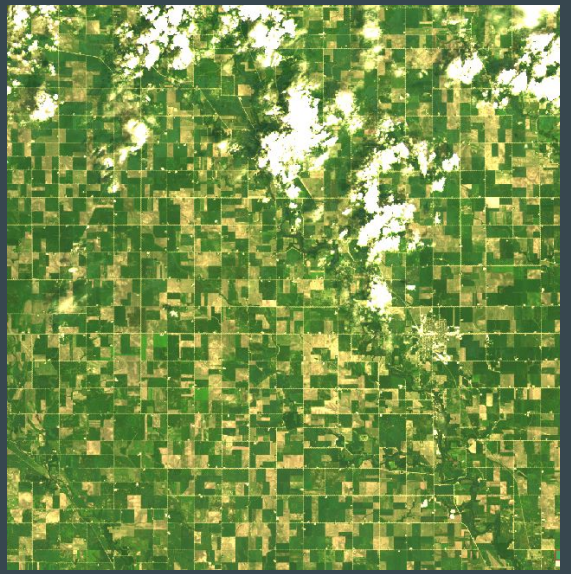
Your Scala Type System

...

Working for You!

The Earth is not Flat

But most of the ways we look at it are:



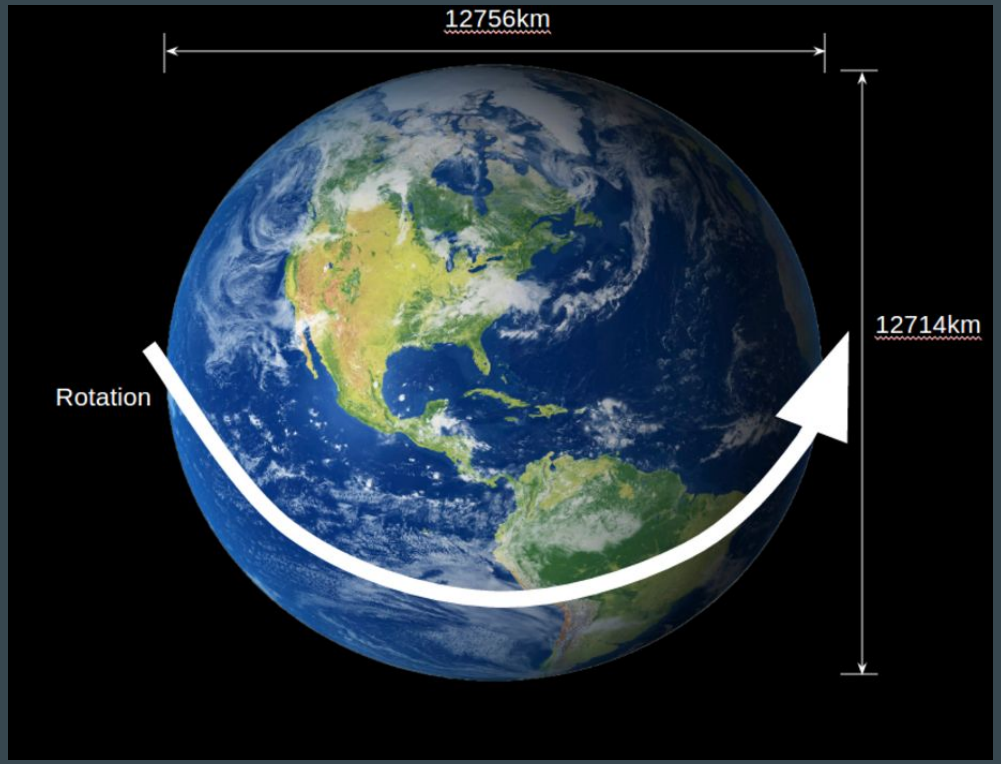
It's not exactly round either...

The Earth has a bit of a "spare-tire" caused by its rotation

This makes the math harder

Some overly simple models ignore this

Including one of the most widely used, Web Mercator (Google, Bing, etc.)



