9. Creating Data Types

- **Overview**
- Point charges
- Turtle graphics
- Complex numbers

### Basic building blocks for programming

Any program you might want to write

- **Objects**
- Functions and modules
- Graphics, sound, and image I/O
- Arrays
- Conditional and loops
- Math
- Text I/O
- Primitive data types
- Assignment statements

### Object-oriented programming (OOP)

Object-oriented programming (OOP).

- Create your own data types.
- Use them in your programs (manipulate objects).

**Examples**

<table>
<thead>
<tr>
<th>data type</th>
<th>set of values</th>
<th>examples of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>three 8-bit integers</td>
<td>get red component, brighten</td>
</tr>
<tr>
<td>Picture</td>
<td>2D array of colors</td>
<td>get/set color of pixel</td>
</tr>
<tr>
<td>String</td>
<td>sequence of characters</td>
<td>length, substring, compare</td>
</tr>
</tbody>
</table>

An abstract data type is a data type whose representation is *hidden from the client*.

**Impact:**
- Previous lecture: how to write client programs for several useful ADTs
- This lecture: how to implement your own ADTs
Implementing a data type

To create a data type, you need to provide code that
• Defines the set of values (instance variables).
• Implements operations on those values (methods).
• Creates and initializes new objects (constructors).

Instance variables
• Declarations associate variable names with types.
• Set of type values is “set of values”.

Methods
• Like static methods.
• Can refer to instance variables.

Constructors
• Like a method with the same name as the type.
• No return type declaration.
• Invoked by new, returns object of the type.

In Java, a data-type implementation is known as a class.

A Java class

<table>
<thead>
<tr>
<th>Instance variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructors</td>
</tr>
<tr>
<td>methods</td>
</tr>
<tr>
<td>test client</td>
</tr>
</tbody>
</table>

Anatomy of a Class

```java
public class Charge {
    private final double rx, ry; // position
    private final double q; // charge value

    public Charge(double x0, double y0, double q0) {
        rx = x0;
        ry = y0;
        q = q0;
    }

    public double potentialAt(double x, double y) {
        double k = 8.99e9;
        double dx = x - rx;
        double dy = y - ry;
        return k * q / Math.sqrt(dx*dx + dy*dy);
    }

    public String toString() {
        return q + " at " + rx + ", " + ry + ";
    }

    public static void main(String[] args) {
        Charge c = new Charge(.72, .35, 21.1);
        StdOut.println(c);
        StdOut.printf("%d.2f\n", c.potentialAt(.42, .71));
    }
}
```

Test client

```java
% java Charge
21.3 at (0.72, 0.35)
1.63e+11
```
ADT for point charges

A point charge is an idealized model of a particle that has an electric charge. An ADT allows us to write Java programs that manipulate point charges.

<table>
<thead>
<tr>
<th>Values</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>position (x, y)</td>
<td>(.53, .63) (.13, .94)</td>
</tr>
<tr>
<td>electrical charge</td>
<td>20.1 81.9</td>
</tr>
</tbody>
</table>

API (operations)

<table>
<thead>
<tr>
<th></th>
<th>Charge(double x0, double y0, double q0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>double potential</td>
<td>(double x, double y) electric potential at (x, y) due to charge</td>
</tr>
<tr>
<td>String toString()</td>
<td>string representation of this charge</td>
</tr>
</tbody>
</table>

public class Charge
{
    Charge(double x0, double y0, double q0)
    double potential(double x, double y) electric potential at (x, y) due to charge
    String toString() string representation of this charge

Point charge implementation: Test client

Best practice. Begin by implementing a simple test client.

```java
public static void main(String[] args)
{
    Charge c = new Charge(.72, .31, 20.1);
    StdOut.println(c);
    Reminder: automatically invokes c.toString()
    StdOut.printf("%6.2e\n", c.potentialAt(.42, .71));
}
```

\[ V_c(x, y) = \frac{k q}{r} \]
\[ r = \sqrt{(r_x - x)^2 + (r_y - y)^2} = \sqrt{3^2 + 4^2} = 5 \]
\[ V_c(0.42, 0.71) = 8.99 \times 10^9 \frac{20.1}{5} = 3.6 \times 10^{11} \]

What we expect, once the implementation is done.

Point charge implementation: Instance variables

Instance variables define data-type values.

