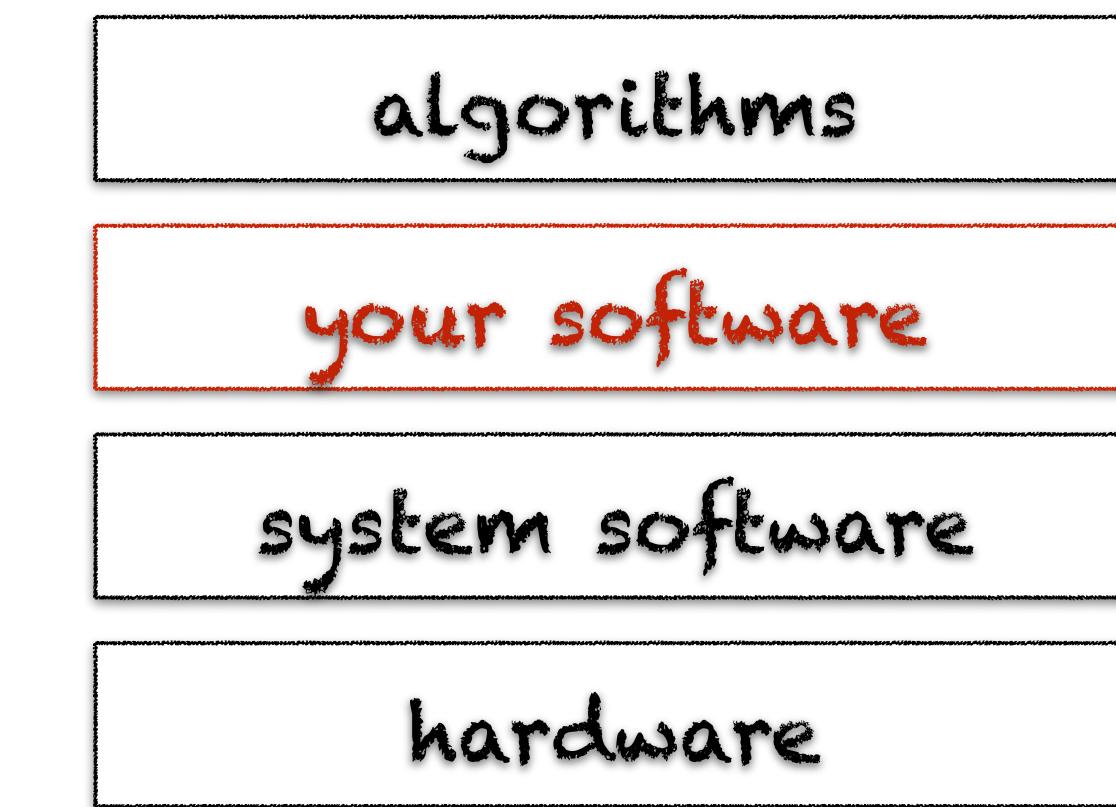




abstract  
classes &  
types

# 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

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

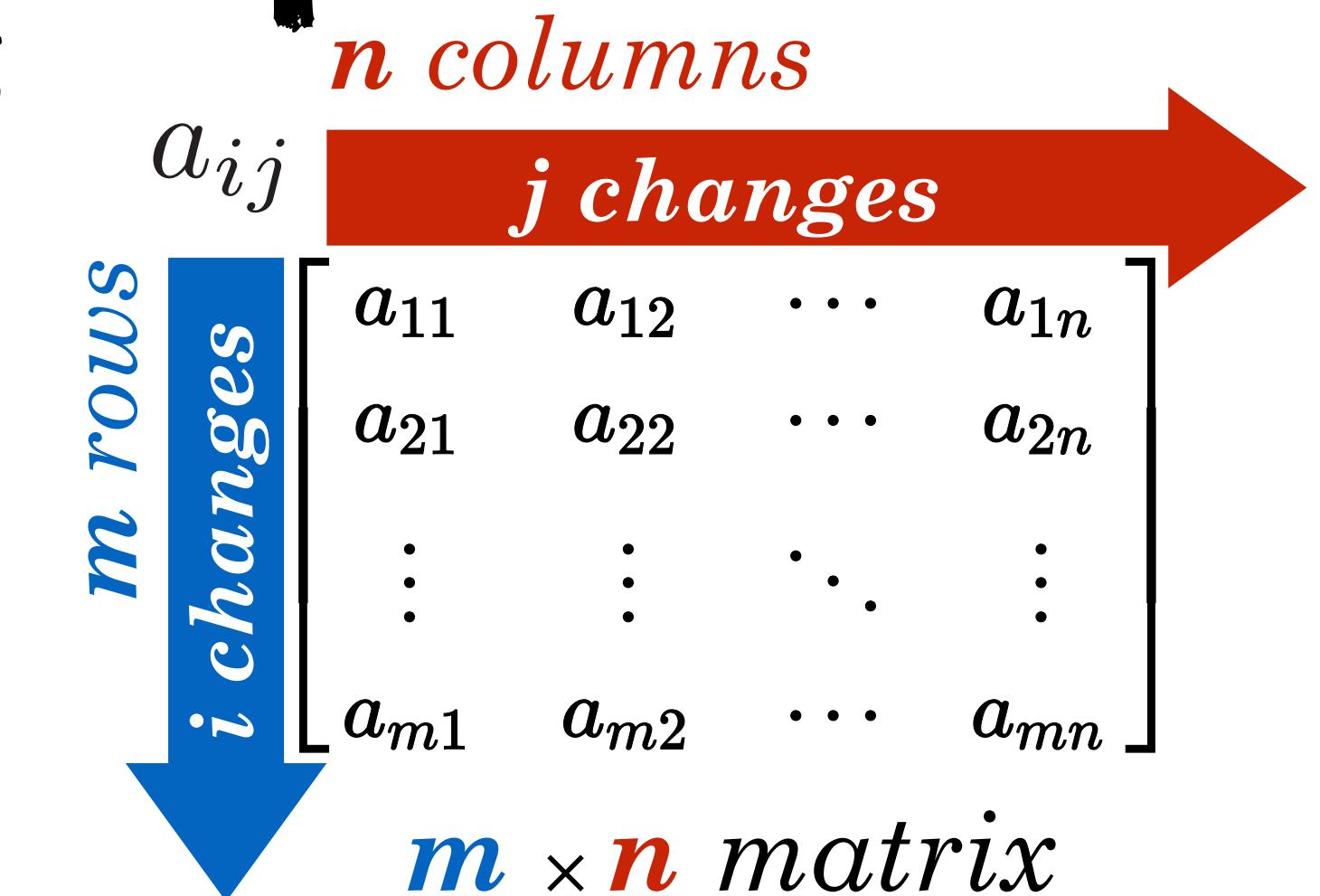
$3 \times 1$  matrix

$$\begin{bmatrix} 3 & 7 & 2 \end{bmatrix}$$