Referencing a Location on Earth

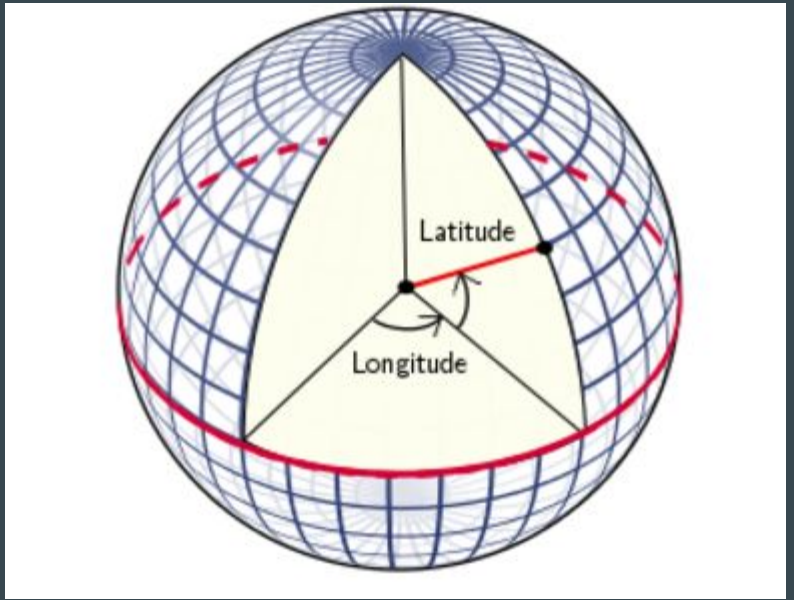
GPS Latitude and Longitude

Defined as 2 angles

One perpendicular to the equator
(with 0 running through London), -180 to 180

One parallel to the poles, -90 to 90

Combined, where they intersect the surface
of the earth defines any point



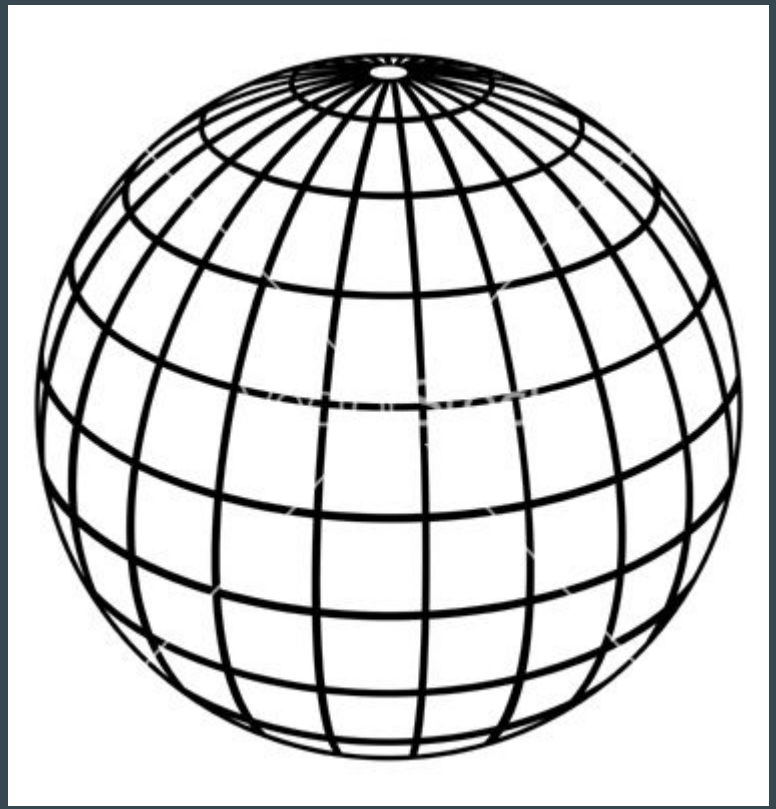
This is great for points, lines and shapes

As long as they are not too big/coarse

Not very suitable for images though

The problem for images is that
as you move further North or South,
the amount of distance covered by
1.0 degrees longitude changes

At the equator, 1 degree longitude
is about 69 miles, in San Jose it's about
55 miles, and in Anchorage, AK, 33 miles

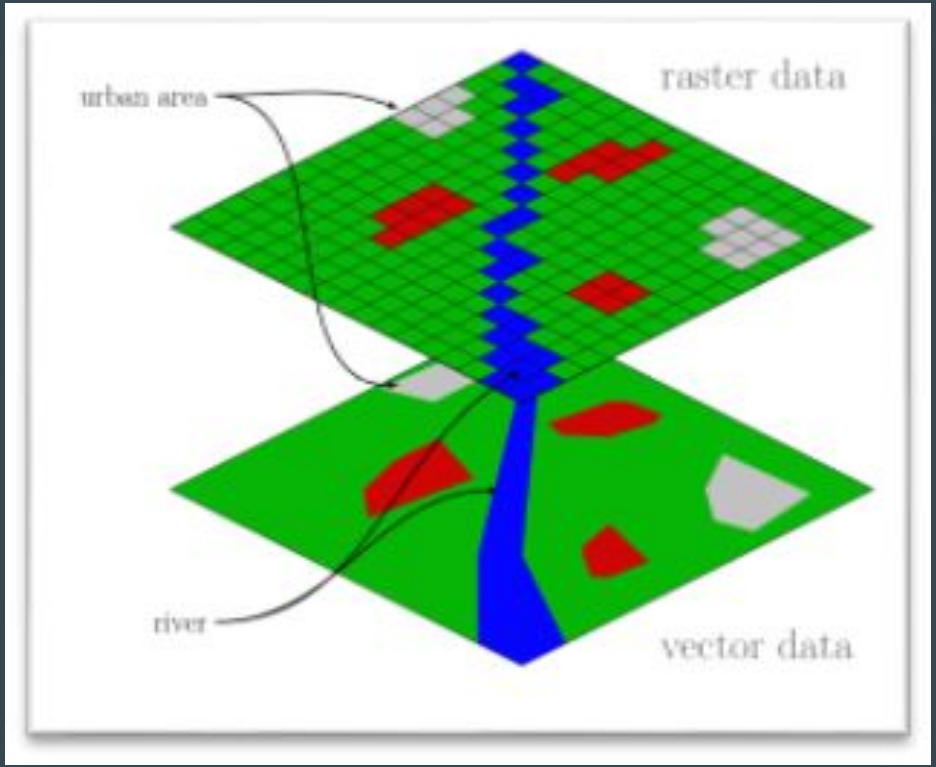


Two Primary Types of Spatial Data

Raster

and

Vector



Raster Images Tend To Assume Constant Measurements

For processing, we usually want constant measurements for our pixels (for areas, distances, etc)