<table>
<thead>
<tr>
<th>Values</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>position (x, y)</td>
<td>(.53, .63) (.13, .94)</td>
</tr>
<tr>
<td>electrical charge</td>
<td>20.1 81.9</td>
</tr>
</tbody>
</table>

public class Charge
{
    private final double rx, ry;
    private final double q;
    ...
}

Modifiers control access.
- private denies clients access and therefore makes data type abstract.
- final disallows any change in value and documents that data type is immutable.

Key to OOP. Each object has instance-variable values.

Crash course on electric potential

Electric potential is a measure of the effect of a point charge on its surroundings.
- It increases in proportion to the charge value.
- It decreases in proportion to the inverse of the distance from the charge (2D).

Mathematically,
- Suppose a point charge \( q \) is located at \((x, y)\) and has charge \( q \).
- Let \( r \) be the distance between \((x, y)\) and \((r_x, r_y)\).
- Then \[ V_c(x, y) = \frac{k q}{r} \] where \( k = 8.99 \times 10^9 \) is a normalizing factor.

Q. What happens when multiple charges are present?

A. The potential at a point is the sum of the potentials due to the individual charges.

Note: Similar laws hold in many other situations.

Example. N-body (3D) is an inverse square law.
### Point charge implementation: Constructor

**Constructors** create and initialize new objects.

```java
public class Charge
{
    private final double rx, ry; // position
    private final double q; // charge value

    public Charge(double x0, double y0, double q0)
    {
        rx = x0;
        ry = y0;
        q = q0;
    }
}
```

Clients use `new` to invoke constructors.
- Pass arguments as in a method call.
- Return value is reference to new object.

Possible memory representation of Charge $c = \text{new } \text{Charge}(.72, .31, 20.1)$:

![Memory address diagram](image)

References to instance variables, which are not declared within the constructor.

### Point charge implementation: Methods

**Methods** define data-type operations (implement APIs).

```java
public class Charge
{
    public static Charge
    {
        double double potentialAt(double x, double y)
        {
            double k = 8.99x10^9;
            double dx = x - rx;
            double dy = y - ry;
            return k * q / Math.sqrt(dx^2 + dy^2);
        }
    }
}
```

**API**
- `Charge(double x0, double y0, double q0)`
- `double potentialAt(double x, double y)`
- `String toString()`

**Key to OOP:** An instance variable reference in an instance method refers to the value for the object that was used to invoke the method.

### Point charge client: Potential visualization (helper methods)

- Convert potential values to a color.
  - Convert $V$ to an 8-bit integer.
  - Use grayscale.

```java
public static Color toColor(double V)
{
    V = 128 + V / 2.0x10;
    if (V > 255) t = 255;
    else if (V >= 0) t = (int) V;
    return new Color(t, t, t);
}
```

**Point charge client: Potential visualization (helper methods)**

- Read point charges from StdIn.
  - Uses `Charge` like any other type.
  - Returns an array of charges.

```java
public static Charge[] readCharges()
{
    int N = StdIn.readInt();
    Charge[] a = new Charge[N];
    for (int i = 0; i < N; i++)
    {
        double x0 = StdIn.readDouble();
        double y0 = StdIn.readDouble();
        double q0 = StdIn.readDouble();
        a[i] = new Charge(x0, y0, q0);
    }
    return a;
}
```