$1 \times 3$  matrix

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

$3 \times 3$  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

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

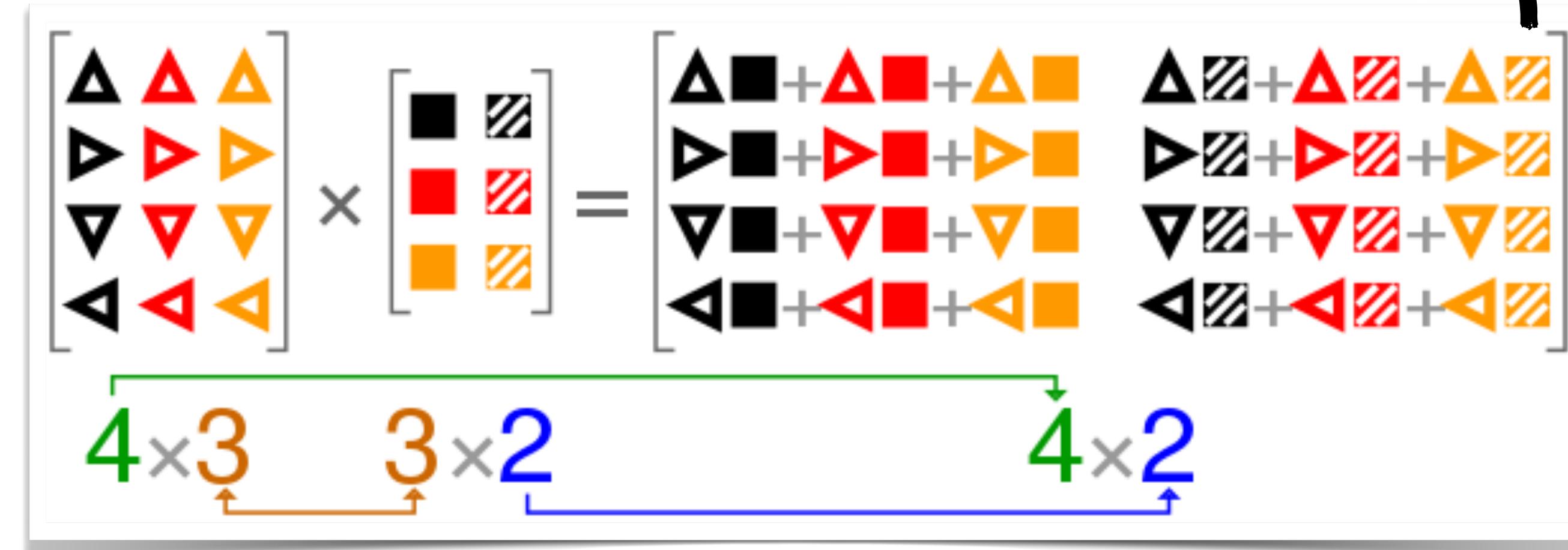
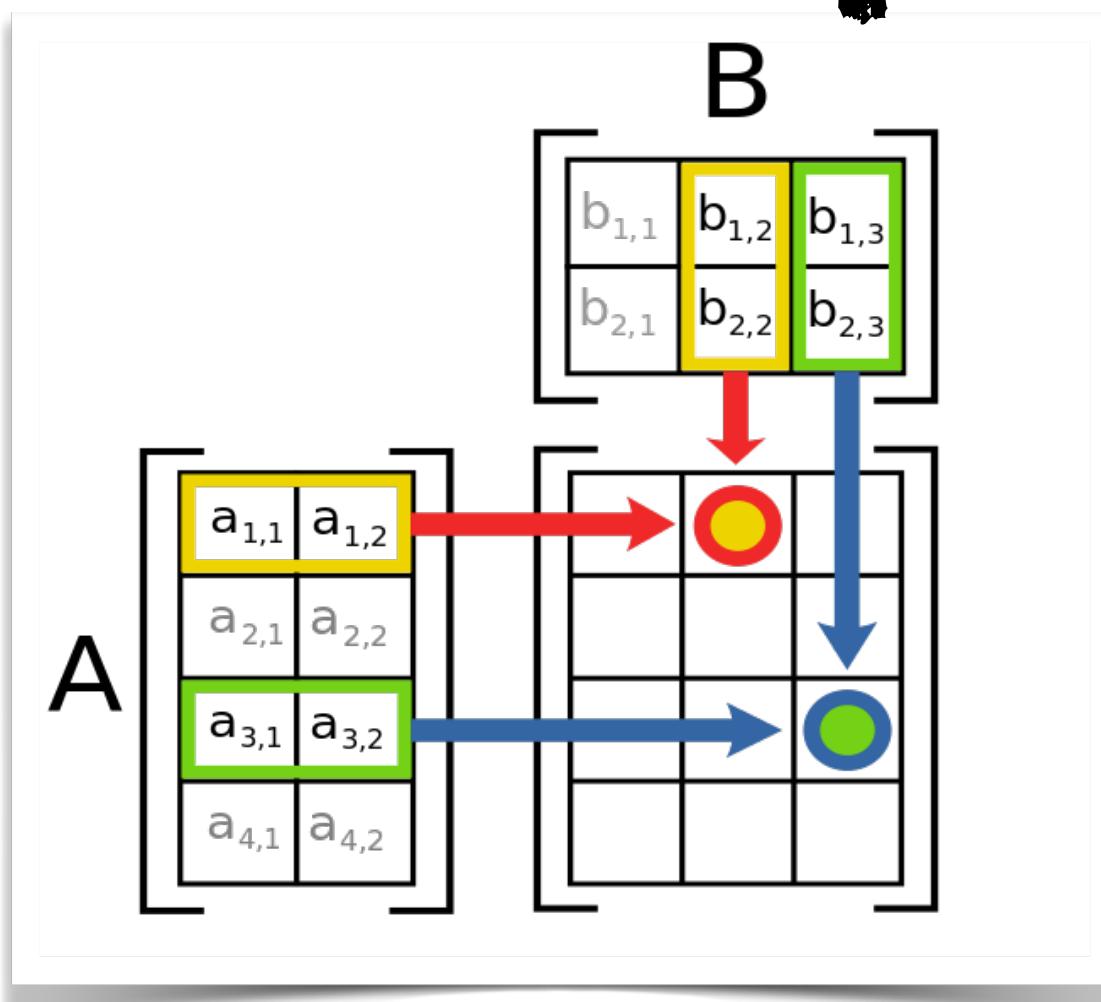
**addition**

