Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden
Christoph Daniel Schulze, Steven Smyth

5. Übung zur Vorlesung „Programmierung“

Fifth Element: Classes
Abgabe am Sonntag, 04. Dezember 2016 - 23:55

In the lecture, you saw how to write and read Javadoc documentation. Here are two useful links you may want to keep handy:

- ACM Library Documentation
- Java 8 Library Documentation (there's *a lot* in the Java library, so don't be overwhelmed by the sheer amount of documentation)

There is one more difference this week: your classes may not be subclasses of `ConsoleProgram` or `GraphicsProgram` anymore. That's because some of them are not supposed to be complete, runnable programs. But if they are not, how will you test them? Well, simply create a second class that *is* a subclass of `ConsoleProgram` and add code to your `run()` method to test your main class.

As usual, always comment your code!

Aufgabe 1 - A (Mandatory) Class for Complex Numbers — 49 Punkte

In this assignment, you will

- practice further to write your own classes and methods,
- learn how to write proper javadoc comments, and
- you will familiarize yourself with complex numbers.

Complex numbers are an important mathematical concept that extends the real numbers by bringing the imaginary unit i into the picture. Your task is to write a class `ComplexNumber` that represents a complex number and offers methods to meddle with it.

Mathematically, a complex number has the form $c = a + bi$, where a and b are real numbers (elements of $\mathbb{R}$) and i satisfies the equation $i^2 = -1$. That's right, i can be thought of as the root of -1, which does not exist in $\mathbb{R}$. a is referred to as the *real part* of c and b is the *imaginary part* of c. The set of complex numbers is denoted $\mathbb{C}$ and is a strict superset of $\mathbb{R}$; you can think of a real number as a complex number where the imaginary part bi is 0. For a further discussion of complex numbers and how the mathematical operations on them are defined see the Wikipedia entry on complex numbers; those of you who participate in *Mathematik A* may also have a look at the chapter on complex numbers in the skript of Prof. Berghammer.

This is what your class `ComplexNumber` should provide:

- Two private fields called `re` and `im` that represent the real and the imaginary part of your complex number.
- Three public constructors. `ComplexNumber()` initializes the real and imaginary parts with 0. `ComplexNumber(double real, double imaginary)` initializes the real and imaginary part with the given values. `ComplexNumber(ComplexNumber cn)` initializes the real and imaginary parts with the values from the given complex number.

- Getters to access the real and imaginary part of your complex number, called `getReal()` and `getImaginary()`. Your class should *not* provide setters. Once a complex number is instantiated, it cannot be changed anymore.
- Methods to calculate stuff, all of which return a new complex number instead of changing the current one.
  - `conjugate()` returns a new complex number which is the conjugate of the one the methods was called on.
  - `add(ComplexNumber other)` returns a new complex number which is the sum of the two.
  - `subtract(ComplexNumber other)` returns a new complex number which is the result of subtracting `other` from the current complex number.
  - `multiply(ComplexNumber other)` returns a new complex number which is the product of the two.
  - `divide(ComplexNumber other)` returns `null` if the real and imaginary part of `other` is 0. Otherwise, it returns a new complex number which is the result of dividing the current complex number by the other one.
  - `abs()` returns a `double` that is the absolute value of the complex number. (Note: The `Math` class provides a method to calculate square roots.)
- A `toString()` method that returns a sensible string representation of the complex number. A complex number with real part `42` and imaginary part `23` could for instance be written as `42 + 23i`.

Remember to **add complete Javadoc comments** to your class as well as all public constructors and methods (adding them to your private fields won't hurt either). Be sure to document parameters and return values as well.

Aufgabe 2 - (Mandatory) Powerful Classes — 49 Punkte

In the lecture you have seen how class hierarchies are built and in Problem Set 2 you created your own hierarchy for vehicles. In this assignment, you should practice how to

- read the specifications of the different vehicles carefully,
- design a reasonable class hierarchy, and
- implement your class hierarchy.

The program you develop should be able to calculate the theoretical maximum velocity of your vehicles with regard to the machine power. However, the calculation of the velocity depends on the method of transportation. Of course, you don't want to duplicate code and work, so think of a reasonable hierarchy.

