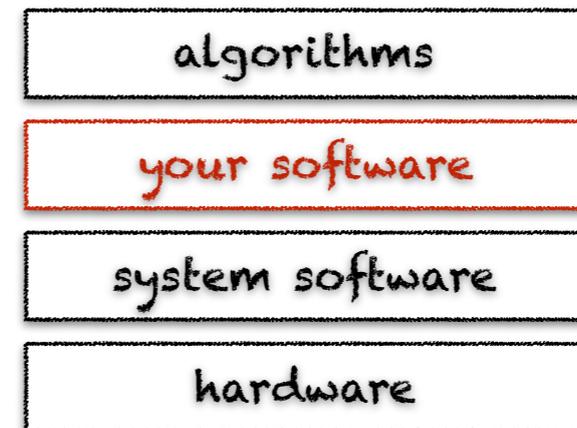




abstract
classes &
types

learning objectives



- ◆ learn how to define and use abstract classes
- ◆ learn how to define types without implementation
- ◆ learn about multiple inheritance of types

a mathematical example

representing matrices

a rectangular array of elements arranged in rows and columns

examples

$$\begin{bmatrix} 4 \\ 1 \\ 8 \end{bmatrix}$$

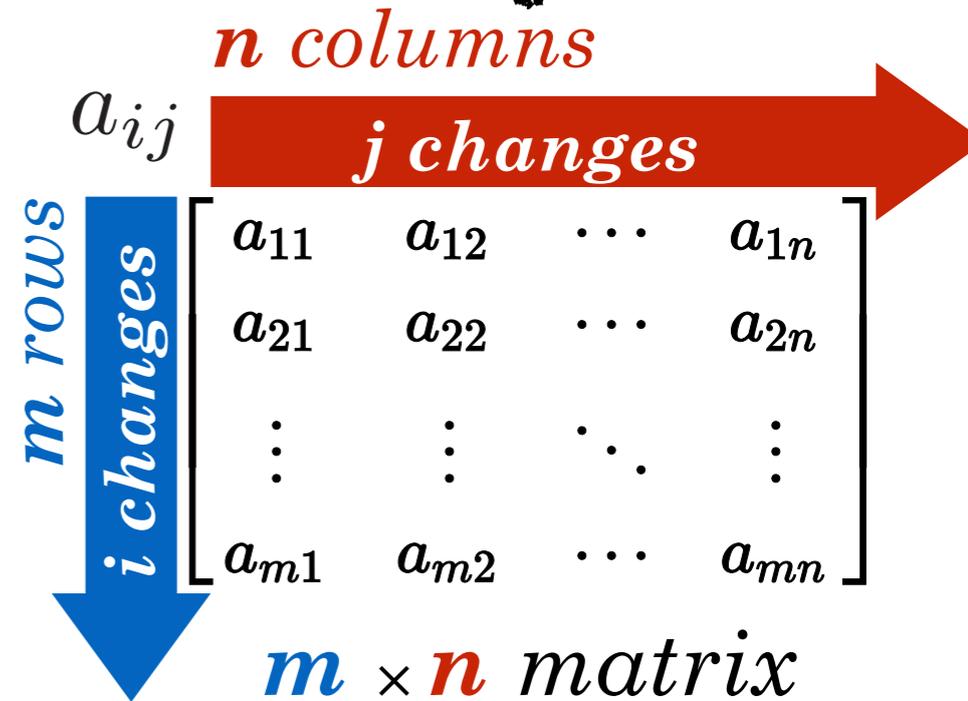
3×1 matrix

$$[3 \quad 7 \quad 2]$$

1×3 matrix

$$\begin{bmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 2 & 6 & 3 \end{bmatrix}$$

3×3 matrix



i designates the row
 j designates the column

matrices are used in many branches of physics, math, computer graphics, etc.

