => ...  
  }
```

```
List(1.5,2.0).asInstanceOf[List[String]]  
// res8: Boolean = true
```

```
// warning: fruitless type test: a value of  
// List[Double] cannot also be a List[String]
```

```
def format[A](value: A): String =  
    if(value.isInstanceOf[String]) ...  
  else if(value.isInstanceOf[Double]) ...  
  else if(value.isInstanceOf[List[String]]) ...  
  else if(value.isInstanceOf[List[Double]]) ...  
  else ...
```





# Why? Poor documentation

```
def format[A](value: A): String
```

Parametric polymorphism (`map`)

`!=`

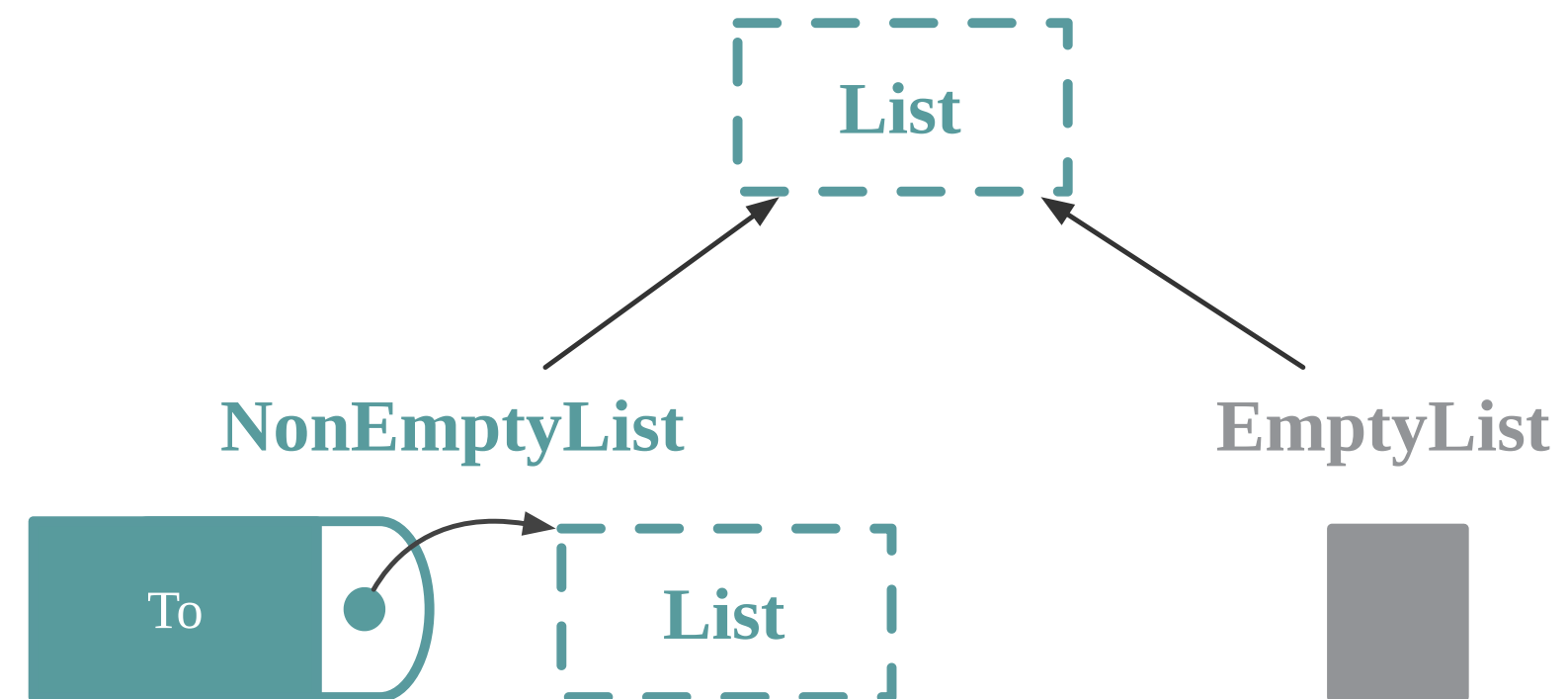
Ad hoc polymorphism (`format`)

# How can we implement map?