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

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

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

# representing matrices multiplication



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

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

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

**AB ≠ BA**

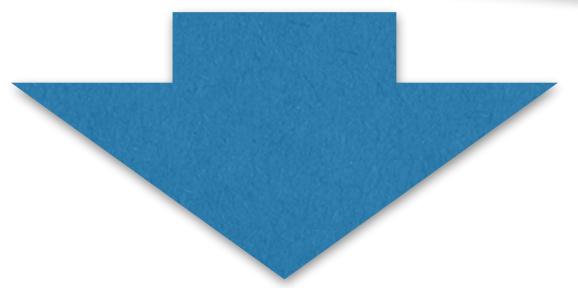
$$\mathbf{BA} = \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} = \begin{pmatrix} \alpha a + \beta p + \gamma u & \alpha b + \beta q + \gamma v & \alpha c + \beta r + \gamma w \\ \lambda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



```
public class ArrayMatrix {  
    final private int[][] matrix;  
    final int numOfRows;  
    final int numOfCols;
```

```
    public ArrayMatrix(int rows, int cols) {  
        if (rows <= 0 || cols <= 0) {  
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");  
        }  
        this.numOfRows = rows;  
        this.numOfCols = cols;  
        matrix = new int[rows][cols];  
    }  
    public ArrayMatrix(int rows, int cols, int[][] data) {  
        this(rows, cols);  
        for (int i = 0; i < rows; i++) {  
            for (int j = 0; j < cols; j++) {  
                this.set(i, j, data[i][j]);  
            }  
        }  
    }  
    public int getNumOfRows() { return numOfRows; }  
    public int getNumOfCols() { return numOfCols; }  
    public int get(int i, int j) { return matrix[i][j]; }  
    public void set(int i, int j, int v) { matrix[i][j] = v; }  
    public String toString() {  
        StringBuilder description = new StringBuilder();  
        description.append("\n").append(this.getClass().getSimpleName()).append("\n");  
        for (int i = 0; i < numOfRows; i++) {  
            description.append("| ");  
            for (int j = 0; j < numOfCols; j++) {  
                String entry = String.format("% 3d", this.get(i, j));  
                description.append(entry).append(" ");  
            }  
            description.append("|\n");  
        }  
        return description.toString();  
    }  
}
```

```
ArrayMatrix m = new ArrayMatrix(3, 3);  
System.out.println("m = " + m);  
  
int[][] data = {  
    {2, 4, 6},  
    {3, 6, 9}  
};  
  
m = new ArrayMatrix(2, 3, data);  
System.out.println("m = " + m);
```



```
m =  
ArrayMatrix  
| 0 0 0 |  
| 0 0 0 |  
| 0 0 0 |
```

```
m =  
ArrayMatrix  
| 2 4 6 |  
| 3 6 9 |
```

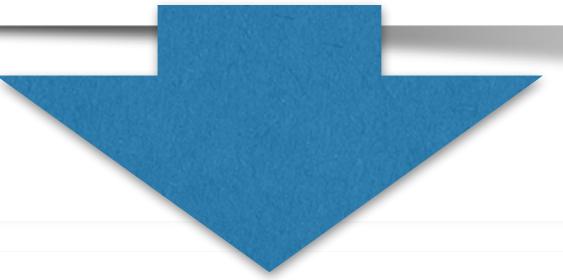
in the following, we assume that indices go from 0 to  $n-1$  rather than 1 to  $n$



```
public class ArrayMatrix {  
    final private int[][] matrix;  
    final int numOfRows;  
    final int numOfCols;  
  
    public ArrayMatrix(int rows, int cols) {  
        if (rows <= 0 || cols <= 0) {  
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");  
        }  
        this.numOfRows = rows;  
        this.numOfCols = cols;  
        matrix = new int[rows][cols];  
    }  
    public ArrayMatrix(int rows, int cols, int[][] data) {  
        this(rows, cols);  
        for (int i = 0; i < rows; i++) {  
            for (int j = 0; j < cols; j++) {  
                this.set(i, j, data[i][j]);  
            }  
        }  
    }  
    public int getNumOfRows() { return numOfRows; }  
    public int getNumOfCols() { return numOfCols; }  
    public int get(int i, int j) { return matrix[i][j]; }  
    public void set(int i, int j, int v) { matrix[i][j] = v; }  
    public String toString() {  
        StringBuilder description = new StringBuilder();  
        description.append("\n").append(this.getClass().getSimpleName()).append("\n");  
        for (int i = 0; i < numOfRows; i++) {  
            description.append("| ");  
            for (int j = 0; j < numOfCols; j++) {  
                String entry = String.format("% 3d", this.get(i, j));  
                description.append(entry).append(" ");  
            }  
            description.append("|\n");  
        }  
        return description.toString();  
    }  
}
```

# representing matrices

```
int[][] data = {  
    {2, 4, 6},  
    {3, 6, 9}  
};  
ArrayMatrix m = new ArrayMatrix(2, 3, data);  
System.out.println("m = " + m);  
  
System.out.println("m(0,0) = " + m.get(0, 0));  
m.set(0, 0, 7);  
System.out.println("m(0,0) = " + m.get(0, 0));  
  
System.out.println();  
System.out.println("m = " + m);
```



```
m =  
ArrayMatrix  
| 2 4 6 |  
| 3 6 9 |
```

```
m(0,0) = 2  
m(0,0) = 7
```

```
m =  
ArrayMatrix  
| 7 4 6 |  
| 3 6 9 |
```

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

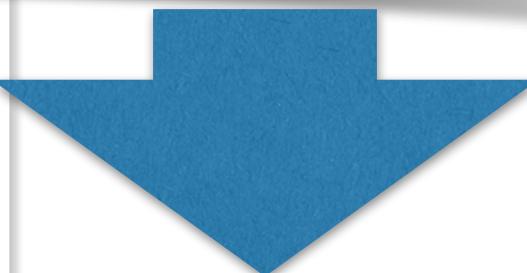


# representing matrices

```
public class ArrayMatrix {  
    private int[][] matrix;  
    final int numOfRows;  
    final int numOfCols;  
    :  
  
    public ArrayMatrix addTo(ArrayMatrix other) {  
        if (this.numOfRows != other.numOfRows || this.numOfCols != other.numOfCols) {  
            throw new IllegalArgumentException("Matrices must have the same number of rows and columns when added");  
        }  
  
        ArrayMatrix result = new ArrayMatrix(numOfRows, numOfCols);  
        for (int i = 0; i < numOfRows; i++) {  
            for (int j = 0; j < numOfCols; j++) {  
                result.set(i, j, this.get(i, j) + other.get(i, j));  
            }  
        }  
  
        return result;  
    }  
  
    public ArrayMatrix multiplyBy(ArrayMatrix other) {  
        if (this.numOfCols != other.getNumOfRows()) {  
            throw new IllegalArgumentException("Matrices must have compatible number of rows and columns when multiplied");  
        }  
  
        ArrayMatrix result = new ArrayMatrix(this.numOfRows, other.getNumOfCols());  
        for (int i = 0; i < result.getNumOfRows(); i++) {  
            for (int j = 0; j < result.getNumOfCols(); j++) {  
                int entry = 0;  
                for (int k = 0; k < this.numOfCols; k++) {  
                    entry = entry + this.get(i, k) * other.get(k, j);  
                }  
                result.set(i, j, entry);  
            }  
        }  
  
        return result;  
    }  
}
```

```
int[][] data1 = {  
    { 1, 2, -4},  
    {-1, -3, 3}  
};  
int[][] data2 = {  
    { 0, 5, 4},  
    {-2, -3, 6},  
    { 1, 2, 3}  
};
```

```
ArrayMatrix m1 = new ArrayMatrix(2, 3, data1);  
ArrayMatrix m2 = new ArrayMatrix(3, 3, data2);  
ArrayMatrix m3 = m2.addTo(m2);  
System.out.println("m3 = m2 + m2 = " + m3);  
  
m3 = m1.multiplyBy(m2);  
System.out.println("m3 = m1 * m2 = " + m3);
```



```
m3 = m2 + m2 =  
ArrayMatrix  
| 0 10 8 |  
| -4 -6 12 |  
| 2 4 6 |
```

```
m3 = m1 * m2 =  
ArrayMatrix  
| -8 -9 4 |  
| 9 10 -13 |
```