This means we need to mathematically project the 3d Earth into a 2d flat system

Coordinate Reference Systems and Projections

A Coordinate Reference System is anything that assigns coordinates to a location

A projection is anything that mathematically projects between spatial models

The projection (+ datum) defines how one CRS relates to another mathematically

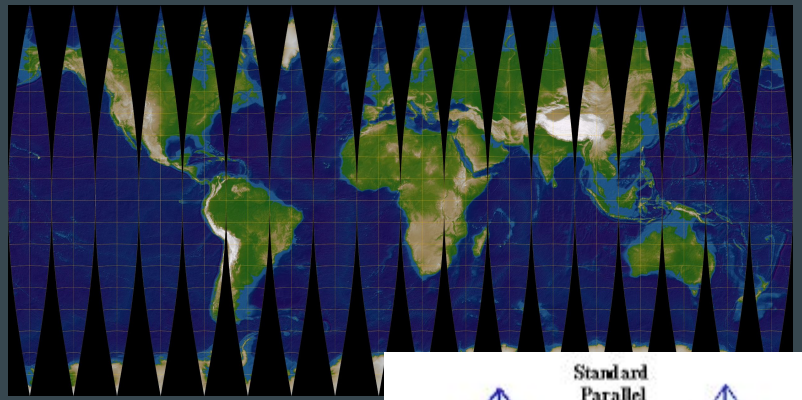
We can convert between the CRS (with varying amounts of error)

Error tends to be smaller for vector data than for raster

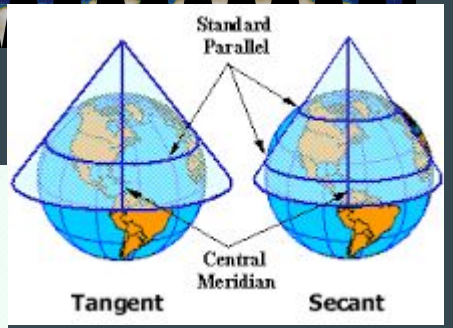
Therefore we resist transforming (or resampling) raster data more