```
def map[From, To](list: List[From], update: From => To): List[To]
```

# How can we implement map?

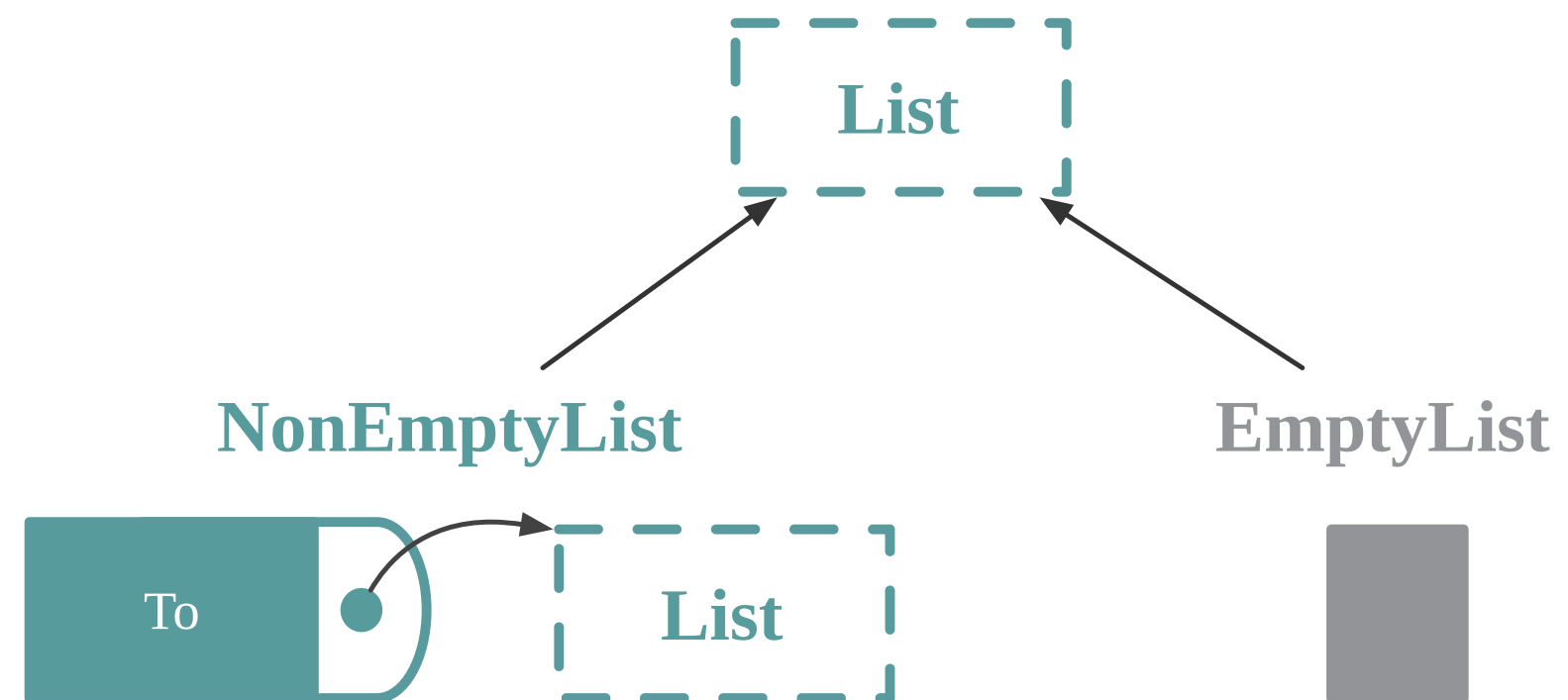
```
def map[From, To](list: List[From], update: From => To): List[To]
```



# How can we implement map?

```
def map[From, To](list: List[From], update: From => To): List[To]
```

1. Return Nil or
2. Must use update on input list



# Does it compile?

```
def map[From, To](list: List[From], update: From => To): List[To] =  
  list
```

# Does it compile?