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

# sparse matrices



```

public class MapMatrix {
    final Map<Index, Integer> matrix;
    final int numberOfRows;
    final int numberOfCols;

    public MapMatrix(int rows, int cols) {
        if (rows <= 0 || cols <= 0) {
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");
        }
        this.numberOfRows = rows;
        this.numberOfCols = cols;
        matrix = new HashMap();
    }
    public MapMatrix(int rows, int cols, Map<Index, Integer> data) {
        this(rows, cols);
        for (Index index : data.keySet()) {
            if (index.i < rows && index.j < cols) {
                this.set(index.i, index.j, data.get(index));
            }
        }
    }
    public int getNumberOfRows() { return numberOfRows; }
    public int getNumberOfCols() { return numberOfCols; }
    public int get(int i, int j) {
        Index index = Index.from(i, j);
        return matrix.containsKey(index) ? matrix.get(index) : 0;
    }
    public void set(int i, int j, int v) { matrix.put(Index.from(i, j), v); }

    public String toString() {
        StringBuilder description = new StringBuilder();
        description.append("\n").append(this.getClass().getSimpleName()).append("\n");
        for (int i = 0; i < numberOfRows; i++) {
            description.append(" | ");
            for (int j = 0; j < numberOfCols; j++) {
                String entry = String.format("% 3d", this.get(i, j));
                description.append(entry).append(" ");
            }
            description.append(" |\n");
        }
        return description.toString();
    }
}

```

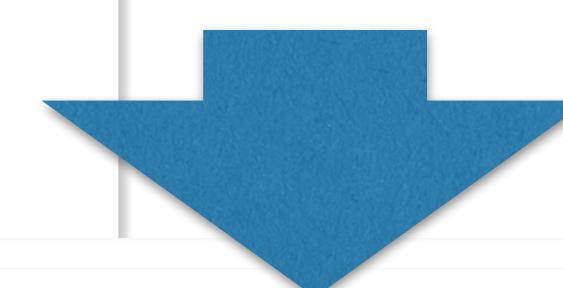
```

MapMatrix m = new MapMatrix(3, 3);
System.out.println("m = " + m);

Map<Index, Integer> data = new HashMap();
data.put(Index.from(2, 4), 66);
data.put(Index.from(0, 2), 44);

m = new MapMatrix(5, 5, data);
System.out.println("m = " + m);

```



m =	MapMatrix					
		0	0	0		
		0	0	0		
		0	0	0		
m =	MapMatrix					
		0	0	44	0	0
		0	0	0	0	0
		0	0	0	0	66
		0	0	0	0	0
		0	0	0	0	0

# sparse matrices



```
public class MapMatrix {  
    final Map<Index, Integer> matrix;  
    final int numOfRows;  
    final int numOfCols;  
  
    public MapMatrix(int rows, int cols) {  
        if (rows <= 0 || cols <= 0) {  
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");  
        }  
        this.numOfRows = rows;  
        this.numOfCols = cols;  
        matrix = new HashMap();  
    }  
    public MapMatrix(int rows, int cols, Map<Index, Integer> data) {  
        this(rows, cols);  
        for (Index index : data.keySet()) {  
            if (index.i < rows && index.j < cols) {  
                this.set(index.i, index.j, data.get(index));  
            }  
        }  
    }  
}
```

```
public class Index {  
    public final int i;  
    public final int j;  
    public Index(int i, int j) {  
        this.i = i;  
        this.j = j;  
    }  
    public static Index from(int i, int j) {  
        return new Index(i, j);  
    }  
    public int hashCode() {  
        int hash = 7;  
        hash = 83 * hash + this.i;  
        hash = 83 * hash + this.j;  
        return hash;  
    }  
    ...  
}
```

```
MapMatrix m = new MapMatrix(3, 3);  
System.out.println("m = " + m);  
  
Map<Index, Integer> data = new HashMap();  
data.put(Index.from(2, 4), 66);  
data.put(Index.from(0, 2), 44);  
  
m = new MapMatrix(5, 5, data);  
System.out.println("m = " + m);
```

```
...  
    public boolean equals(Object obj) {  
        if (this == obj) {  
            return true;  
        }  
        if (obj == null) {  
            return false;  
        }  
        if (getClass() != obj.getClass()) {  
            return false;  
        }  
        final Index other = (Index) obj;  
        if (this.i != other.i) {  
            return false;  
        }  
        if (this.j != other.j) {  
            return false;  
        }  
        return true;  
    }
```



# problem

```
public class MapMatrix {  
    final Map<Index, Integer> matrix;  
    final int numberOfRows;  
    final int numberOfCols;  
  
    public MapMatrix(int rows, int cols) {  
        if (rows <= 0 || cols <= 0) {  
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");  
        }  
        this.numberOfRows = rows;  
        this.numberOfCols = cols;  
        matrix = new HashMap<>();  
    }  
    public MapMatrix(int rows, int cols, Map<Index, Integer> data) {  
        this(rows, cols);  
        for (Index index : data.keySet()) {  
            if (index.i < rows && index.j < cols) {  
                this.set(index.i, index.j, data.get(index));  
            }  
        }  
    }  
    public int getNumberOfRows() { return numberOfRows; }  
    public int getNumberOfCols() { return numberOfCols; }  
    public int get(int i, int j) {  
        Index index = Index.from(i, j);  
        return matrix.containsKey(index) ? matrix.get(index) : 0;  
    }  
    public void set(int i, int j, int v) { matrix.put(Index.from(i, j), v); }  
  
    public String toString() {  
        StringBuilder description = new StringBuilder();  
        description.append("\n").append(this.getClass().getSimpleName()).append("\n");  
        for (int i = 0; i < numberOfRows; i++) {  
            description.append(" | ");  
            for (int j = 0; j < numberOfCols; j++) {  
                String entry = String.format("% 3d", this.get(i, j));  
                description.append(entry).append(" ");  
            }  
            description.append(" | \n");  
        }  
        return description.toString();  
    }  
    ...  
}
```

```
public class ArrayMatrix {  
    private int[][] matrix;  
    final int numberOfRows;  
    final int numberOfCols;  
  
    public ArrayMatrix(int rows, int cols) {  
        if (rows <= 0 || cols <= 0) {  
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");  
        }  
        this.numberOfRows = rows;  
        this.numberOfCols = cols;  
        matrix = new int[rows][cols];  
    }  
    public ArrayMatrix(int rows, int cols, int[][] data) {  
        this(rows, cols);  
        for (int i = 0; i < rows; i++) {  
            for (int j = 0; j < cols; j++) {  
                this.set(i, j, data[i][j]);  
            }  
        }  
    }  
    public int getNumberOfRows() { return numberOfRows; }  
    public int getNumberOfCols() { return numberOfCols; }  
    public int get(int i, int j) { return matrix[i][j]; }  
  
    public void set(int i, int j, int v) { matrix[i][j] = v; }  
  
    public String toString() {  
        StringBuilder description = new StringBuilder();  
        description.append("\n").append(this.getClass().getSimpleName()).append("\n");  
        for (int i = 0; i < numberOfRows; i++) {  
            description.append(" | ");  
            for (int j = 0; j < numberOfCols; j++) {  
                String entry = String.format("% 3d", this.get(i, j));  
                description.append(entry).append(" ");  
            }  
            description.append(" | \n");  
        }  
        return description.toString();  
    }  
    ...  
}
```