linear equation

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

$$A\vec{x} = \vec{b} \text{ where } A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

representing matrices

sparse matrix

one with most of its elements equal to zero

dense matrix

one with most of its elements **not** equal to zero

square matrix

one with equal number of rows and columns

diagonal matrix

one with all off-diagonal elements equal to zero

$$d_{i,j} = 0 \text{ if } i \neq j \forall i, j \in \{1, 2, \dots, n\}$$

addition

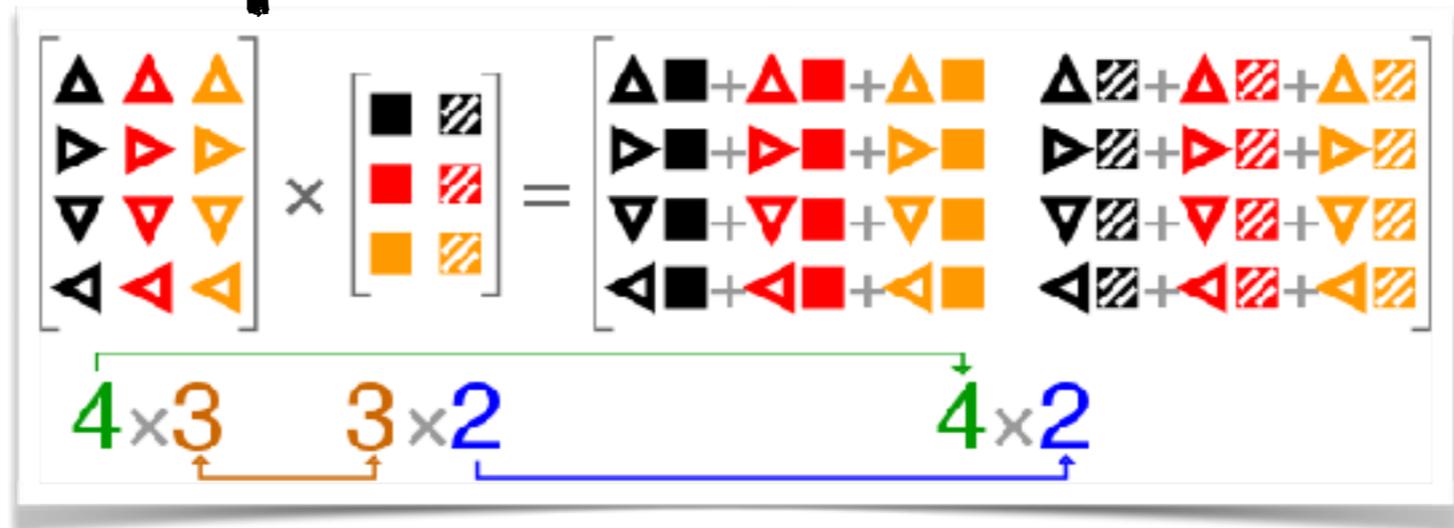
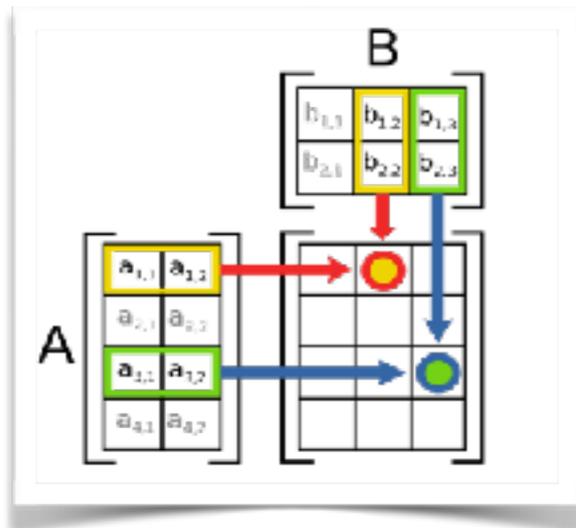
$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix}$$

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}$$

A and B must have the same number of rows and columns

representing matrices

multiplication



$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix} \quad \mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix} \quad (\mathbf{AB})_{ij} = \sum_{k=1}^m A_{ik} B_{kj}$$

$$\mathbf{A} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\lambda + c\rho & a\beta + b\mu + c\sigma & a\gamma + b\nu + c\tau \\ p\alpha + q\lambda + r\rho & p\beta + q\mu + r\sigma & p\gamma + q\nu + r\tau \\ u\alpha + v\lambda + w\rho & u\beta + v\mu + w\sigma & u\gamma + v\nu + w\tau \end{pmatrix}$$