```
def map[From, To](list: List[From], update: From => To): List[To] =  
  list  
  // error: type mismatch;  
  // found   : List[From]  
  // required: List[To]  
  // list  
  // ^^^^
```

# Does it compile?

```
def map[A](list: List[A], update: A => A): List[A] =  
  list
```



# Does it compile?

```
def map(list: List[Int], update: Int => Int): List[Int] =  
  List(1, 2, 3)
```

# How can we test map?

```
test("map"){  
  map(Nil, _ => ???) shouldEqual Nil  
  
  map(List(1,2,3), (x: Int) => ???) shouldEqual ???  
  map(List(1,2,3,4,5), (x: Int) => ???) shouldEqual ???  
  
  map(List("Hello", "World"), (x: String) => ???) shouldEqual ???  
  map(List("a", "b", "c", "d"), (x: String) => ???) shouldEqual ???  
}
```

# How can we test map?

```
test("map"){  
  map(Nil, _ => ???) shouldEqual Nil  
  
  map(List(1,2,3), (x: Int) => ???) shouldEqual ???  
  map(List(1,2,3,4,5), (x: Int) => ???) shouldEqual ???  
  
  map(List("Hello", "World"), (x: String) => ???) shouldEqual ???  
  map(List("a", "b", "c", "d"), (x: String) => ???) shouldEqual ???  
}
```

All types must be treated in the same way

# How can we test map?

```
test("map"){  
  map(Nil, _ => ???) shouldEqual Nil  
  
  map(List(1,2,3), (x: Int) => x + 1) shouldEqual List(2,3,4)  
  map(List(1,2,3,4,5), (x: Int) => x + 1) shouldEqual List(2,3,4,5,6)  
  
  map(List(1,2,3), (x: Int) => x > 2) shouldEqual List(false,false,true)  
  map(List(1,2,3,4,5), (x: Int) => x > 2) shouldEqual List(false,false,true,true,true)  
}
```

# How can we test map?

```
test("map"){  
  map(Nil, _ => ???) shouldEqual Nil  
  
  map(List(1,2,3), (x: Int) => x + 1) shouldEqual List(2,3,4)  
  map(List(1,2,3,4,5), (x: Int) => x + 1) shouldEqual List(2,3,4,5,6)  
  
  map(List(1,2,3), (x: Int) => x > 2) shouldEqual List(false,false,true)  
  map(List(1,2,3,4,5), (x: Int) => x > 2) shouldEqual List(false,false,true,true,true)  
}
```

All output elements must be produced by update

# How can we test map?

```
test("map"){  
  map(Nil, _ => ???) shouldEqual Nil  
  
  map(List(1,2,3), (x: Int) => x) shouldEqual List(1,2,3)  
  map(List(1,2,3,4,5), (x: Int) => x) shouldEqual List(1,2,3,4,5)  
}
```

# How can we test map?

```
test("map"){  
  map(Nil, _ => ???) shouldEqual Nil  
  
  map(List(1,2,3), identity) shouldEqual List(1,2,3)  
  map(List(1,2,3,4,5), identity) shouldEqual List(1,2,3,4,5)  
}
```

```
object Predef {  
  def identity[A](value: A): A = value  
}
```

# Property Based Testing (PBT)

```
test("map"){  
  forAll((list: List[Int]) =>  
    map(list, identity) shouldEqual list  
  )  
}
```



# Summary

- More reusable
- Caller decides which underlying type to use
- Implementation must be generic
  - better documentation
  - less tests

# Not everything in a language should be used

```
class Spaceship {  
  private def autoDestroy(): Unit =  
    println("Boom!")  
}
```

```
val shuttle = new Spaceship()
```

```
shuttle.autoDestroy()  
// error: method autoDestroy in class Spaceship cannot be accessed as a member of App1.this.Spa  
// shuttle.autoDestroy()  
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

# Not everything in a language should be used

```
class Spaceship {  
  private def autoDestroy(): Unit =  
    println("Boom!")  
}
```

```
val shuttle = new Spaceship()
```

```
val method = classOf[Spaceship].getDeclaredMethod("autoDestroy")  
method.setAccessible(true)
```

```
method.invoke(shuttle)  
// Boom!
```