code duplication... again!



# problem

## can we blend array and map matrices?

```
public ArrayMatrix multiplyBy(ArrayMatrix other) {  
    if (this.numOfCols != other.getNumberOfRows()) {  
        throw new IllegalArgumentException("Matrices must have compatible number of rows and columns when multiplied");  
    }  
    ArrayMatrix result = new ArrayMatrix(this.numberOfLines, other.getNumberOfCols());  
    for (int i = 0; i < result.getNumberOfRows(); i++) {  
        for (int j = 0; j < result.getNumberOfCols(); j++) {  
            int entry = 0;  
            for (int k = 0; k < this.numberOfLines; k++) {  
                entry = entry + this.get(i, k) * other.get(k, j);  
            }  
            result.set(i, j, entry);  
        }  
    }  
    return result;  
}  
  
public MapMatrix multiplyBy(MapMatrix other) {  
    if (this.numOfCols != other.getNumberOfRows()) {  
        throw new IllegalArgumentException("Matrices must have compatible number of rows and columns when multiplied");  
    }  
    MapMatrix result = new MapMatrix(this.numberOfLines, other.getNumberOfCols());  
    for (int i = 0; i < result.getNumberOfRows(); i++) {  
        for (int j = 0; j < result.getNumberOfCols(); j++) {  
            int entry = 0;  
            for (int k = 0; k < this.numberOfLines; k++) {  
                entry = entry + this.get(i, k) * other.get(k, j);  
            }  
            result.set(i, j, entry);  
        }  
    }  
    return result;  
}
```