(The calculations presented here are simplifications. Realistic models may be more complex. You can think about the simplifications of each model for fun. What might be missing?)

Cars The theoretical maximum velocity of cars depends on the power P of the engine (in W), the size A of the front surface of the car, the density of the air ρ, and the drag coefficient c_w. All cars we know (here in the wonderous world of iLearn) have a front surface $A = 2.5m^2$ and a drag coefficient $c_w = 0.35$. The air density is assumed to be $\rho = 1.3\frac{kg}{m^3}$.

The theoretical maximum velocity can then be calculated with $v = \sqrt[3]{\frac{2P}{\rho \cdot A \cdot c_w}}$.

Nevertheless, car manufactureres often use metric Horsepower (PS) and so do we… at least for cars.

Steam Ships The drag coefficient can also be used to calculate the velocity of ships. Contrary to the car model, the air resistance is less important for ships. Instead, the water resistance is the crucial factor. We can use the same formula of the cars model, but since a ship's "front surface" is a bit different from a car's, we need to talk about how to calculate that first.

A ship has a certain displacement tonnage (Wasserverdrängung) which determines the amount of water volume V it displaces. If we imagine the ship's solid body as a prism, we can calculate the front surface $A = \frac{V}{l}$ with l being the length of the ship. Assume $c_w = 0.3$ and $\rho = 1.028\frac{t}{m^3}$.

Ships often use knots instead of km/h for speed measurements: 1 knot = 1 nautic mile/h = 1,85 km/h

Rowing Boats (Ruderboot) The fact that we can use the aforementioned formula to calculate the speed of an old Greek Trireme as well is maybe more interesting. Here, the model of a prism for the ship's body is maybe not justified. We use an isosceles triangle (gleichschenkliges Dreieck) to model the front of the boat and, hence, calculate $A = \frac{1}{2} \cdot b \cdot h$ with b being the width of the boat and h the draught (Tiefgang). Use the c_w and ρ values of the steam ship's model.

Usually, the power a rower (Ruderer) can produce is not measured in PS. A typical value for a sustainable amount of mechanical power a human can produce is $100W$. Thus, the maximum power of a rowing boat is $P = n \cdot 100W$ with n set to the number of rowers (without looking at rower synchronization issues or people being plain lazy).

Bicycles Everything above gets more complicated when using the bike. (It really is… you can check the links below if you are interested.) So, to keep things simple enough, we only consider flat routes with no supporting wind. Then, the maximum velocity is restricted by the air resistance and the surface friction.

You can calculate the maximum velocity with $v = \sqrt[3]{a + \sqrt{a^2 + b^3}} + \sqrt[3]{a - \sqrt{a^2 + b^3}} - \frac{2}{3} \cdot \frac{0.1}{c_d \cdot A \cdot \rho}$ in $\frac{m}{s}$.

Assuming an average person ($1.72m$. $71.3kg$) and a sustainable power output of $P = 160W$, depending on the bike you are riding, a, b, and the effective front head surface $c_d \cdot A$ varies:

- Hands on the tops racing bike (Oberlenker Rennrad): $a = 277.376$, $b = 3.078$, $c_d \cdot A = 0.4891m^2$
- Hands on the drops racing bike (Unterlenker Rennrad): $a = 399.611$, $b = 4.4226$, $c_d \cdot A = 0.3397m^2$
- Roadster (Hollandrad): $a = 181.0455$, $b = 3.3899$, $c_d \cdot A = 0.7457m^2$

Set $\rho = 1.2\frac{kg}{m^3}$ for all bikes.

For the sake of completeness: you can use Cardano's method to obtain a and b when solving $P = c_m \cdot v \cdot (c_d \cdot A \cdot \frac{\rho}{2} \cdot (v + w)^2 + F_{rg} + v \cdot c_{rvh})$ which describes the correlation between power and velocity here. For more details feel free to consult the links below.