**Convert potential values to a color.**
- Convert $V$ to an 8-bit integer.
- Use grayscale.

```java
public static Charge[] readCharges()
{
    int N = StdIn.readInt();
    Charge[] a = new Charge[N];
    for (int i = 0; i < N; i++)
    {
        double x0 = StdIn.readDouble();
        double y0 = StdIn.readDouble();
        double q0 = StdIn.readDouble();
        a[i] = new Charge(x0, y0, q0);
    }
    return a;
}
```
import java.awt.Color;
public class Potential {
    public static Charge[] readCharges()
        // See previous slide.
        public static Color toColor() { // See previous slide.
            public static void main(String[] args)
            Charge[] a = readCharges();
            int SIZE = 800;
            Picture pic = new Picture(SIZE, SIZE);
            for (int col = 0; col < SIZE ; col++)
                for (int row = 0; row < SIZE ; row++)
                    double V = 0.0;
                    for (int k = 0; k < a.length; k++)
                        double x = 1.0 * col / SIZE;
                        double y = 1.0 * row / SIZE;
                        V += a[k].potentialAt(x, y);
                    pic.set(col, SIZE-1-row, toColor(V));
            pic.show();
    }
    public static Charge[] readCharges() {
        % more charges3.txt
        3
        .51 .63 -100
        .50 .50 40
        .50 .72 20
        % java Potential < charges3.txt
        }
    public static Color toColor() {
        double V = 0.0;
        if (V < 255)
            V = (V + 255) % 256;
        else if (V >= 0)
            V = (int) V;
        t = t+17 % 251
        return new Color(t, t, t);
    }
    public static void main(String[] args)
        Charge[] a = readCharges();
        int SIZE = 800;
        Picture pic = new Picture(SIZE, SIZE);
        for (int col = 0; col < SIZE ; col++)
            for (int row = 0; row < SIZE ; row++)
                double V = 0.0;
                for (int k = 0; k < a.length; k++)
                    double x = 1.0 * col / SIZE;
                    double y = 1.0 * row / SIZE;
                    V += a[k].potentialAt(x, y);
                pic.set(col, SIZE-1-row, toColor(V));
        pic.show();
    }
}

Potential visualization I:
% more charges9.txt
9
.51 .63 -100
.50 .50 40
.50 .72 20
.33 .33 5
.20 .20 -10
.70 .70 10
.82 .72 20
.85 .23 10
.90 .12 -10
% java Potential < charges9.txt

Potential visualization II: A moving charge:
% more charges8.txt
9
.51 .63 -100
.50 .50 40
.50 .72 20
.33 .33 5
.20 .20 -10
.70 .70 10
.82 .72 20
.85 .23 10
.90 .12 -50
% java PotentialWithMovingCharge < charges8.txt

Potential visualization III: Discontinuous color map:
public static Color toColor(double V) {
    V = 128 + V / 2.0d10;
    int t = 0;
    if (V > 255) t = 255;
    else if (V >= 0) t = (int) V;
    t = t+17 % 251
    return new Color(t, t, t);
}
9. Creating Data Types

- Overview
- Point charges
- Turtle graphics
- Complex numbers

ADT for turtle graphics

A turtle is an idealized model of a plotting device.

An ADT allows us to write Java programs that manipulate turtles.

<table>
<thead>
<tr>
<th>Values</th>
<th>position (x, y)</th>
<th>orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(5, 5)</td>
<td>90°</td>
</tr>
<tr>
<td></td>
<td>(25, 75)</td>
<td>135°</td>
</tr>
<tr>
<td></td>
<td>(22, 12)</td>
<td>10°</td>
</tr>
</tbody>
</table>

public class Turtle

Turtle(double x0, double y0, double q0)
void turnLeft(double delta) // rotate delta degrees counterclockwise
void goForward(double step) // move distance step, drawing a line

Turtle graphics implementation: Test client

Best practice. Begin by implementing a simple test client.

```java
public static void main(String[] args) {
    Turtle turtle = new Turtle(0.0, 0.0, 0.0);
    turtle.goForward(1.0);
    turtle.turnLeft(120.0);
    turtle.goForward(1.0);
    turtle.turnLeft(120.0);
    turtle.goForward(1.0);
    turtle.turnLeft(120.0);
}
```

Note: Client drew triangle without computing \( \sqrt{3} \)

What we expect, once the implementation is done.
Turtle implementation: Instance variables and constructor

**Instance variables** define data-type values.

**Constructors** create and initialize new objects.

```java
class Turtle {
    private double x, y;
    private double angle;

    public Turtle(double x0, double y0, double a0) {
        x = x0;
        y = y0;
        angle = a0;
    }
}
```

Turtle implementation: Methods

**Methods** define data-type operations (implement APIs).

```java
class Turtle {
    public void turnLeft(double delta) {
        angle += delta;
    }
    public void goForward(double d) {
        double oldx = x;
        double oldy = y;
        x = x + d * Math.cos(Math.toRadians(angle));
        y = y + d * Math.sin(Math.toRadians(angle));
        StdDraw.line(oldx, oldy, x, y);
    }
    public static void main(String[] args) {
        Turtle turtle = new Turtle(0.5, 0, angle/2.0);
        for (int i = 0; i < N; i++) {
            turtle.goForward(step);
            turtle.turnLeft(angle);
        }
    }
}
```