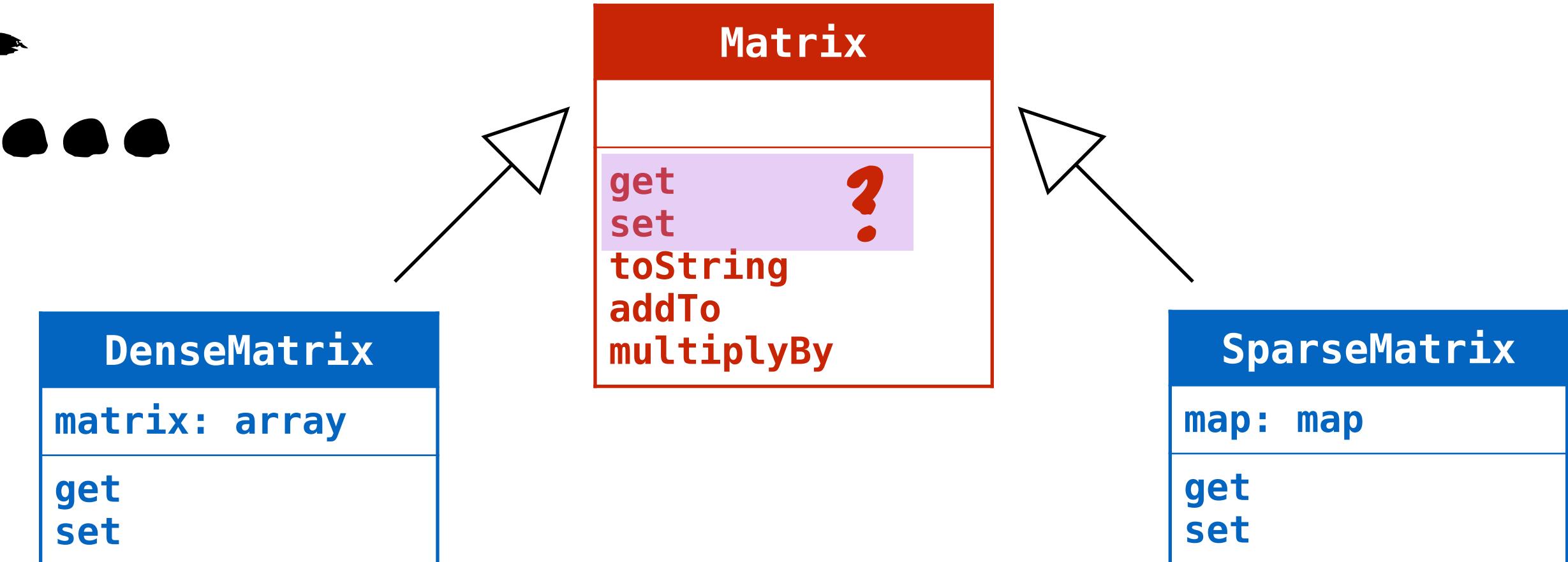


incompatible types

# solution inheritance?



# yes, but...



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

so methods `get` and `set` have no default or shared implementation

these methods are **abstract** in the superclass



# solution

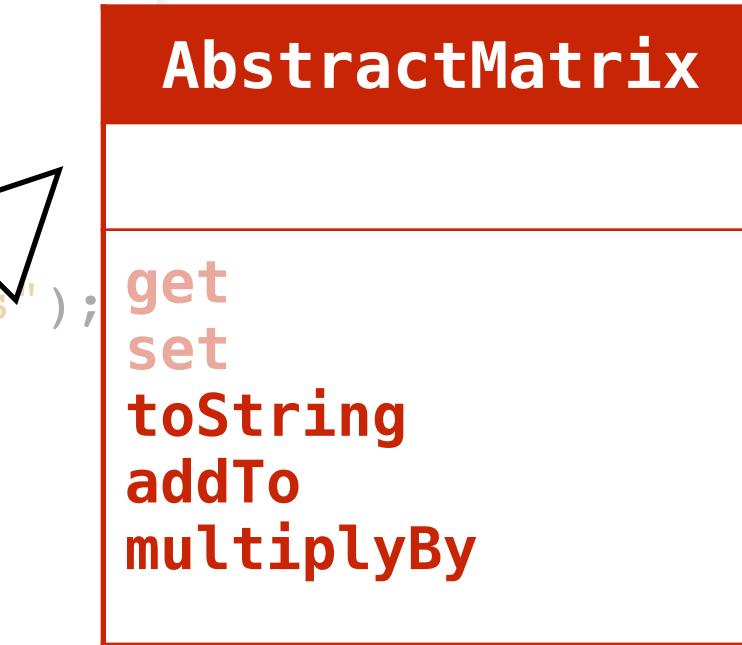


# abstract class!



```
public abstract class AbstractMatrix {  
    final int numberOfRows;  
    final int numberOfCols;  
  
    public AbstractMatrix(int rows, int cols) {  
        if (rows <= 0 || cols <= 0) {  
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");  
        }  
        this.numberOfRows = rows;  
        this.numberOfCols = cols;  
    }  
    public abstract int get(int i, int j);  
    public abstract void set(int i, int j, int v);  
  
    public int getNumberOfRows() { return numberOfRows; }  
    public int getNumberOfCols() { return numberOfCols; }  
  
    final protected void init(int[][] data) {  
        for (int i = 0; i < data.length; i++) {  
            for (int j = 0; j < data[i].length; j++) {  
                this.set(i, j, data[i][j]);  
            }  
        }  
    }  
    final protected void init(Map<Index, Integer> data) {  
        for (Index index : data.keySet()) {  
            if (index.i < numberOfRows && index.j < numberOfRows) {  
                this.set(index.i, index.j, data.get(index));  
            }  
        }  
    }  
    public String toString() {  
        StringBuilder description = new StringBuilder();  
  
        description.append("\n").append(this.getClass().getSimpleName());  
        for (int i = 0; i < numberOfRows; i++) {  
            description.append(" | ");  
            for (int j = 0; j < numberOfCols; j++) {  
                String entry = String.format("% 3d", matrix.get(i, j));  
                description.append(entry).append(" ");  
            }  
            description.append(" |\n");  
        }  
        return description.toString();  
    }  
};
```

# abstract class



```
:  
    public AbstractMatrix addTo(AbstractMatrix other) {  
        if (this.numberOfRows != other.numberOfRows || this.numberOfCols != other.numberOfCols) {  
            throw new IllegalArgumentException("Matrices must have the same number of rows and columns");  
        }  
  
        DenseMatrix result = new DenseMatrix(numberOfRows, numberOfCols);  
        for (int i = 0; i < numberOfRows; i++) {  
            for (int j = 0; j < numberOfCols; j++) {  
  
                result.set(i, j, this.get(i, j) + other.get(i, j));  
            }  
        }  
        return result;  
    }  
:
```

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

could we come with a better scheme?



# abstract class

```
public class SparseMatrix extends AbstractMatrix {  
    final Map<Index, Integer> matrix;  
  
    public SparseMatrix(int rows, int cols) {  
        super(rows, cols);  
        matrix = new HashMap();  
    }  
    public SparseMatrix(int rows, int cols, int[][] data) {  
        this(rows, cols);  
        init(data);  
    }  
    public SparseMatrix(int rows, int cols, Map<Index, Integer> data) {  
        this(rows, cols);  
        init(data);  
    }  
    public int get(int i, int j) {  
        Index index = Index.from(i, j);  
        if (matrix.containsKey(index)) {  
            return matrix.get(index);  
        } else {  
            return 0;  
        }  
    }  
    public void set(int i, int j, int v) {  
        matrix.put(Index.from(i, j), v);  
    }  
}
```

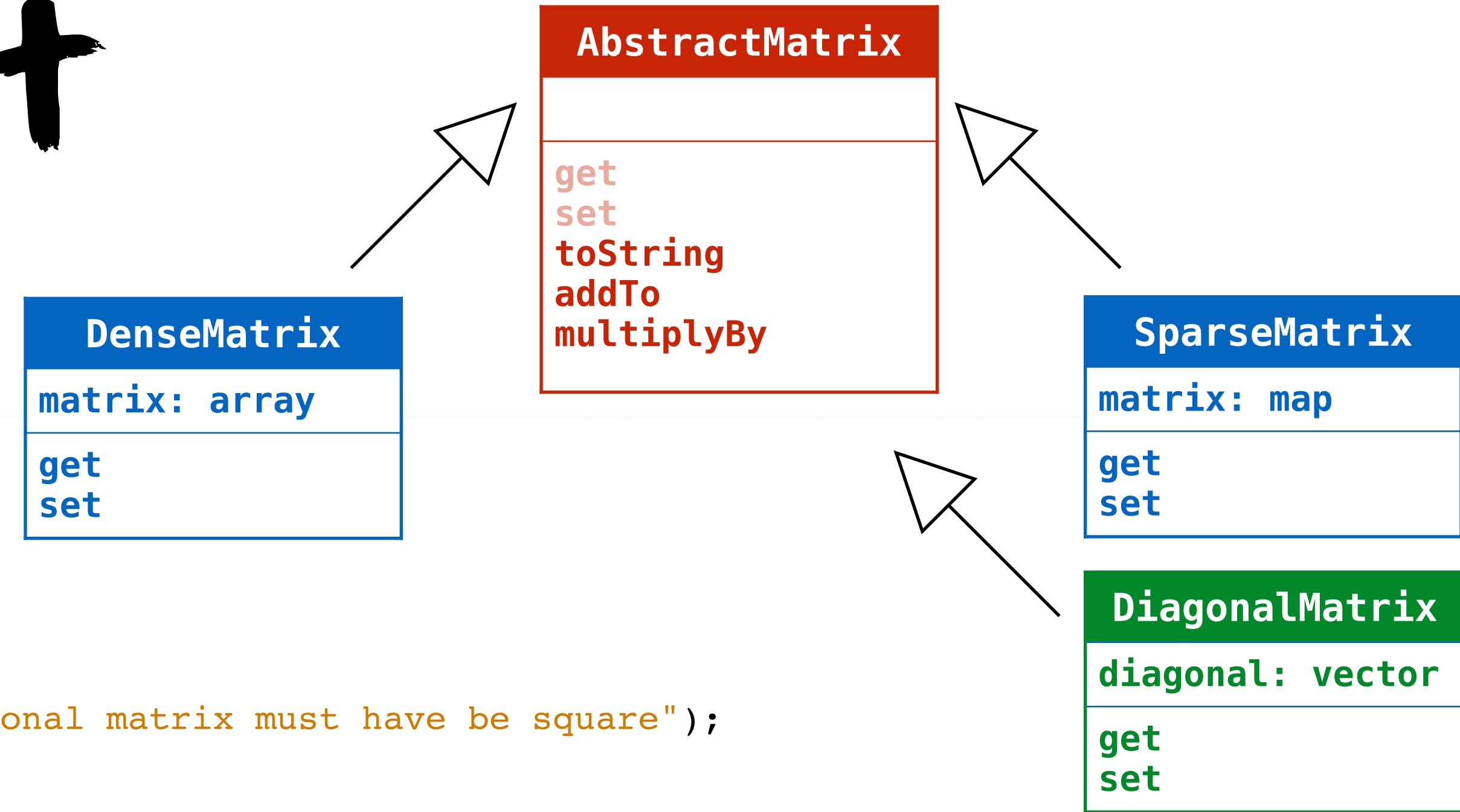


```
public class DenseMatrix extends AbstractMatrix {  
    private int[][] matrix;  
  
    public DenseMatrix(int rows, int cols) {  
        super(rows, cols);  
        matrix = new int[rows][cols];  
    }  
    public DenseMatrix(int rows, int cols, int[][] data) {  
        this(rows, cols);  
        init(data);  
    }  
    public DenseMatrix(int rows, int cols, Map<Index, Integer> data) {  
        this(rows, cols);  
        init(data);  
    }  
    public int get(int i, int j) {  
        return matrix[i][j];  
    }  
    public void set(int i, int j, int v) {  
        matrix[i][j] = v;  
    }  
}
```