Write your main class Eventually, write a class `MaximumVelocity` which extends `ConsoleProgram` that prints out the following template with correct and **truncated integer** velocity values ("41km/h" instead of "41.3km/h", "56km/h" instead of "56.8km/h")

```
VW Polo (45 PS): XXXkm/h
Porsche 911 (218 PS): XXXkm/h
Lamborghini Countach (454 PS): XXXkm/h
HMS Titanic (51,000 PS, 45,000m³, 269m): XXXkm/h (YYYkts)
USS Nimitz (280,000 PS, 80,000m³, 332m): XXXkm/h (YYYkts)
Greek Trireme (170 rowers, b = 6.1m, h = 0.9m ): XXXkm/h (YYYkts)
Bicycle (Hands on the tops): XXXkm/h
Bicycle (Hands on the drops): XXXkm/h
Bicycle (Roadster): XXXkm/h
```

Besides the usual comments, **explain your class hierarchy in the comments of your source code**.

Submitting Your Solution **Important:** You can simply put all your classes in one Java file and paste the code as usual here in iLearn. However, since you can only have one public class in your Java file and this has to be the class `MaximumVelocity`, make sure that your other classes do not have a visibility modifier (public or private).

If you want to structure your hierarchy in different files, you can also upload an archive file. To turn your project into an archive file that you can upload, perform the following steps in Eclipse:

1. Right-click your project in the *Package Explorer* and click *Export*.
2. From the *Java* category, select *JAR file* and clickt *Next*.
3. Make sure your project's `src` folder is exported. Choose not to export generated class files and resources, but only Java source files and resources.
4. Choose where to save the file and click *Finish*.

Hints

- $1PS = 735,49875W$
- Think about your class hierarchy **before** starting to write code!
- **To get the best possible support in the practical class, think about a meaningful class hierarchy beforehand.**

Acknowledgements:

- Fermi-Probleme zum Strömungswiderstand (R. Müller, Univ. München)
- Wikipedia: Drag coefficient
- Geschwindigkeit & Leistung von www.kreuzotter.de

Aufgabe 3 - 14159265 — 1 Punkt

One reason programmers like drawing circles so much is that there are no corner cases to be wary of. But drawing a simple circle is easy; you've already done that numerous times. Instead, what we want to do in this assignment is to approximate π and, in the process, draw an approximate circle as well. To do this, you are going to use the `RandomGenerator` class from the lecture. This assignment focuses on

- writing your own methods,
- the `RandomGenerator` class, and
- different `GObjects`.

Imagine a circle with radius $r = 1$ centered at the coordinate system's origin, $(0, 0)$. If we start generating random points within the circle's bounding box (the smallest square that contains the whole circle), then all of them will lie in the bounding box, but only a certain percentage of them will also lie in the circle. The percentage is equal to the ratio of their area:

$$\frac{\text{points within the circle}}{\text{points within the bounding box}} = \frac{\text{area of the circle}}{\text{area of the bounding box}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

This means that if we count the number of points that fall into the circle, we can actually approximately calculate π. If we then also visualize our generated points and give those inside the circle a different color than those outside, the result may look something like this (in this case for 50.000 points):

Write a program `PiApproximator` that extends `GraphicsProgram` and draws such pictures and that prints the approximate value of π to the console. Start by writing the following two methods:

```
/**
 * Randomly generates a new point whose x and y coordinates lie between -1
 * and 1.
```