Projection Examples

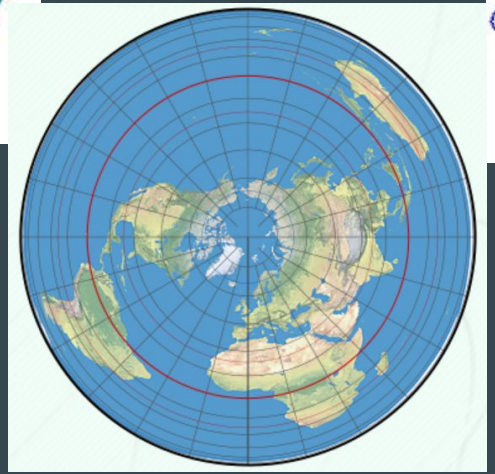
Transverse Mercator



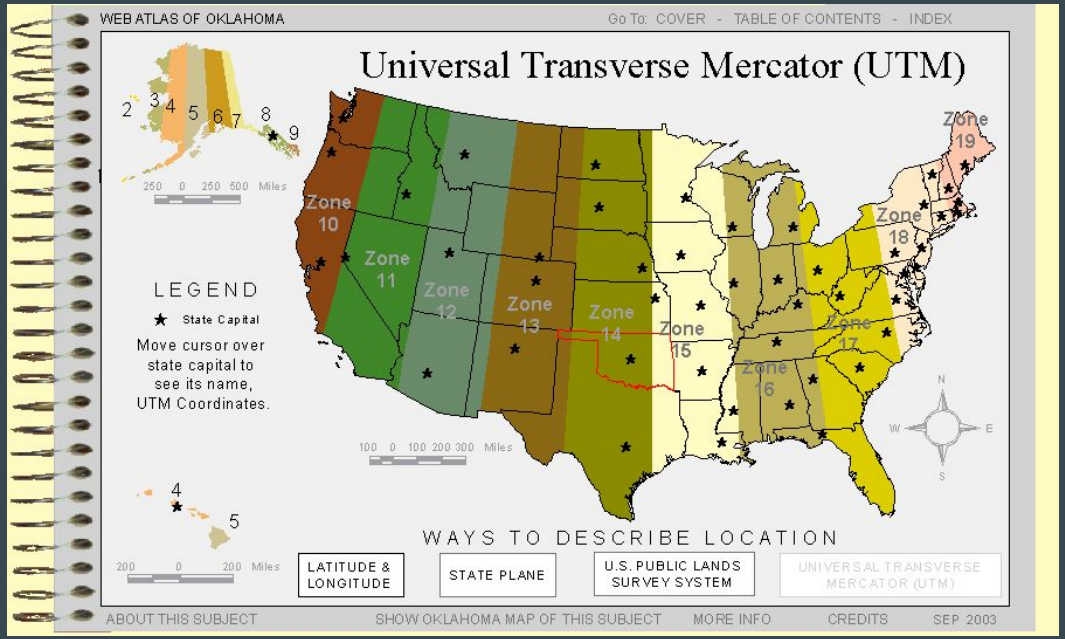
Conical



Polar



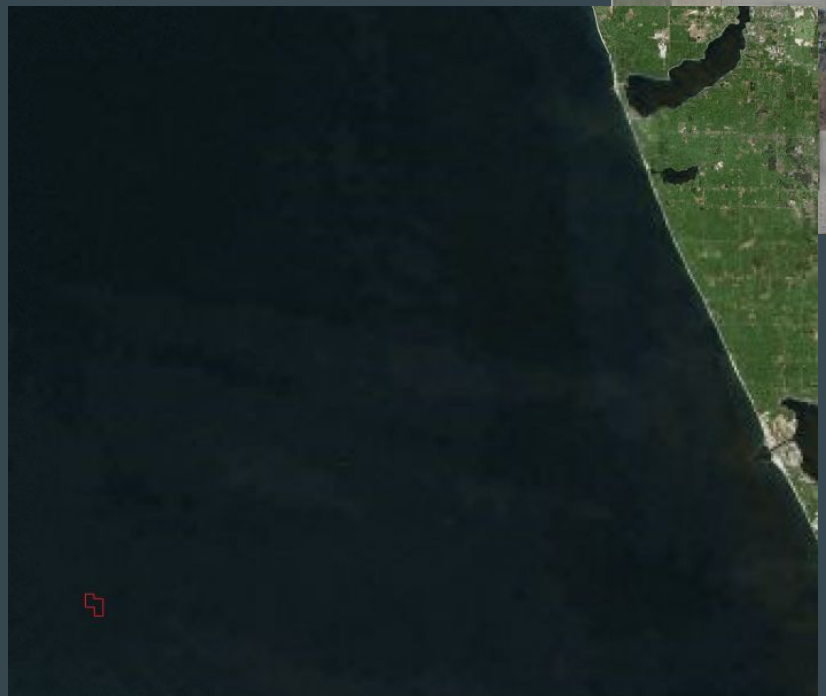
WGS 84 UTM Zones



The Importance of Enforcing CRS

Many CRS space coordinates can overlap

It's critical not to get them mixed up, or you will get the wrong data, sometimes undetectably!



What does most code do?

Lots of redundant CRS checks, or they get the wrong CRS...

```
import com.escalatesoft.crs.CRSDef.CRSDefinitions.longitudeFirst
import org.opengis.referencing.crs.CoordinateReferenceSystem
import org.geotools.referencing.CRS

val epsg4326 = CRS.decode( code = "EPSG:4326", longitudeFirst)
val epsg32615 = CRS.decode( code = "EPSG:32615")
val epsg32616 = CRS.decode( code = "EPSG:32616")


def ensureSameCRS(crs1: CoordinateReferenceSystem,
  crs2: CoordinateReferenceSystem): Unit =
  if (crs1.getCoordinateSystem != crs2.getCoordinateSystem)
    throw new IllegalStateException("Incompatible CRSs")

ensureSameCRS(epsg4326, epsg4326)
ensureSameCRS(epsg4326, epsg32615)
```

```
res0: Unit = ()
java.lang.IllegalStateException: Incompatible CRSs
```

Works, but it's runtime...

We can do better.

Type Parameters are Part of the Type!

E.g. Scala's **Set** (which is invariant)

```
val s1 = Set(1,2,3)
val s2 = Set("Hello", "CRS", "Types")

def stringInSet(s: String, set: Set[String]): Boolean =
  set.contains(s)

stringInSet("CRS", s2) // true

stringInSet("CRS", s1)
```

Type mismatch, expected: Set[String], actual: Set[Int]

This is not a talk on Scala variance

But here's what you need to know for now: covariant vs invariant

```
val xs1 = List(1,2,3)
val xs2 = List("Hello", "CRS", "Types")

def sizeOfList(xs: List[Any]): Int = xs.size

sizeOfList(xs1)                          res0: Int = 3
sizeOfList(xs2)                          res1: Int = 3
```

```
15    def lengthOfSet(set: Set[Any]): Int = set.size
16
17    lengthOfSet(s1)
18  Type mismatch, expected: Set[Any], actual: Set[Int]
19
```

CRS Types! Let's start with a simple definition

```
abstract class CRSDef(val crsCode: String) {
  val crs: CoordinateReferenceSystem = CRS.decode(crsCode, longitudeFirst)
}
```

```
case class EPSG_4326() extends CRSDef( crsCode = "EPSG:4326")
```

You may be wondering why I didn't make this a case object...