$$\mathbf{BA} = \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} = \begin{pmatrix} \alpha a + \beta p + \gamma u & \alpha b + \beta q + \gamma v & \alpha c + \beta r + \gamma w \\ \lambda a + \mu p + \nu u & \lambda b + \mu q + \nu v & \lambda c + \mu r + \nu w \\ \rho a + \sigma p + \tau u & \rho b + \sigma q + \tau v & \rho c + \sigma r + \tau w \end{pmatrix}$$

$$\mathbf{AB} \neq \mathbf{BA}$$

representing matrices

```
class Matrix(private val rows: Int, private val cols: Int) {  
  if (rows <= 0)  
    throw new IllegalArgumentException("A matrix must have a positive number of rows")  
  if (cols <= 0)  
    throw new IllegalArgumentException("A matrix must have a positive number of columns")  
  
  private var matrix = Array.ofDim[Int](rows, cols)
```

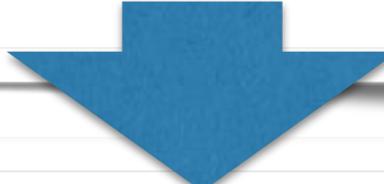
```
  def this(rows: Int, cols: Int, mtx: Array[Array[Int]]) {  
    this(rows, cols)  
    for (i <- 0 to rows - 1)  
      mtx(i).copyToArray(matrix(i))  
  }
```

```
  def numRows: Int = this.rows  
  def numCols: Int = this.cols
```

```
  def apply(i: Int, j: Int): Int = matrix(i)(j)  
  def update(i: Int, j: Int, v: Int) = matrix(i)(j) = v
```

```
  def print: Unit = {  
    println(this.getClass.getSimpleName)  
    for (i <- 0 to this.numRows - 1) {  
      Console.print("| ")  
      for (j <- 0 to this.numCols - 1) {  
        Console.print(s"${this(i,j)} ")  
      }  
      println("|")  
    }  
  }
```

```
var m = new Matrix(3,3)  
m.print()  
  
val mtx : Array[Array[Int]] = Array( Array(2, 4, 6) , Array(3, 6, 9))  
  
m = new Matrix(2,3,mtx)  
m.print()
```



```
Matrix  
| 0 0 0 |  
| 0 0 0 |  
| 0 0 0 |  
  
Matrix  
| 2 4 6 |  
| 3 6 9 |
```

in the following, we assume that indices go from 0 to $n-1$ rather than 1 to n

representing matrices

```
class Matrix(private val rows: Int, private val cols: Int) {  
  if (rows <= 0)  
    throw new IllegalArgumentException("A matrix must have a positive number of rows")  
  if (cols <= 0)  
    throw new IllegalArgumentException("A matrix must have a positive number of columns")  
  
  private var matrix = Array.ofDim[Int](rows, cols)
```

```
  def this(rows: Int, cols: Int, mtx: Array[Array[Int]]) {  
    this(rows, cols)  
    for (i <- 0 to rows - 1)  
      mtx(i).copyToArray(matrix(i))  
  }
```

```
  def numOfRows: Int = this.rows  
  def numOfCols: Int = this.cols
```

```
  def apply(i: Int, j: Int): Int = matrix(i)(j)  
  def update(i: Int, j: Int, v: Int) = matrix(i)(j) = v
```

```
  def print: Unit = {  
    println(this.getClass.getSimpleName)  
    for (i <- 0 to this.numOfRows - 1) {  
      Console.print("| ")  
      for (j <- 0 to this.numOfCols - 1) {  
        Console.print(s"${this(i,j)} ")  
      }  
      println("|")  
    }  
  }
```

```
val mtx : Array[Array[Int]] = Array( Array(2, 4, 6) , Array(3, 6, 9))  
m = new Matrix(2,3,mtx)  
m.print()  
  
println(s"m(0,0) = ${m(0,0)}"); m(0,0) = 7; println(s"m(0,0) = ${m(0,0)}")  
m.print()
```



```
Matrix  
| 2 4 6 |  
| 3 6 9 |  
m(0,0) = 2  
m(0,0) = 7  
Matrix  
| 7 4 6 |  
| 3 6 9 |
```