# abstract class



```
public class DiagonalMatrix extends AbstractMatrix {  
    private int[] diagonal;  
  
    public DiagonalMatrix(int rows, int cols) {  
        super(rows, cols);  
        if (rows != cols) {  
            throw new IllegalArgumentException("A diagonal matrix must have be square");  
        }  
        diagonal = new int[rows];  
    }  
    public DiagonalMatrix(int rows, int cols, int[] diagonal) {  
        this(rows, cols);  
        int limit = Math.min(rows, diagonal.length);  
        for (int i = 0; i < limit; i++) {  
            this.diagonal[i] = diagonal[i];  
        }  
    }  
    public int get(int i, int j) {  
        return (i == j) ? diagonal[i] : 0;  
    }  
    public void set(int i, int j, int v) {  
        if (i == j) {  
            diagonal[i] = v;  
        } else {  
            throw new IllegalArgumentException("A diagonal matrix should only have zeros outside its diagonal");  
        }  
    }  
}
```



# what if all methods could be abstract?

identity matrix

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

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

```
public class UnityMatrix extends AbstractMatrix {  
    public UnityMatrix(int size) {  
        super(size, size);  
    }  
    public int get(int i, int j) { return (i == j) ? 1 : 0; }  
    public void set(int i, int j, int v) { throw new UnsupportedOperationException("Not supported: a unity matrix is constant!"); }  
}
```

```
public AbstractMatrix addTo(AbstractMatrix other) {  
    AbstractMatrix result = other.clone();  
    for (int i = 0; i < numOfRows; i++) {  
        result.set(i, i, other.get(i, i) + 1);  
    }  
    return result;  
}  
public AbstractMatrix multiplyBy(AbstractMatrix other) {  
    return other.clone();  
}
```



we have to add a `clone()` method to all classes  
(`clone` = deep copy)