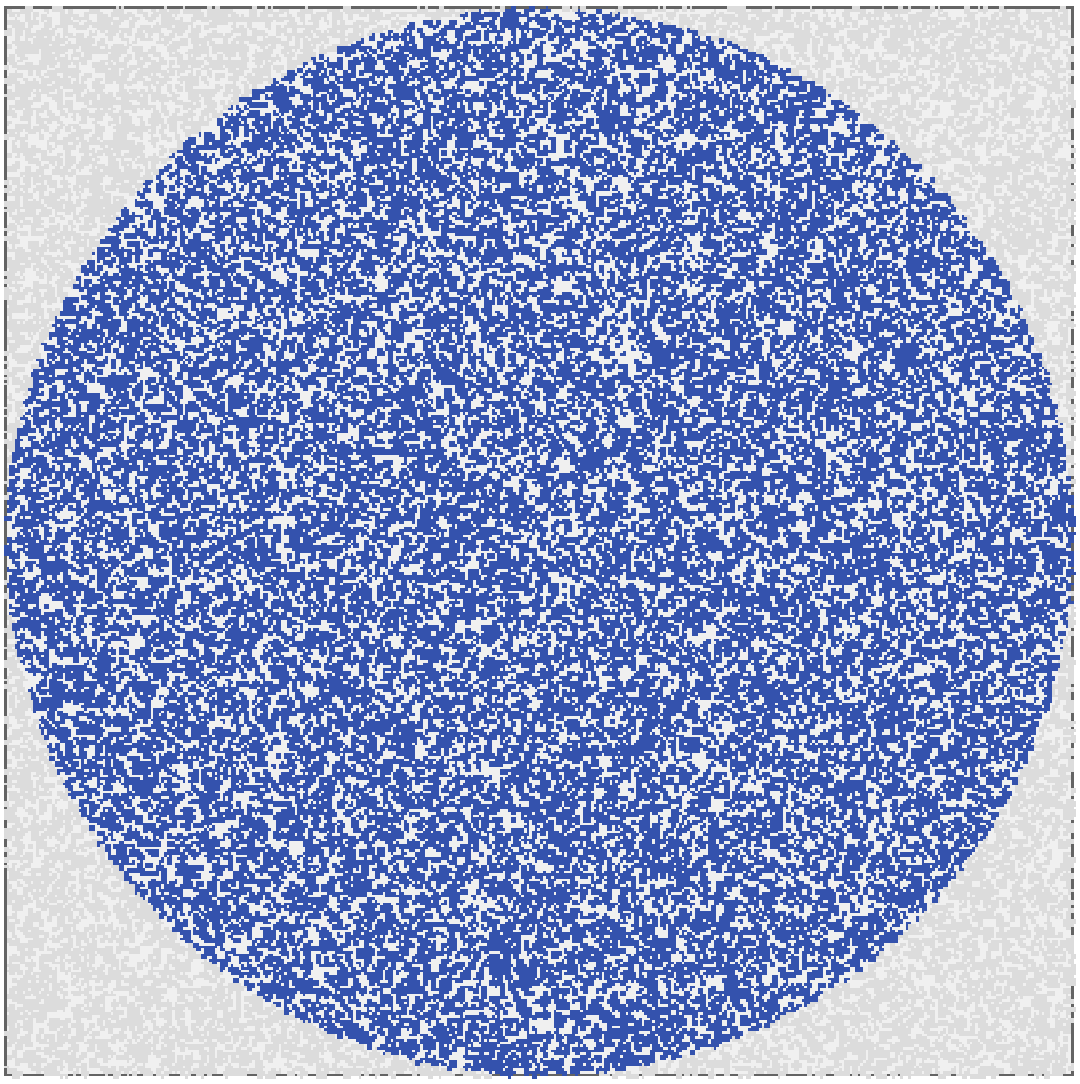
Abbildung 1:

```
 *
 * @return random point.
 */
public GPoint randomPoint() {
    // Implement this method
}

/**
 * Checks if the point with the given coordinates is inside the circle with
 * radius 1 centered at the coordinate system's origin.
 *
 * @param unitPoint
 *            the point to check.
 * @return {@code true} if the point is inside the circle, {@code false}if
 *         it's not.
 */
public boolean isInCircle(GPoint unitPoint) {
    // Implement this method
}
```

Note that to check if a point lies inside our unit circle, we only have to check if the point's distance to the coordinate system's origin is below 1:

$$\sqrt{x^2 + y^2} < 1$$

Since square roots are expensive to compute, you should square each side of the condition:

$$x^2 + y^2 < 1$$

Next, write a `run()` method that does the following:

- Draw a square that visualizes the bounding box of the circle. Of course, the square should not be centered at coordinates $(0, 0)$, since then we would only see a quarter of it. It should also be bigger than two by two, because otherwise we wouldn't be able to see much. Thus, choose a good size (for our example, we have settled for a side length of 400) and place the rectangle such that you can properly see all of it.
- Generate 10.000 points. Visualize each point by adding a `GOval` with width and height set to 1. For each point, check if it falls inside the unit circle or not. Set its oval's colour depending on whether it does or not. Once the program has run, you should be able to clearly distinguish between those points that fall inside the circle and those that don't. Take care to convert the coordinates of the random points such that the resulting circle fills the bounding box.
- Along the way, count how many of the points fall inside the circle. That number divided by the total number of generated points equals $\frac{\pi}{4}$. Calculate π and print it (without anything else) to the console.