representing matrices

```
class Matrix(private val rows: Int, private val cols: Int) {
```

```
    ...
    def +(other: Matrix): Matrix = {
        if (this.numOfRows != other.numOfRows || this.numOfCols != other.numOfCols)
            throw new IllegalArgumentException("Matrices must have the same number of rows" +
                " and columns when added")

        val result = new Matrix(this.numOfRows, this.numOfCols)

        for (i <- 0 to this.numOfRows - 1; j <- 0 to this.numOfCols - 1)
            result(i,j) = this (i, j) + other(i, j)
        return result
    }

    def *(other: Matrix): Matrix = {
        if (this.numOfCols != other.numOfRows)
            throw new IllegalArgumentException("The number of columns in the first matrix must be" +
                "equal to the number of rows of the second matrix when multiplied")

        val result = new Matrix(this.numOfRows, other.numOfCols)

        for (i <- 0 to result.numOfRows - 1; j <- 0 to result.numOfCols - 1) {
            var entry: Int = 0
            for (k <- 0 to this.numOfCols - 1)
                entry = entry + this (i, k) * other(k, j)
            result(i,j) = entry
        }
        return result
    }
}
```

$$(\mathbf{AB})_{ij} = \sum_{k=1}^m A_{ik} B_{kj}$$

problem



```
class Matrix(private val rows: Int, private val cols: Int) {
  if (rows <= 0)
    throw new IllegalArgumentException("A matrix must have a positive number of rows")
  if (cols <= 0)
    throw new IllegalArgumentException("A matrix must have a positive number of columns")

  private var matrix = Array.ofDim[Int](rows, cols)

  def this(rows: Int, cols: Int, mtx: Array[Array[Int]]) {
    this(rows, cols)
    for (i <- 0 to rows - 1)
      mtx(i).copyToArray(matrix(i))
  }

  def numOfRows: Int = this.rows
  def numOfCols: Int = this.cols

  def apply(i: Int, j: Int): Int = matrix(i)(j)
  def update(i: Int, j: Int, v: Int) = matrix(i)(j) = v

  def print: Unit = {
    println(this.getClass.getSimpleName)
    for (i <- 0 to this.numOfRows - 1) {
      Console.print("| ")
      for (j <- 0 to this.numOfCols - 1) {
        Console.print(s"${this(i,j)} ")
      }
      println("|")
    }
  }
}
```

```
class SparseMatrix(private val rows: Int, private val cols: Int) {
  if (rows <= 0)
    throw new IllegalArgumentException("A matrix must have a positive number of rows")
  if (cols <= 0)
    throw new IllegalArgumentException("A matrix must have a positive number of columns")

  private var map = scala.collection.mutable.Map[(Int,Int),Int]()

  def this(rows: Int, cols: Int, mtx: Map[(Int,Int),Int]) {
    this(rows, cols)
    map = collection.mutable.Map(mtx.toSeq: _*)
  }

  def numOfRows: Int = this.rows
  def numOfCols: Int = this.cols

  def apply(i: Int, j: Int): Int = map.getOrElse((i,j),0)
  def update(i: Int, j: Int, v: Int): Unit = this.map((i,j)) = v

  def print: Unit = {
    println(this.getClass.getSimpleName)
    for (i <- 0 to this.numOfRows - 1) {
      Console.print("| ")
      for (j <- 0 to this.numOfCols - 1) {
        Console.print(s"${this(i,j)} ")
      }
      println("|")
    }
  }
}
```

code duplication ... again!

problem



can we blend dense and sparse matrices?

```
def *(other: Matrix): Matrix = {
  if (this.numOfCols != other.numOfRows)
    throw new IllegalArgumentException("The number of columns in the first matrix must be" +
      "equal to the number of rows of the second matrix when multiplied")

  val result = new Matrix(this.numOfRows, other.numOfCols)

  for (i <- 0 to result.numOfRows - 1; j <- 0 to result.numOfCols - 1) {
    var entry: Int = 0
    for (k <- 0 to this.numOfCols - 1)
      entry = entry + this(i, k) * other(k, j)
    result(i, j) = entry
  }
  return result
}
```

```
def *(other: SparseMatrix): SparseMatrix = {
  if (this.numOfCols != other.numOfRows)
    throw new IllegalArgumentException("The number of columns in the first matrix must be" +
      "equal to the number of rows of the second matrix when multiplied")

  val result = new SparseMatrix(this.numOfRows, other.numOfCols)

  for (i <- 0 to result.numOfRows - 1; j <- 0 to result.numOfCols - 1) {
    var entry: Int = 0
    for (k <- 0 to this.numOfCols - 1)
      entry = entry + this(i, k) * other(k, j)
    result(i, j) = entry
  }
  return result
}
```