We need that case class and its companion

to do this...

```
// general WGS84
case class EPSG_4326() extends CRSDef( crsCode = "EPSG:4326")
object EPSG_4326 extends CRSType[EPSG_4326]
```

by doing this...

```
abstract class CRSType[T] {
  type CRS_TYPE = T
  def apply(): CRSDef
  val crsDef: CRSDef = apply()
  val crs: CoordinateReferenceSystem = crsDef.crs
  val crsId: String = crsDef.crs.getIdentifiers.toArray.head.toString
  implicit val implicitCrsDef: CRSType[CRS_TYPE] = this
}
```

Type and definition in harmony

We can now use a type parameter (compile time verified) and still look up the CRS definition at runtime (so we no longer need to pass it as a regular parameter):

```
abstract class GeometryType[CRS: CRSType] {
  lazy val crs: CoordinateReferenceSystem = implicitly[CRSType[CRS]].crs

  protected def findTransformTo[NEW_CRS: CRSType]: MathTransform = {
    val newCRS = implicitly[CRSType[NEW_CRS]]
    val transform = GCRS.findMathTransform(crs, newCRS.crs, lenient = true)
    transform
  }
}
```

Geometries have the CRS as part of the type

```scala
case class Coord2D[CRS: CRSType](x: Double, y: Double) extends GeometryType[CRS] {

  private lazy val jtsCoordinate: Coordinate =
    new Coordinate(x, y)

  def transformCRS[NEW_CRS: CRSType]: Coord2D[NEW_CRS] = {
    val transform: MathTransform = findTransformTo[NEW_CRS]
    transformWith[NEW_CRS](transform)
  }

  private[geometry] lazy val jtsPoint: JTSPoint = {
    val gf = new GeometryFactory()
    gf.createPoint(jtsCoordinate)
  }

  private[geometry] def transformWith[NEW_CRS: CRSType](transform: MathTransform): Coord2D[NEW_CRS] = {
    val newCoord = if (transform.isIdentity) jtsCoordinate else
      JTS.transform(new Coordinate(x, y), new Coordinate(), transform)

    Coord2D[NEW_CRS](newCoord.x, newCoord.y)
  }

  def within(poly: Polygon[CRS]): Boolean = poly.contains(this)
}
```

Now, the compiler has your back

```
1  import com.escalatesoft.crs.CRSType.CRSDefinitions._
2  import com.escalatesoft.geometry.Coord2D
3
4  val coordsWGS84: Coord2D[EPSG_4326] = Coord2D[EPSG_4326](-121.6544, 37.1305)
5
6  def showMap(center: Coord2D[EPSG_4326]): Unit = {
7    println(s"Showing map centered on $center")
8  }
9
10 showMap(coordsWGS84)          Showing map centered on Coord2D(-121.6544,37.1305)
11
12
13 val coordsUTM10: Coord2D[EPSG_32610] = coordsWGS84.transformCRS[EPSG_32610]
14                                                    Coord2D(619524.2832872181,4110196.620520887)
15 showMap(coordsUTM10)
   Type mismatch, expected: Coord2D[CRSType.CRSDefinitions.EPSG_4326], actual: Coord2D[CRSType.CRSDefinitions.EPSG_32610]
17
18 showMap(coordsUTM10.transformCRS[EPSG_4326])   Showing map centered on Coord2D(-121.65439999999933,37.13049999999965)
```

I See a Flaw! You Don't Always Know the CRS at Compile Time

```scala
trait SomeCRS {
  type SOME_CRS
  implicit val SOME_CRS: CRSType[SOME_CRS]

  override def toString: String = s"SomeCRS($SOME_CRS)"
  override def equals(obj: Any): Boolean = {...}
  override def hashCode(): Int = {...}
}
```

```scala
def optCrsFromId(id: String): Option[SomeCRS] = {
  for (ccrs <- CRSDefinitions._registered.get(id)) yield {
    new SomeCRS {
      type SOME_CRS = ccrs.CRS_TYPE
      val SOME_CRS: CRSType[ccrs.CRS_TYPE] =
        ccrs.asInstanceOf[CRSType[SOME_CRS]]
    }
  }
}
```

Reifying the CRS type from the existential

```
describe ( description = "Existential CRS lookup") {
  val anyOldCRSID = "EPSG:32615"

  val someCRS = CRSType.optCrsFromId(anyOldCRSID).get
  import someCRS.SOME_CRS

  val someCRSCoord = Coord2D[SOME_CRS](10.0, 10.0)
  val utmZone15NCoord = someCRSCoord.transformCRS[EPSG_32615]

  utmZone15NCoord.x should be (10.0 +- 1e-9)
  utmZone15NCoord.y should be (10.0 +- 1e-9)

}
```

Other geometries

```scala
case class Polygon[CRS: CRSType](coords: Vector[Coord2D[CRS]]) extends GeometryType[CRS] {

  private val closed: Seq[Coord2D[CRS]] =
    if (coords.last == coords.head) coords
    else coords :+ coords.head

  private[geometry] lazy val jtsPolygon: JTSPolygon = {
    val gf = new GeometryFactory()
    val points = closed.map(c => new geom.Coordinate(c.x, c.y)).asJava
    gf.createPolygon(new LinearRing(new CoordinateArraySequence(points.toArray(new Array[Coordinate](points.size))), gf), holes = null)
  }

  def transformCRS[NEW_CRS: CRSType]: Polygon[NEW_CRS] = {
    val transform = findTransformTo[NEW_CRS]
    val convertedCoords = coords.map(c => c.transformWith[NEW_CRS](transform))
    Polygon[NEW_CRS](convertedCoords)
  }

  def contains(coord: Coord2D[CRS]): Boolean = {
    coord.jtsPoint.within(jtsPolygon)
  }

  private def area: Double =
    jtsPolygon.getArea
}

object Polygon {
  def apply[CRS: CRSType](coords: (Double, Double)*): Polygon[CRS] = {
    val gCoords = coords.toVector.map { case (x, y) => Coord2D[CRS](x, y) }
    new Polygon[CRS](gCoords)
  }
```