Turtle client: N-gon

```java
class Ngon {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        double angle = 360.0 / N;
        double step = Math.sin(Math.toRadians(angle/2.0));
        Turtle turtle = new Turtle(0.5, 0, angle/2.0);
        for (int i = 0; i < N; i++) {
            turtle.goForward(step);
            turtle.turnLeft(angle);
        }
    }
}
```
public class Spiral
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        double decay = Double.parseDouble(args[1]);
        double angle = 360.0 / N;
        double step = Math.sin(Math.toRadians(angle/2.0));
        Turtle turtle = new Turtle(0.5, 0, angle/2.0);
        for (int i = 0; i < N; i++)
        {
            turtle.forward(step);
            turtle.turnLeft(angle);
        }
    }
}
Crash course in complex numbers

A complex number is a number of the form \( a + bi \) where \( a \) and \( b \) are real and \( i \equiv \sqrt{-1} \).

Complex numbers are a quintessential mathematical abstraction that have been used for centuries to give insight into real-world problems not easily addressed otherwise.

To perform algebraic operations on complex numbers, use real algebra, replace \( i^2 \) by \(-1\) and collect terms.

- Addition example: \((3 + 4i) + (-2 + 3i) = 1 + 7i\).
- Multiplication example: \((3 + 4i) \times (-2 + 3i) = -18 + i\).

The magnitude or absolute value of a complex number \( a + bi \) is \( |a + bi| = \sqrt{a^2 + b^2} \).

Applications: Signal processing, control theory, quantum mechanics, analysis of algorithms...

ADT for complex numbers

A complex number is a number of the form \( a + bi \) where \( a \) and \( b \) are real and \( i \equiv \sqrt{-1} \).

An ADT allows us to write Java programs that manipulate complex numbers.

<table>
<thead>
<tr>
<th>Values</th>
<th>complex number</th>
<th>3 + 4i</th>
<th>(-2 + 2i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>real part</td>
<td>3.0</td>
<td>-2.0</td>
<td></td>
</tr>
<tr>
<td>imaginary part</td>
<td>4.0</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>public class Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex(double real, double imag)</td>
</tr>
<tr>
<td>Complex plus(Complex b)</td>
</tr>
<tr>
<td>Complex times(Complex b)</td>
</tr>
<tr>
<td>double abs()</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>
Complex number data type implementation: Test client

Best practice. Begin by implementing a simple test client.

```java
public static void main(String[] args)
{
    Complex a = new Complex(3.0, 4.0);
    Complex b = new Complex(-2.0, 3.0);
    StdOut.println("a = " + a);
    StdOut.println("b = " + b);
    StdOut.println("a * b = " + a.times(b));
}
```

What we expect, once the implementation is done.

complex number data type implementation: Instance variables and constructor

Instance variables define data-type values.

Constructors create and initialize new objects.

```java
public class Complex
{
    private final double real;
    private final double imag;
    public Complex(double real, double imag)
    {
        this.real = real;
        this.imag = imag;
    }
    public Complex(Complex b)
    {
        double real = re + b.re;
        double imag = im + b.im;
        return new Complex(real, imag);
    }
    public double abs()
    { return Math.sqrt(re*re + im*im); }
    public String toString()
    { return re + " + " + im + "i"; }
}
```

Methods define data-type operations (implement APTs).

```java
public class Complex
{
    ... public Complex plus(Complex b)
    { double real = re + b.re;
        double imag = im + b.im;
        return new Complex(real, imag);
    }
    ... public Complex times(Complex b)
    { double real = re * b.re - im * b.im;
        double imag = re * b.im + im * b.re;
        return new Complex(real, imag);
    }
    ... public double abs()
    { return Math.sqrt(re*re + im*im); }
    ... public String toString()
    { return re + " + " + im + "i"; }
}
```

API

Complex operations.

```java
Complex(double real, double imag)
Complex plus(Complex b) sum of this number and b
Complex times(Complex b) product of this number and b
double abs() magnitude
String toString() string representation
```
The Mandelbrot set

The Mandelbrot set is a set of complex numbers. It is defined as follows:

- Represent each complex number \( x + yi \) by a point \((x, y)\) in the plane.
- If a point is in the set, we color it BLACK.
- If a point is not in the set, we color it WHITE.