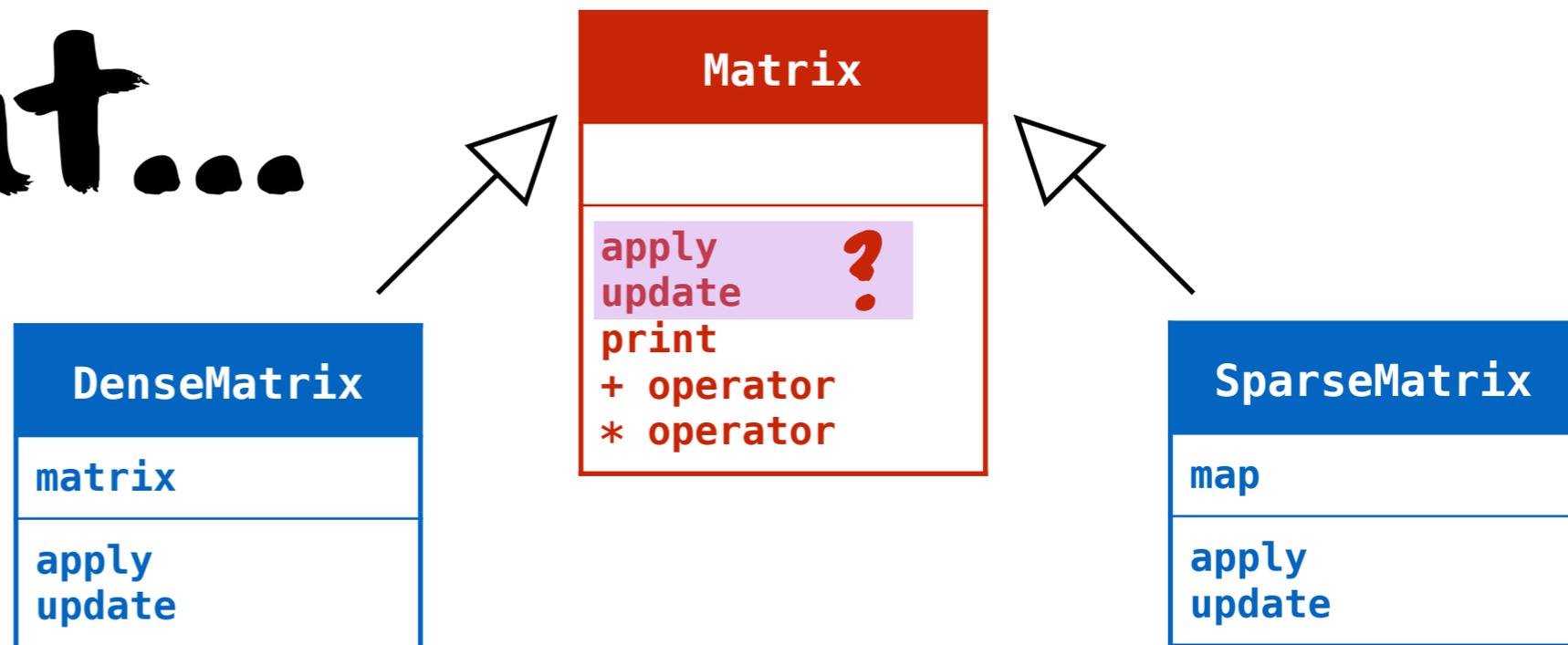
incompatible types

solution

inheritance



yes, but...