The API is simple, and hard to get wrong

```
describe ( description = "Point in Polygon with CRSs") {
  it ("should be true only when a point is in the polygon for the same CRS") {
    val poly = Polygon[EPSG_32615](
      (0.0, 0.0), (0.0, 10.0), (10.0, 10.0), (10.0, 0.0), (0.0, 0.0)
    )

    poly.contains(Coord2D[EPSG_32615](5.0, 5.0)) should be (true)
    poly.contains(Coord2D[EPSG_32615](0.001, 0.001)) should be (true)
    poly.contains(Coord2D[EPSG_32615](9.999, 9.999)) should be (true)
```

```
it ("should not even compile if the wrong CRSs are used") {
  val poly = Polygon[EPSG_32615](
    (0.0, 0.0), (0.0, 10.0), (10.0, 10.0), (10.0, 0.0), (0.0, 0.0)
  )

  "poly.contains(Coord2D[EPSG_32616](5.0, 5.0))" shouldNot typeCheck
```

```
poly.contains(Coord2D[EPSG_32616](5.0, 5.0))
```

Type mismatch, expected: Coord2D[CRSType.CRSDefinitions.EPSG_32615], actual: Coord2D[CRSType.CRSDefinitions.EPSG_32616]

```
}
```

Adding some more nuance

```scala
abstract class AngleCRSType[T] extends CRSType[T] {
  override implicit val implicitCrsDef: AngleCRSType[CRS_TYPE] = this
}

abstract class MeterCRSType[T] extends CRSType[T] {
  override implicit val implicitCrsDef: MeterCRSType[CRS_TYPE] = this
}
```

```scala
// general WGS84
case class EPSG_4326() extends CRSDef( crsCode = "EPSG:4326")
object EPSG_4326 extends AngleCRSType[EPSG_4326]

// WGS84 UTM zone 10N
case class EPSG_32610() extends CRSDef( crsCode = "EPSG:32610")
object EPSG_32610 extends MeterCRSType[EPSG_32610]
```

Now we can limit certain methods

```
 def areaSquareMeters[CRS: MeterCRSType](poly: Polygon[CRS]): Double = {
   poly.area
 }
it ("should work only for meter based CRSs with Polygon.areaSquareMeters") {
  val poly1 = Polygon[EPSG_32615](
    (0.0, 0.0), (0.0, 10.0), (10.0, 10.0), (10.0, 0.0), (0.0, 0.0)
  )

  val poly2 = Polygon[EPSG_4326](
    (0.0, 0.0), (0.0, 10.0), (10.0, 10.0), (10.0, 0.0), (0.0, 0.0)
  )

  Polygon.areaSquareMeters(poly1) should be (100.0 +- 1e-6)

  "Polygon.areaSquareMeters(poly2)" shouldNot typeCheck

  Polygon.areaSquareMeters(poly2) should be (100.0 +- 1e-6)
```

Error:(77, 31) could not find implicit value for evidence parameter of type com.escalatesoft.crs.MeterCRSType[com.escalatesoft.crs.CRSType.CRSDefinitions.EPSG_4326]
Polygon.areaSquareMeters(poly2) should be (100.0 +- 1e-6)

Introducing Features

- Features are Geometries with Attributes
- Attributes are fairly loose, and in geotools are untyped (`Map[String, Any]`)
- This leads to all sorts of abuses
- We can do better
- But it would be good to keep a simple "Attach something" capability

Feature Definition

```scala
case class Feature[CRS: CRSType, GEOM <: GeometryType[CRS], +T](geometry: GEOM, attributes: T) {
  def mapAttributes[U](transform: T => U): Feature[CRS, GEOM, U] = {
    Feature[CRS, GEOM, U](geometry, transform(attributes))
  }
```

- Geometry is a type parameter, as is CRS
- Attributes is a covariant type parameter
- We can map over the type parameter to obtain a new feature with different attributes and attribute type, but the same CRS and Geometry

Simple Covariant Behavior

```
def countAttributes(feature: Feature[_, _, List[Any]]): Int = feature.attributes.size
```

```
it ("should allow covariant behavior in the attributes") {
  val feature: Feature[EPSG_32615, Coord2D[EPSG_32615], List[Int]] =
    Feature(Coord2D[EPSG_32615](5.0, 5.0), List(1, 2, 3))
  countAttributes(feature) should be (3)

  val feature2: Feature[EPSG_32615, Coord2D[EPSG_32615], List[String]] =
    Feature(Coord2D[EPSG_32615](5.0, 5.0), List("hello", "world"))
  countAttributes(feature2) should be (2)

  val feature3: Feature[EPSG_32615, Coord2D[EPSG_32615], Int] =
    Feature(Coord2D[EPSG_32615](5.0, 5.0), 20)

  "countAttributes(feature3)" shouldNot typeCheck
}
```

