# Parametric functions - Part 2

# Interpretation

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

# Interpretation

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

1. A parametric function can handle any types
2. All types must be treated **IN THE SAME WAY**

# All types must be treated in the same way

```
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 in the same way

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

```scala
show("Hello")
// res0: String = "Hello"
show(123.123456)
// res1: String = "123.12"
show(true)
// res2: String = "N/A"
```

# Why? Type erasure

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

```
show("Hello")
// res4: String = "Hello"
show(123.123456)
// res5: String = "123.12"
show(true)
// res6: String = "N/A"
show(List("Hello", "World"))
// res7: String = "List(Hello, World)"
```

```
show(List(123.123456, 0.1234))
// res8: String = "List(123.123456, 0.1234)"
```

# Why? Type erasure

```scala
def show[A](value: A): String =
  value match {
    case x: String    => x
    case x: Double    => truncate(2, x)
    case x: List[Any] => x.toString
    case x: List[Any] => x.map(truncate(2, _)).toString
    case _            => "N/A"
  }
```

```scala
show("Hello")
// res9: String = "Hello"
show(123.123456)
// res10: String = "123.12"
show(true)
// res11: String = "N/A"
show(List("Hello", "World"))
// res12: String = "List(Hello, World)
```

```scala
show(List(123.123456, 0.1234))
// res13: String = "List(123.123456, 0.1234)"
```

# Why? Bad documentation

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

# Solution 1: Overloaded functions

```
def show(value: String): String =
  value

def show(value: Double): String =
  truncate(2, value)

def defaultShow[A](value: A): String =
  "N/A"
```

```
show("Hello")
// res15: String = "Hello"
show(123.123456)
// res16: String = "123.12"
defaultShow(true)
// res17: String = "N/A"
```

# Solution 1: Overloaded functions

```
def show(value: String): String =
  value

def show(value: Double): String =
  truncate(2, value)

def defaultShow[A](value: A): String =
  "N/A"
```

```
show("Hello")
// res15: String = "Hello"
show(123.123456)
// res16: String = "123.12"
defaultShow(true)
// res17: String = "N/A"
```

```
def show(value: List[String]): String = ???
def show(value: List[Double]): String = ???
// error: double definition:
// def show(value: List[String]): String at line 64 and
// def show(value: List[Double]): String at line 65
// have same type after erasure: (value: List)String
// def show(value: List[Double]): String = ???
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

# Solution 2: Enumeration

```scala
sealed trait ShowValue
case class ShowString(value: String) extends ShowValue
case class ShowDouble(value: Double) extends ShowValue
case class ShowStrings(value: List[String]) extends ShowValue
case class ShowDoubles(value: List[Double]) extends ShowValue
case class ShowDefault[A](value: A) extends ShowValue

def show(value: ShowValue): String =
  value match {
    case ShowString(x)  => x
    case ShowDouble(x)  => truncate(2, x)
    case ShowStrings(x) => x.toString
    case ShowDoubles(x) => x.map(truncate(2, _)).toString
    case ShowDefault(_) => "N/A"
  }
```

```scala
show(ShowStrings(List("Hello", "World")))
// res19: String = "List(Hello, World)"
show(ShowDoubles(List(123.123456, 0.1234)))
// res20: String = "List(123.12, 0.12)"
```

# Solution 2: Enumeration (Dotty)

```
enum ShowValue {
  case class ShowString(value: String)
  case class ShowDouble(value: Double)
  case class ShowStrings(value: List[String])
  case class ShowDoubles(value: List[Double])
  case class ShowDefault[A](value: A)
}

def show(value: ShowValue): String =
  value match {
    case ShowString(x)  => x
    case ShowDouble(x)  => truncate(2, x)
    case ShowStrings(x) => x.toString
    case ShowDoubles(x) => x.map(truncate(2, _)).toString
    case ShowDefault(_) => "N/A"
  }
```

```
show(ShowStrings(List("Hello", "World")))
// res21: String = "List(Hello, World)"
show(ShowDoubles(List(123.123456, 0.1234)))
// res22: String = "List(123.12, 0.12)"
```

# Solution 3: Interface

```
trait Show[A] {
  def show(value: A): String
}

val showString: Show[String] = new Show[String] {
  def show(value: String): String = value
}

val showDouble: Show[Double] = new Show[Double] {
  def show(value: Double): String = truncate(2, value)
}

def showOption[A](showA: Show[A]): Show[Option[A]] = new Show[Option[A]]{
  def show(value: Option[A]): String =
    value match {
      case Some(x) => showA.show(x)
      case None    => defaultShow.show(value)
    }
}

def defaultShow[A]: Show[A] = new Show[A]{
  def show(value: A): String = "N/A"
}
```

# How can we implement `map`?

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

# How can we implement `map`?

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

- Always return `List.empty` (`Nil`)

# How can we implement `map`?

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

- Always return `List.empty` (`Nil`)
- Somehow call `f` on the elements of `list`

# Does it compile?

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

# Does it compile?

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

On line 3: error: type mismatch;
        found   : Int(1)
        required: To
```

# Does it compile?

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

On line 3: error: type mismatch;
        found   : Int(1)
        required: To
```

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

# Does it compile?

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

On line 3: error: type mismatch;
        found   : Int(1)
        required: To
```

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

#2 Benefit: require less tests and less documentation

# Summary

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

# Exercise 2: Parametric functions

`exercises.function.FunctionExercises.scala`