the superclass delegates the **internal representation** to its subclasses

so methods **apply** and **update** have no default or shared implementation

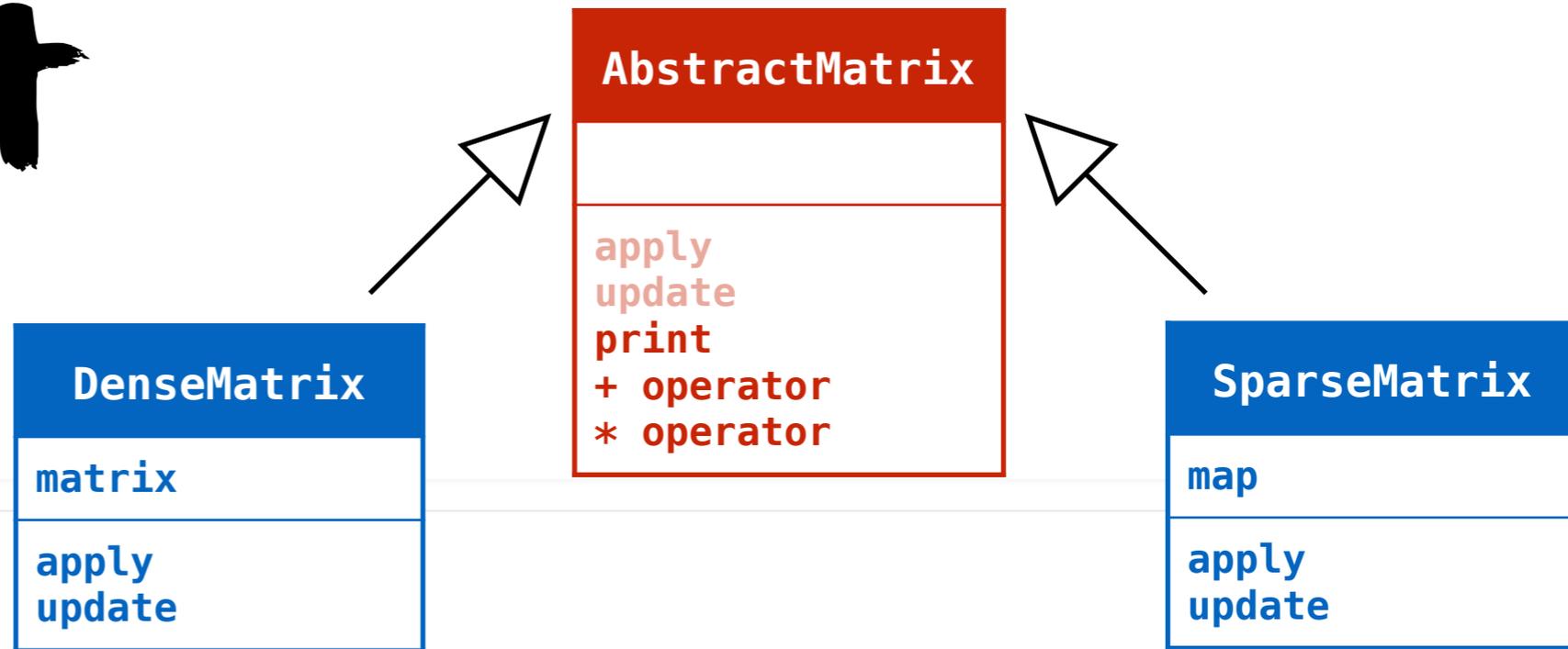
these methods are **abstract** in the superclass