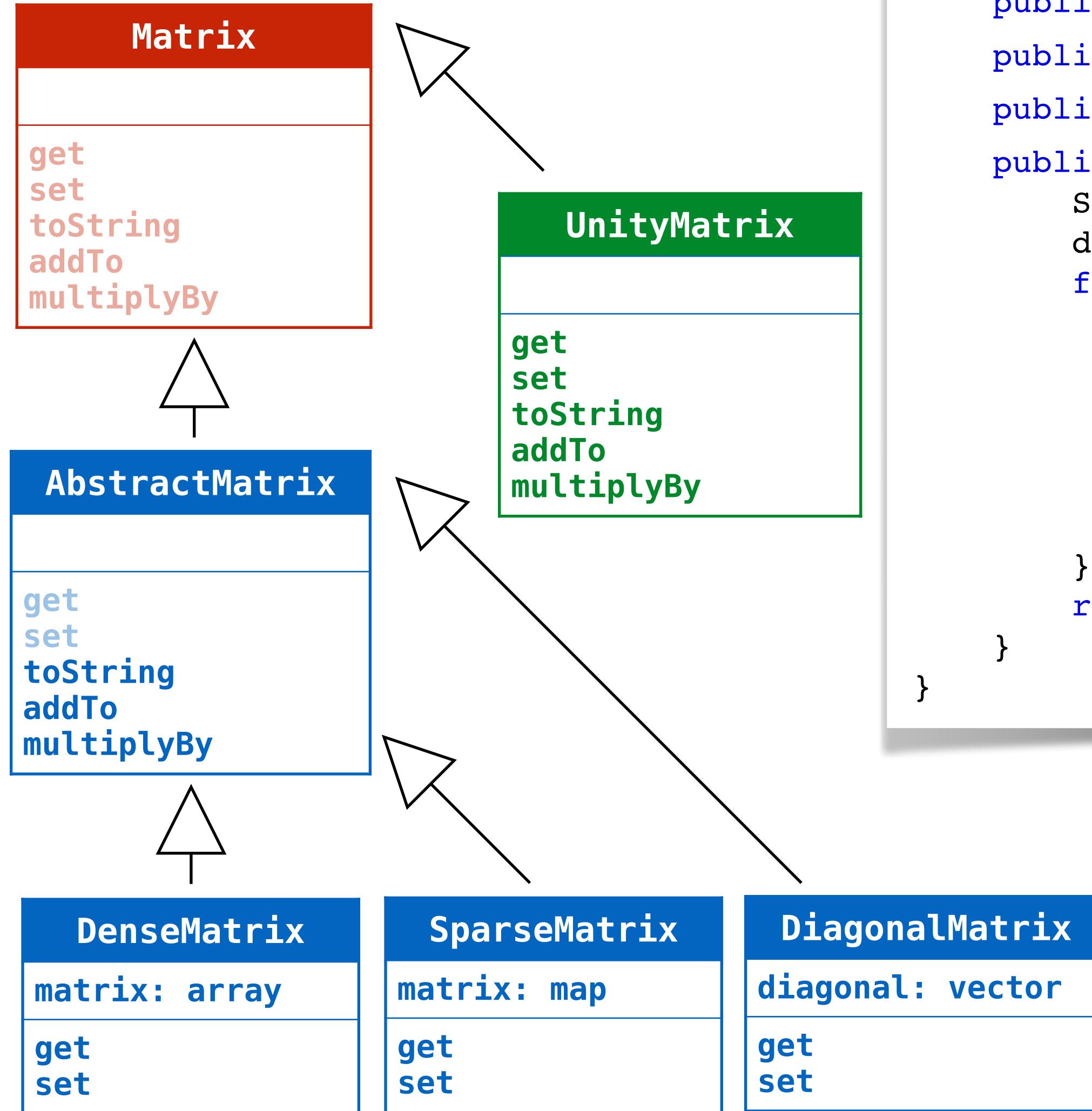
solution



interfaces

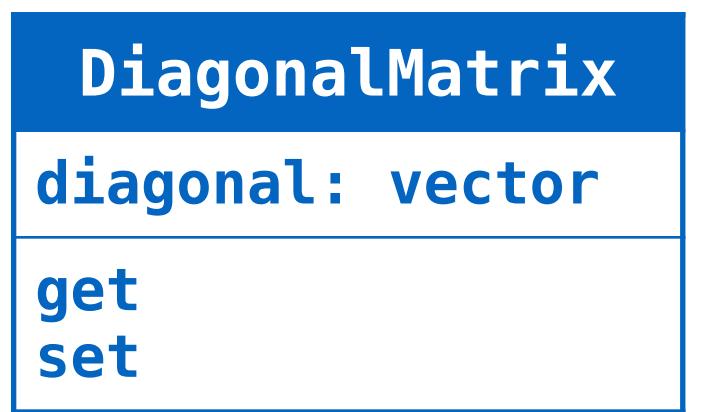
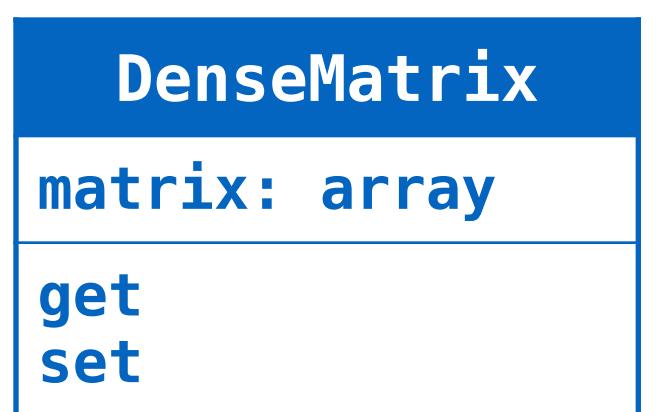
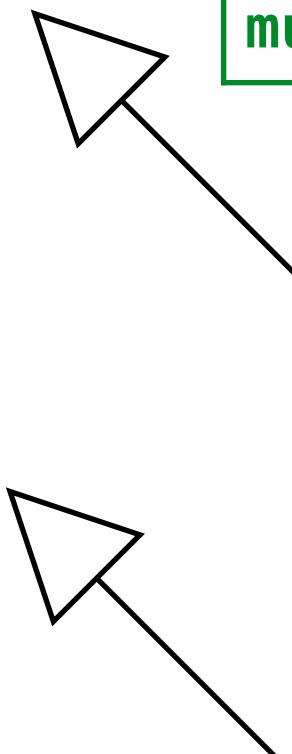
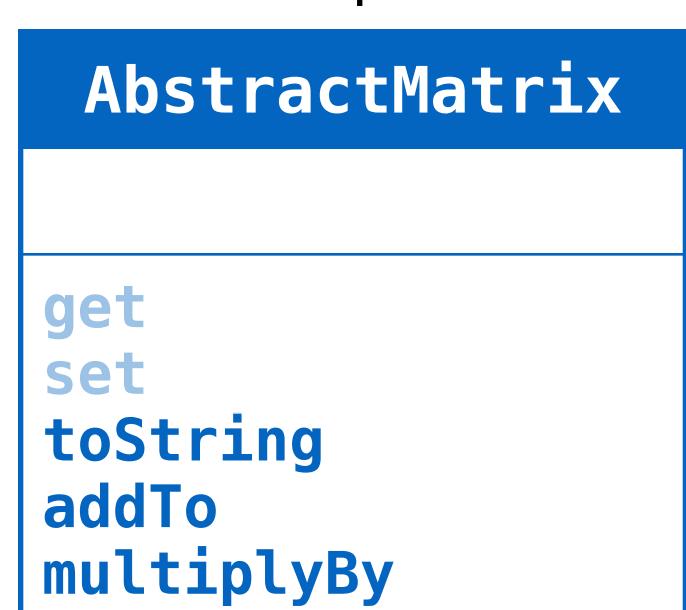
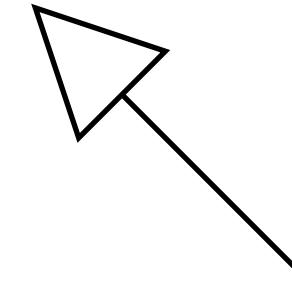


# interface



```
public interface Matrix {  
    public Matrix addTo(Matrix other);  
    public Matrix multiplyBy(Matrix other);  
    public int get(int i, int j);  
    public void set(int i, int j, int v);  
    public int getNumberOfRows();  
    public int getNumberOfCols();  
    public Matrix clone();  
    public static String toString(Matrix matrix) {  
        StringBuilder description = new StringBuilder();  
        description.append("\n").append(matrix.getClass().getSimpleName()).append("\n");  
        for (int i = 0; i < matrix.getNumberOfRows(); i++) {  
            description.append(" | ");  
            for (int j = 0; j < matrix.getNumberOfCols(); j++) {  
                String entry = String.format("% 3d", matrix.get(i, j));  
                description.append(entry).append(" ");  
            }  
            description.append(" | \n");  
        }  
        return description.toString();  
    }  
}  
  
abstract class AbstractMatrix implements Matrix { ... }  
  
class DenseMatrix extends AbstractMatrix { ... }  
  
class SparseMatrix extends AbstractMatrix { ... }  
  
class DiagonalMatrix extends AbstractMatrix { ... }  
  
class UnityMatrix implements Matrix { ... }
```

# interface



```

public class UnityMatrix extends AbstractMatrix {
    int size;
    public UnityMatrix(int size) {
        if (size <= 0) {
            throw new IllegalArgumentException("A matrix must have a positive number of rows and columns");
        }
        this.size = size;
    }
    public String toString() { return Matrix.toString(this); }
    public int getNumOfRows() { return size; }
    public int getNumOfCols() { return size; }
    public Matrix clone() { return this; }
}
  
```

```

public abstract class AbstractMatrix implements Matrix {
    private final int numRows;
    private final int numCols;

    final protected void init(int[][] data) {
    final protected void init(Map<Index, Integer> data) { ... }
    public Matrix addTo(Matrix other) { ... }
    public Matrix multiplyBy(Matrix other) { ... }
    if (this.numCols != other.getNumOfRows()) {
    public int getNumOfRows() { return numRows; }
    public int getNumOfCols() { return numCols; }
    public abstract int get(int i, int j);
    public abstract void set(int i, int j, int v);
    public abstract Matrix clone();
    public String toString() {
        return Matrix.toString(this);
    }
}
  
```

```

public class SparseMatrix extends AbstractMatrix {
    private final Map<Index, Integer> matrix;
    ...
    public Matrix clone() {
        return new SparseMatrix(getNumOfRows(), getNumOfCols(), matrix);
    }
}
  
```

```

public class DiagonalMatrix extends AbstractMatrix {
    private final int[] diagonal;
    ...
    public Matrix clone() {
        return new DiagonalMatrix(getNumOfRows(), diagonal);
    }
}
  
```

# 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

in **java**, an **interface** forces you to define types without an implementation

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

in **python**, these notions have **no equivalent** due to the absence of static typing