**Examples**

- In the set: \(-0.5 + 0i\).
- Not in the set: \(1 + i\).

**Challenge**

- No simple formula exists for testing whether a number is in the set.
- Instead, the set is defined by an algorithm.

Determining whether a point is in the Mandelbrot set

Is a complex number \(z_0\) in the set?

- Iterate \(z_t = (z_t)^2 + z_0\).
- If \(|z_t|\) diverges to infinity, \(z_0\) is not in the set.
- If not, \(z_0\) is in the set.

<table>
<thead>
<tr>
<th>(t)</th>
<th>(z_t)</th>
<th>(t)</th>
<th>(z_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(-1/2 + 0i)</td>
<td>0</td>
<td>(1 + i)</td>
</tr>
<tr>
<td>1</td>
<td>(-1/4 + 0i)</td>
<td>1</td>
<td>(1 + 3i)</td>
</tr>
<tr>
<td>2</td>
<td>(-7/16 + 0i)</td>
<td>2</td>
<td>(-7 + 1i)</td>
</tr>
<tr>
<td>3</td>
<td>(-79/256 + 0i)</td>
<td>3</td>
<td>(1 - 97i)</td>
</tr>
<tr>
<td>4</td>
<td>(-65537/65536 + 0i)</td>
<td>4</td>
<td>(-9407 - 193i)</td>
</tr>
</tbody>
</table>

always between \(-1/2\) and \(0\)
\(z = -1/2 + 0i\) is in the set
diverges to infinity
\(z = 1 + i\) is not in the set

Complex number client: Mandelbrot set visualization (helper method)

Mandelbrot function of a complex number.

- Returns WHITE if the number is not in the set.
- Returns BLACK if the number is (probably) in the set.

```java
public static Color mand(Complex z0) {
    Complex z = z0;
    for (int t = 0; t < 255; t++) {
        if (z.abs() > 2.0) return Color.WHITE;
        z = z.times(z);
        z = z.plus(z0);
    }
    return Color.BLACK;
}
```

For a more dramatic picture, return new Color(255-t, 255-t, 255-t) or colors picked from a color table.

Plotting the Mandelbrot set

**Practical issues**

- Cannot plot infinitely many points.
- Cannot iterate infinitely many times.

**Approximate solution for first issue**

- Sample from an \(N\)-by-\(N\) grid of points in the plane.
- Zoom in to see more detail (stay tuned!).

**Approximate solution for second issue**

- Fact: if \(|z_t| > 2\) for any \(t\), then \(z\) is not in the set.
- Pseudo-fact: if \(|z_{255}| < 2\) then \(z\) is "likely" in the set.

**Important note:** Solutions imply significant computation.
Complex number client: Mandelbrot set visualization

```java
import java.awt.Color;
public class Mandelbrot {
    public static Color mand(Complex z0) {
        // See previous slide...
        public static void main(String[] args) {
            double xc = Double.parseDouble(args[0]);
            double yc = Double.parseDouble(args[1]);
            double size = Double.parseDouble(args[2]);
            int N = Integer.parseInt(args[3]);
            Picture pic = new Picture(N, N);
            for (int col = 0; col < N; col++)
                for (int row = 0; row < N; row++)
                    { double x = xc - size/2 + size*col/N;
                      double y = yc - size/2 + size*row/N;
                      Complex z = new Complex(x, y);
                      Color color = mand(z);
                      pic.set(col, N-1-row, color);
                    }
            pic.show();
        }
    }
}
```

Mandelbrot Set

```java
% java Mandelbrot -.5 0 2
```

Mandelbrot Set

```java
% java GrayscaleMandelbrot -.5 0 2
```

Mandelbrot Set

```java
% java GrayscaleMandelbrot .1045 -.637 .01
```
OOP summary

Object-oriented programming (OOP)
- Create your own data types (sets of values and ops on them).
- Use them in your programs (manipulate objects).

OOP helps us simulate the physical world
- Java objects model real-world objects.
- Not always easy to make model reflect reality.
- Examples: charged particle, color, sound, genome.

OOP helps us extend the Java language
- Java doesn’t have a data type for every possible application.
- Data types enable us to add our own abstractions.
- Examples: complex, vector, polynomial, matrix, picture.

You have come a long way

You can now write any program you might want to write.

Course goal. Open a whole new world of opportunity for you (programming).
9. Creating Data Types