solution



abstract class

abstract class



```
abstract class AbstractMatrix {
  def apply(i: Int, j: Int): Int
  def update(i: Int, j: Int, v: Int)

  def +(other: AbstractMatrix): AbstractMatrix = {
    if (this.numOfRows != other.numOfRows || this.numOfCols != other.numOfCols)
      throw new IllegalArgumentException("Matrices must have the same number of rows and columns when added")

    val result = new DenseMatrix(this.numOfRows, this.numOfCols)
    for (i <- 0 to this.numOfRows - 1; j <- 0 to this.numOfCols - 1)
      result(i,j) = this (i, j) + other(i, j)
    return result
  }

  def *(other: AbstractMatrix): AbstractMatrix = {
    if (this.numOfCols != other.numOfRows)
      throw new IllegalArgumentException("The number of columns in the first matrix must be equal" +
        " to the number of rows of the second matrix when multiplied")

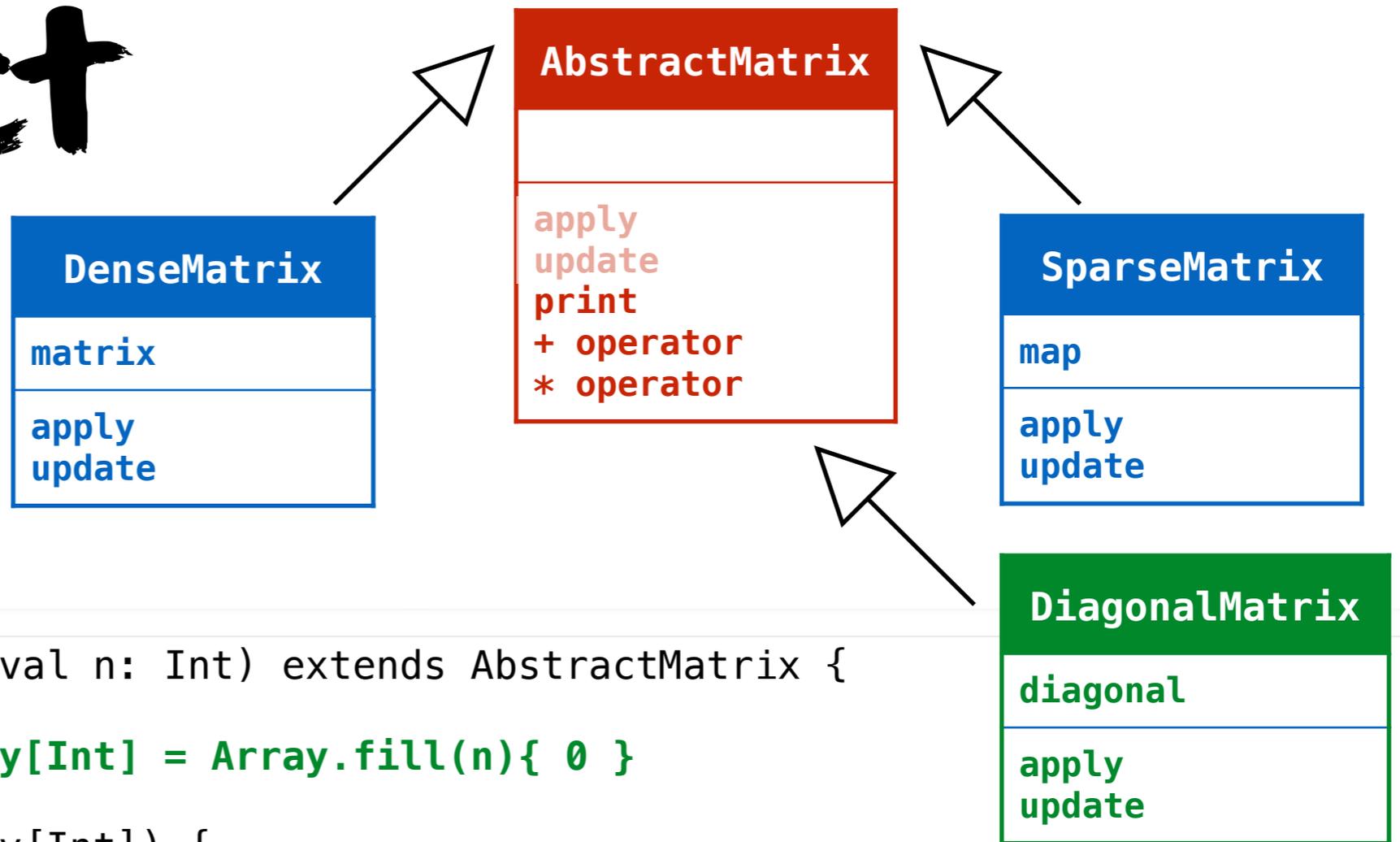
    val result = new DenseMatrix(this.numOfRows, other.numOfCols)
    for (i <- 0 to result.numOfRows - 1; j <- 0 to result.numOfCols - 1) {
      var entry: Int = 0
      for (k <- 0 to this.numOfCols - 1)
        entry = entry + this (i, k) * other(k, j)
      result(i,j) = entry
    }
    return result
  }
}
```

why use **DenseMatrix**, not **SparseMatrix**?