Type-Indexed Maps (TMaps)

```scala
class TMap[+T] private (private val values: List[(Type, Any)]) {
  def apply[E >: T](implicit tt: TypeTag[E]): E = {
    values.find(_._1 <:< tt.tpe).get._2.asInstanceOf[E]
  }

  def get[E](implicit tt: TypeTag[E]): Option[E] = {
    values.find(_._1 <:< tt.tpe).map(_._2.asInstanceOf[E])
  }

  def ++[S](other: TMap[S]): TMap[T with S] =
    new TMap[T with S](other.values ++ values)

  def +[S: TypeTag](other: S): TMap[T with S] =
    this.++[S](TMap[S](other))

  override def toString: String =
    "TMap(" + values.map { case (k,v) => s"$k -> $v" }.mkString(",") + ")"
}

object TMap {
  def apply[A](value: A)(implicit tt: TypeTag[A]): TMap[A] =
    new TMap[A](List(tt.tpe -> value))
```

Using TMaps

```
describe ( description = "A type indexed map") {
  it ("should be easily creatable with inferred type") {
    val tmap: TMap[Int] = TMap(10)
    tmap[Int] should be (10)

    // tmap[String] should be (empty)
    "tmap[String]" shouldNot typeCheck
  }

  it ("should accumulate items and types as they are added") {
    val tmap1: TMap[Int] = TMap(10)
    val tmap2: TMap[Int with String] = tmap1 + "hello"
    val tmap3: TMap[Int with String with List[Int]] = tmap2 + List(1,2,3)

    tmap3[Int] should be (10)
    tmap3[String] should be ("hello")
    tmap3[List[Int]] should be (List(1,2,3))
    "tmap3[List[String]]" shouldNot typeCheck
  }
```

TMap Covariance

```
  def calc(attrs: TMap[List[Int] with Int]): List[Int] = {
    val list = attrs[List[Int]]
    val mult = attrs[Int]

    list.map(_ * mult)
  }

  it ("should allow a function to be called that only needs some of the types") {
    val tmap1: TMap[Int] = TMap(10)
    val tmap2: TMap[Int with String] = tmap1 + "hello"
    val tmap3: TMap[Int with String with List[Int]] = tmap2 + List(1,2,3)

    calc(tmap3) should be (List(10, 20, 30))
    "calc(tmap2)" shouldNot typeCheck
  }
}
```

TMaps as Feature Attributes

```scala
case class Feature[CRS: CRSType, GEOM <: GeometryType[CRS], +T](geometry: GEOM, attributes: T) {
  def mapAttributes[U](transform: T => U): Feature[CRS, GEOM, U] = {
    Feature[CRS, GEOM, U](geometry, transform(attributes))
  }

  private def addAttrInner[TMT, TM <: TMap[TMT], ATTR: TypeTag](attr: ATTR)(
    implicit ev: T <:< TM): Feature[CRS, GEOM, TMap[TMT with ATTR]] = {

    val asTmap: TMap[TMT] = attributes

    val newTMap = asTmap + attr
    Feature[CRS, GEOM, TMap[TMT with ATTR]](geometry, newTMap)
  }
}

object Feature {
  implicit class RichFeature[CRS: CRSType, GEOM <: GeometryType[CRS], T](
      feature: Feature[CRS, GEOM, TMap[T]]) {

    def addAttr[ATTR: TypeTag](attr: ATTR): Feature[CRS, GEOM, TMap[T with ATTR]] =
      feature.addAttrInner[T, TMap[T], ATTR](attr)
  }
}
```

In Use

```
def calcYield(attributes: TMap[Flow with Duration]): Yield = {
  Yield(attributes[Flow].amnt * attributes[Duration].amnt)
}
```

```
it ("should allow a type-map attribute to accumulate types") {
  val feature: Feature[EPSG_32615, Coord2D[EPSG_32615], TMap[FieldName]] =
    Feature(Coord2D(5.0, 5.0), TMap(FieldName("Yarick 80")))

  feature.attributes[FieldName].name should be ("Yarick 80")
  "feature.attributes[Flow].amnt" shouldNot typeCheck
  "feature.attributes[Duration].amnt" shouldNot typeCheck

  val featureWithFlowAndDuration: Feature[EPSG_32615, Coord2D[EPSG_32615], TMap[FieldName with Flow with Duration]] =
    feature.mapAttributes(_ + Flow(10.6)).mapAttributes(_ + Duration(2.5))

  featureWithFlowAndDuration.attributes[FieldName].name should be ("Yarick 80")
  featureWithFlowAndDuration.attributes[Flow].amnt should be (10.6 +- 1e-9)
  featureWithFlowAndDuration.attributes[Duration].amnt should be (2.5 +- 1e-9)

  val featureWithYield: Feature[EPSG_32615, Coord2D[EPSG_32615], TMap[FieldName with Flow with Duration with Yield]] =
    featureWithFlowAndDuration.addAttr(calcYield(featureWithFlowAndDuration.attributes))

  featureWithYield.attributes[Yield].amnt should be (26.5 +- 1e-9)


  val featureWithYield2: Feature[EPSG_32615, Coord2D[EPSG_32615], TMap[FieldName with Yield]] =
    feature.mapAttributes(_ + Yield(26.5))

  "calcYield(featureWithYield2)" shouldNot typeCheck
}
```

