Day 4: Generics vs Abstract Types

Generics vs. Abstract types

Abstract types are preceded by the `type` keyword, whereas generics are on classes (which includes methods).

- Abstract types allow for easier path-dependent type usage: `class T { type A }` can be accessed as `T#A`.
- Abstract types have some trickier variance relationships - once you inject a type in a child class, you can't further specify that type in a subsequent subclass. This is the same as generics, but simple override mechanisms for normal objects lead to coder confusion. Code example for this:

```
// with generics
class A[X]
class B[Y <: AnyVal] extends A[Y]

// this works fine - notice how the scope of the generic extends to the constructor
class C[Z <: Int](z: Z) extends B[Z]

// with type members
class X { type A }
class Y extends X { override type A = AnyVal }

// this is an error! AND -- you cannot refer to the abstract type in the case class constructor / apply method!
case class Z() extends Y { override type A = Int; val a: A = 3 }
```

- Generics are good for type instantiation - so standard collections, objects of generic classes. Abstract types require a little more verbosity.
- Generics are way more verbose in general inheritance-based polymorphism (must specify types all the time), see two examples:

```
// basic trait with abstract types
abstract class PrintData
abstract class PrintMaterial
abstract class PrintMedia
trait Printer {
  type Data <: PrintData
  type Material <: PrintMaterial
  type Media <: PrintMedia

  def print(data: Data, material: Material, media: Media) = s"Printing $data with $material material on $media media."
```

```
}

// create related subclasses of each type
case class Paper() extends PrintMedia
case class Air() extends PrintMedia
case class Text() extends PrintData
case class Model() extends PrintData
case class Toner() extends PrintMaterial
case class Plastic() extends PrintMaterial

// create specific Printer instance subtyped per abstract type
class LaserPrinter extends Printer {
  type Media = Paper
  type Data = Text
  type Material = Toner
}
class ThreeDPrinter extends Printer {
  type Media = Air
  type Data = Model
  type Material = Plastic
}

// these same classes look like this with generics
trait GenericPrinter[Data <: PrintData, Material <: PrintMaterial, Media <: PrintMedia] {
  def print(data: Data, material: Material, media: Media) = s"Printing $data with $material material on $media media."
}

class GenericLaserPrinter[Data <: Text, Material <: Toner, Media <: Paper] extends GenericPrinter[Data, Material, Media]
class GenericThreeDPrinter[Data <: Model, Material <: Plastic, Media <: Air] extends GenericPrinter[Data, Material, Media]

// much more verbose and error-prone! must specify types for each new instance!
val genericLaser = new GenericLaserPrinter[Text, Toner, Paper]
val genericThreeD = new GenericThreeDPrinter[Model, Plastic, Air]
```