could we come with a better scheme?



abstract class



```
class DiagonalMatrix(private val n: Int) extends AbstractMatrix {  
  
    private var diagonal : Array[Int] = Array.fill(n){ 0 }  
  
    def this(n: Int, diag: Array[Int]) {  
        this(n)  
        this.diagonal = diag  
    }  
  
    def apply(i: Int, j: Int): Int = if (i != j) 0 else diagonal(i)  
  
    def update(i: Int, j: Int, v: Int): Unit = {  
        if (i == j)  
            this.diagonal(i) = v  
        else (i != j && v != 0)  
            throw new IllegalArgumentException("A diagonal matrix should only" +  
                " have zeros outside its diagonal")  
    }  
}
```

what if all methods could be abstract?

identity matrix

$$A \times I_n = I_n \times A = A$$

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} = I_n$$

```
class IdentityMatrix(private val n: Int) extends AbstractMatrix {  
  def apply(i: Int, j: Int): Int = if (i == j) 1 else 0  
  def update(i: Int, j: Int, v: Int): Unit = {  
    if (i != j && v != 1)  
      throw new IllegalArgumentException("A unity matrix can only have ones on its diagonal")  
  }  
  
  override def *(other: Matrix): Matrix = other.duplicate  
  
  override def +(other: Matrix): Matrix = {  
    val result = other.duplicate  
    for (i <- 0 to n - 1)  
      result(i,i) = result(i,i) + 1  
    return result  
  }  
}
```



solution

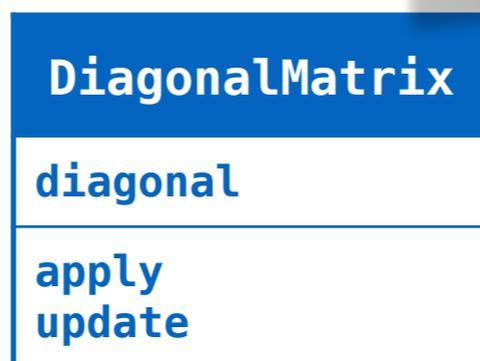
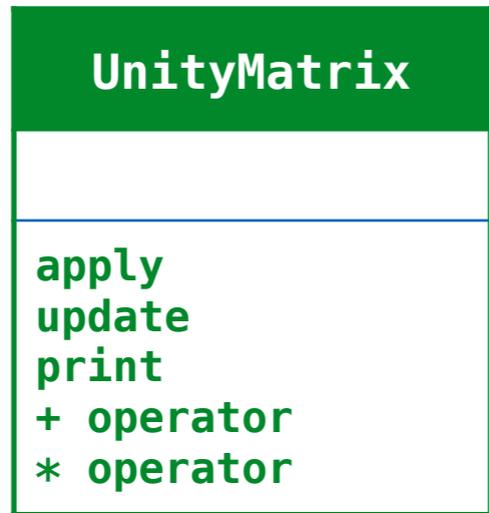


traits

trait



```
trait Matrix {  
  def numRows(): Int  
  def numCols(): Int  
  
  def apply(i: Int, j: Int): Int  
  def update(i: Int, j: Int, v: Int)  
  
  def +(other: Matrix): Matrix  
  def *(other: Matrix): Matrix  
  
  def print  
}
```



```
abstract class AbstractMatrix extends Matrix { ... }
```

```
class DenseMatrix(...) extends AbstractMatrix { ... }
```

```
class SparseMatrix(...) extends AbstractMatrix { ... }
```

```
class DiagonalMatrix(...) extends AbstractMatrix { ... }
```

```
class UnityMatrix(...) extends Matrix { ... }
```

it's time to...

RECAP

a **class** defines a type & provides an implementation for it

a **subclass** defines a subtype & provides an implementation for it

a **type** is just a specification

an **abstract class** defines a type & provides a **partial implementation** for it

a **trait** allows you to define types **without an implementation**

a **class** can inherit from **only one class** but from **multiple traits**

in **swift**, the notion of **protocols** is similar to traits

in **python**, this notion has **no equivalent** due to the absence of static typing