If you increase the number of generated points, you will notice how the approximation of π improves. However, for your submission, be sure to generate only 10.000 points.

Zusatzaufgabe 4 - Mandelbrot 1 Punkt

The *Mandelbrot set* is an interesting mathematical construct that has the additional benefit of looking so good that it makes for a great poster in your room. Well, in this assignment you'll be writing code to draw the Mandelbrot set (which, incidentally, will teach you everything there is to know about art). You will draw the set by placing squares of a certain size next to each other.

Write a program called `Mandelbrot` that extends `GraphicsProgram` and asks the user for three things:

1. The length of the edges of the squares.
2. The width of the drawing in terms of the number of squares.
3. The height of the drawing in terms of the number of squares.

This image was generated using 50 times 30 squares with a side length of 10:

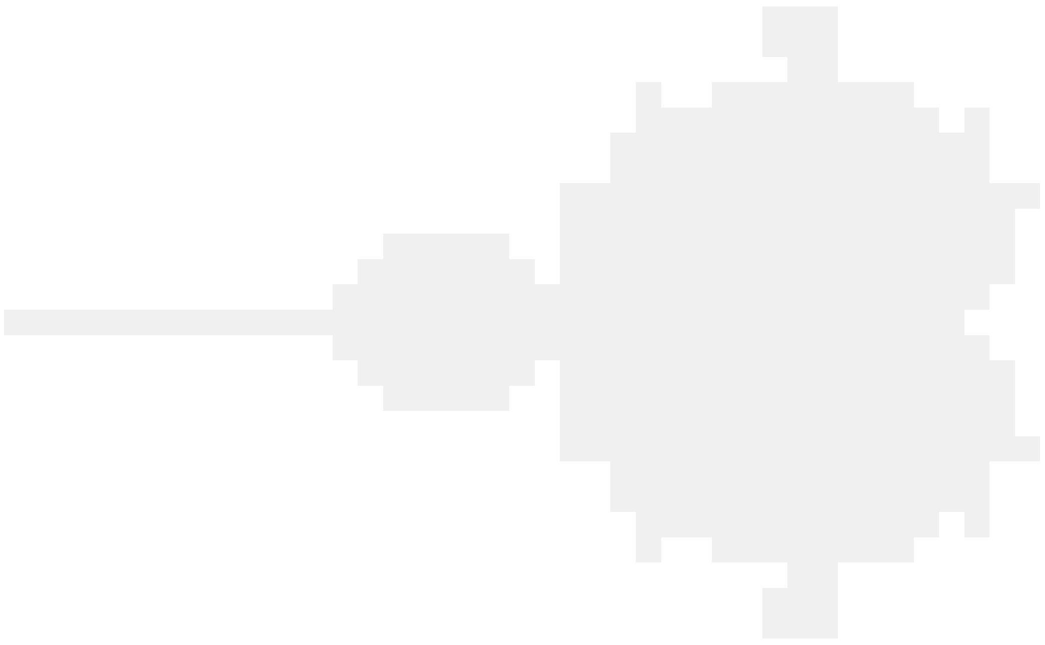

Abbildung 2:

This image, in turn, was generated using 250 times 150 squares with a side length of 2:

Now that you know what the results should look like, here's how to compute them. The Mandelbrot set is the set of all complex numbers $c \in \mathbb{C}$ for which the sequence $(z_{c,n})_{n \in \mathbb{N}}$ with $z_{c,0} = 0$ and $z_{c,n+1} = z_{c,n}^2 + c$ does not approach infinity. Elements of the Mandelbrot set shall be visualized using gray (`new Color(240, 240, 240)`) squares. Elements outside the Mandelbrot set shall be visualized using white (`new Color(255, 255, 255)`) squares.