So why the implicit extension class?

```
case class Feature[CRS: CRSType, GEOM <: GeometryType[CRS], +T](geometry: GEOM, attributes: T) {
  def mapAttributes[U](transform: T => U): Feature[CRS, GEOM, U] = {
    Feature[CRS, GEOM, U](geometry, transform(attributes))
  }

  def addAttr[TMT, TM <: TMap[TMT], ATTR: TypeTag](attr: ATTR)(
    implicit ev: T <:< TM): Feature[CRS, GEOM, TMap[TMT with ATTR]] = {

    val asTmap: TMap[TMT] = attributes

    val newTMap = asTmap + attr
    Feature[CRS, GEOM, TMap[TMT with ATTR]](geometry, newTMap)
  }
}
```

```
    val featureWithFlowAndDuration: Feature[EPSG_32615, Coord2D[EPSG_32615], TMap[FieldName with Flow with Duration]] =
      feature.
        addAttr(Flow(10.6)).
        addAttr(Duration(2.5))
```

Error:(89, 18) Cannot prove that com.escalatesoft.tmap.TMap[com.escalatesoft.geometry.FieldName] <:< com.escalatesoft.tmap.TMap[Nothing].
addAttr(Flow(10.6)).

Use =:= instead of <:< ?

```
11
12      def addAttr[TMT, TM <: TMap[TMT], ATTR: TypeTag](attr: ATTR)(
13        implicit ev: T =:= TM): Feature[CRS, GEOM, TMap[TMT with ATTR]] = {
```

Covariant type T occurs in invariant position in type T =:= TM of value ev

```
16
```

```
sealed abstract class =:=[From, To] extends (From => To)
```

```
sealed abstract class <:<[-From, +To] extends (From => To)
```

- T is Covariant, so <:< must be used, =:= cannot be
- <:< is too loose, so inference fails us
- implicit class RichFeature "fixes" type T first, then addAttr works with inference

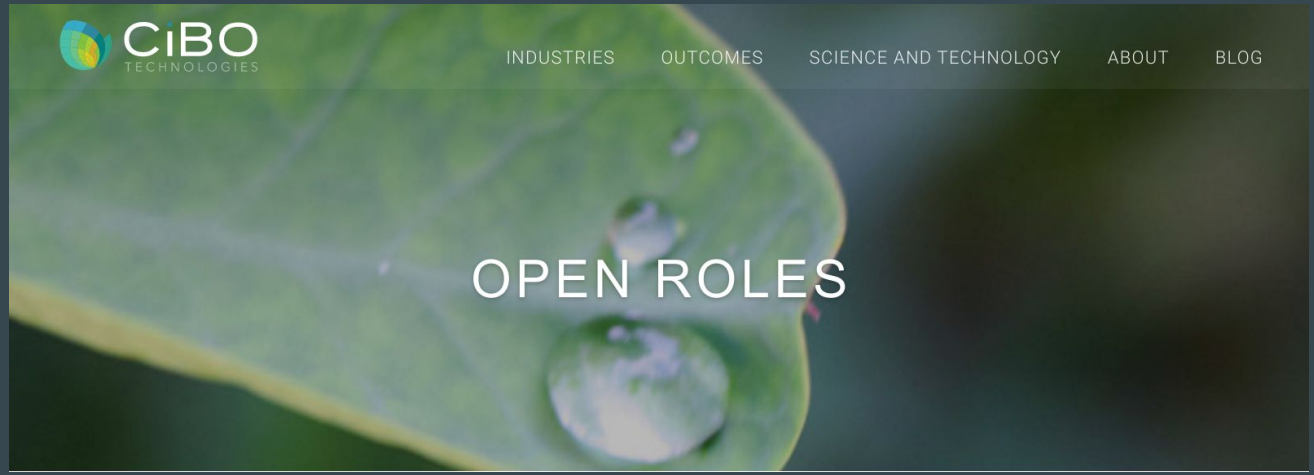
Safety Latches

```scala
object ResampleCRS {
  implicit class ResampleCRS[CRS: CRSType](gc: GridCoverage[CRS]) {

    def resampleCRS[NEW_CRS]: GridCoverage[NEW_CRS] = ???

    def resampleCRS[NEW_CRS](
      resolution: Double, interpolationType: InterpolationType
    ): GridCoverage[NEW_CRS] = ???
  }
}
```

```scala
object DisplayOnly {
  case class EPSG_3857() extends CRSDef(crsCode = "EPSG:3857")
  object EPSG_3857 extends CRSType[EPSG_3857]
}
```

Like This Idea?

We have more,
and we're hiring...



careers@cibotechnologies.com

http://www.cibotechnologies.com/careers/

And if you would like to brush up on your types (and more)

Escalate Software does training, online and in-person:

http://www.escalatesoft.com/training

https://www.udemy.com/user/richard-wall/