Now, this may all look a bit daunting, so let's walk you through what your program has to do for each square to decide whether it should be gray or white:

1. Each square represents a specific $c \in \mathbb{C}$ for which we need to decide whether it is in the Mandelbrot set or not. Thus, your program will first have to work out which c the square represents. Looking at a Mandelbrot set in a coordinate system (the Wikipedia article linked to above has a nice picture) gives us that $c_{re} = \frac{3j}{w} - 2$ and $c_{im} = \frac{2i}{h} - 1$ (where i is the row of the square, j is the column of the square, and w and h are the width and height in squares).
2. Next, your program will have to find out whether the sequence $(z_{c,n})_{n \in \mathbb{N}}$ that belongs to c approaches infinity (not in the Mandelbrot set) or not (in the Mandelbrot set). We start with $z_{re} = z_{im} = 0$ and simply compute the next element of the sequence until we have determined that the sequence does not approach infinity. The next element z' is given by $z'_{re} = z_{re}^2 - z_{im}^2 + c_{re}$ and $z'_{im} = 2 z_{re} z_{im} + c_{im}$. We know that the sequence does approach infinity if we find that for our current element z, $z_{re}^2 + z_{im}^2 > 4$, at which point we can stop computing more elements. However, finding such an element may require us to compute a whole lot of elements, which takes time and may actually never happen. We thus do the

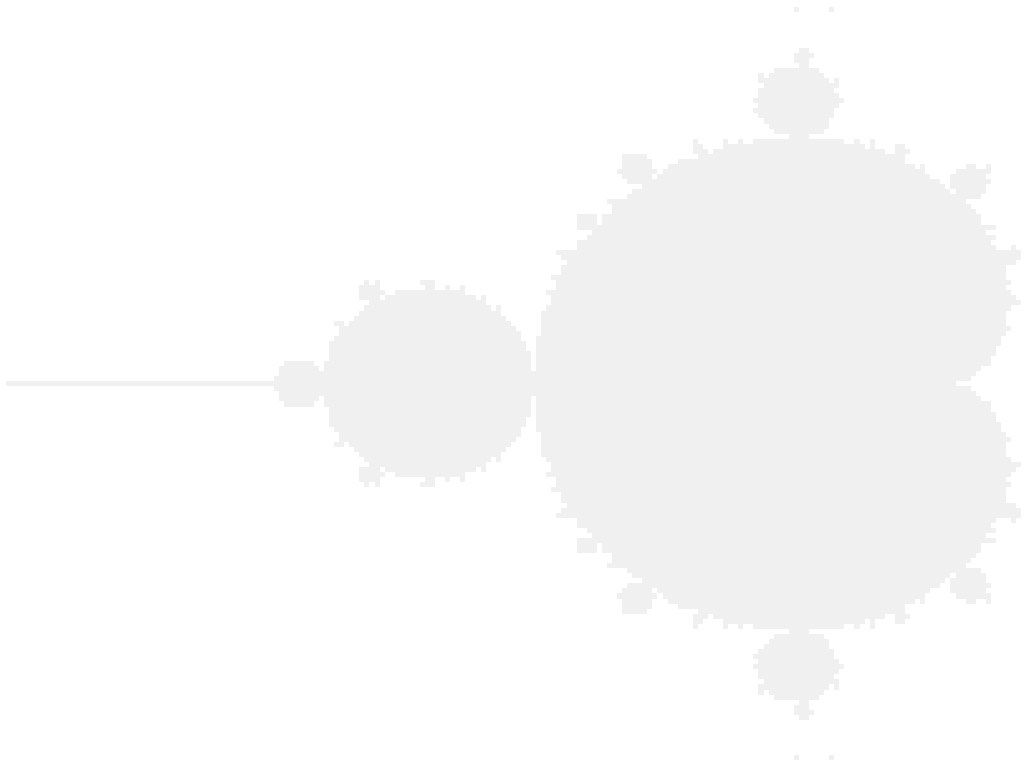

Abbildung 3:

following: if after 400 iterations we haven't found one such element, we assume that the sequence will not approach infinity and c is thus part of the Mandelbrot set.

3. Compute the x and y coordinate where the square needs to be placed and set its color depending on whether it's part of the Mandelbrot set or not.

Hints:

- Use your `ComplexNumber` class from Assignment 2 to handle complex numbers. Of course, you may extend your class if necessary.