Write your name on each sheet that contains your answers! Clearly cross out any notes that are not part of your answer.

Your name: Joshua Lagrange

# Exam — H02C5A Object Oriented Programming

Prof. Bart Jacobs
10 June 2020, 08:00-11:00, room ON1.04.28

**Question 1: Fill in the blanks.**

A decomposition of a software system is *modular* if each module can be developed, understood, verified, and evolved independently from and in parallel with the other modules.

The main approach for managing the complexity of developing software systems is by achieving modularity through abstraction. This means the system is split into a client module that implements the system's functionality in a programming language extended with additional operations (this approach is called procedural abstraction) or additional datatypes (this approach is called data abstraction), and a module that implements the API using the constructs of the base programming language.

This approach works best if the API is documented sufficiently precisely and abstractly.

class implemenation can further be categorized as immutable if an object's class properties is fixed at construction time, or mutable if it can change during the object's lifetime.

Formal class documentation includes public invariants which define the valid abstract values of an instance and private field which define the valid state of an instance. Constructors and methods are documented mainly using invar which specify results and side-effects, and

pre (in case of contractual programming) or throws (in case of defensive programming).

Multi-object abstractions typically involve bidirectional associations, whose consistency must be preserved at all times. This means that if according to an object o1's representation, o1 is associated with an object o2, then

O2 is also associated with object o1

.

inheritance means that classes can be declared as subclass of other classes. In that case, instances of the subclass are also considered to be instances of the superclass. A class that only serves as a generalization of other classes and that is not intended to be instantiated directly is called a/an abstract class.

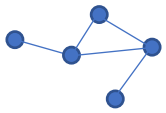
A polymorphic variable is one that can refer to objects of different classes. Java's static type checker allows a field access or a method call only if the target expression's class declares or inherits the field or method. Programmers can work around this by using typecast(ing). These check an object's class at runtime; if the check fails, it is reported as a/an ClassCastException.

If a class declares a method whose name and number of types of parameters match a method of the superclass, then this method is said to override the other one. A method that only serves to be overriden can be declared abstract; for such methods, you do not need to provide a/an body.

There are two kinds of method binding: in case of dynamic binding, the method to be executed is determined at runtime based on the class of the target expression of the call; in case of static binding, the method to be executed is determined at compiletime based on the class of the target expression of the call.

**Question 2: Programming Task: Networks**

We want to extend DrawIt with the ability to draw *networks*. A network (pictured to the right) consists of a number of *nodes* and a number of *links* connecting two nodes. We say two nodes are *neighbors* if they are connected by a link. There is at most one link between any two nodes.



Q2.1. Develop a class (= write down the Java code for a class on blank paper) such that each instance represents a node in a network. Allow clients to retrieve the set of neighbors of a node, to link a node to another node, and to remove the link between two given nodes.

Note: a node's set of neighbors is the only property you need to store. Of course, to be able to draw a network, each node's position in the drawing is needed; however, this is outside the scope of this exam.

Make sure your abstraction is properly encapsulated. Include full formal public and internal documentation. (You need not write any informal documentation.) Deal with illegal cases of adding a link defensively; deal with illegal cases of removing a link contractually.

In formal documentation, you can express the set $s \cup \{e\}$ as LogicalSet.plus(s, e) and the set $s \setminus \{e\}$ as LogicalSet.minus(s, e).

Q2.2. Write a test suite for testing your abstraction. You need not test illegal cases. Make sure every statement of your abstraction is tested, except for statements that run only in illegal cases.

It is generally recommended to split a test suite into multiple test methods. However, for simplicity, we here ask that you fully test your abstraction using a single test method. It is sufficient to write down the body of your single test method.

**Question 3: Programming Task: Network Node Appearances**

We want DrawIt users to be able to assign different appearances to different nodes of a network. We want DrawIt to support square node appearances and circular node appearances. Each node appearance specifies a color for the node (represented as a java.awt.Color object). A square node appearance is fully defined by the color and the width of the square (an int). A circular node appearance is fully defined by the color and the radius of the circle (an int).

Q3.1. Develop a class hierarchy for representing node appearances. Node appearance objects shall be immutable. You need not write any documentation. You need not write code for dealing with illegal cases. Make it so that the Java Collections API considers two node appearance objects o1 and o2 equal (for example, List.of(o1).contains(o2) should return true) if and only if they represent the same node appearance. When writing code for this functionality, implement common functionality (i.e. checking the color) once and reuse it.

Q3.2. Write a small test suite to test your class hierarchy. You need not test illegal cases. Write it in the form of a single test method. It is sufficient to write down the body of your single test method. Make sure every statement of your class hierarchy is tested.

```java
package drawit;

import java.util.Objects;

/**
 * An immutable abstraction for a point in the two-dimensional plane with {@code int} coordinates.
 *
 * @immutable
 */
public class IntPoint {

	private final int x;
	private final int y;

	/** Returns this point's X coordinate. */
	public int getX() { return x; }
	/** Returns this point's Y coordinate. */
	public int getY() { return y; }

	/**
	 * Returns {@code true} if this point has the same coordinates as the given point; returns {@code false} otherwise.
	 *
	 * @pre | other != null
	 * @post | result == (this.getX() == other.getX() && this.getY() == other.getY())
	 */
	public boolean equals(IntPoint other) {
		return other.x == x && other.y == y;
	}

	/**
	 * Returns {@code true} if the given object is an {@code IntPoint} object and
	 * this point has the same coordinates as the given object; returns {@code false} otherwise.
	 *
	 * @post | result == (other instanceof IntPoint && this.equals((IntPoint)other))
	 */
	@Override
	public boolean equals(Object other) {
		return other instanceof IntPoint && this.equals((IntPoint)other);
	}

	/**
	 * Returns a number that depends only on this object's coordinates.
	 *
	 * @post
	 *     | result == 31 * (31 + getX()) + getY()
	 */
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + x;
		result = prime * result + y;
		return result;
	}

	/**
	 * Returns a textual representation of this object.
	 *
	 * @post | Objects.equals(result, "IntPoint [x=" + getX() + ", y=" + getY() + "]")
	 */
	@Override
	public String toString() {
		return "IntPoint [x=" + x + ", y=" + y + "]";
	}

	/** Initializes this point with the given coordinates.
	 *
	 * @mutates | this
	 * @post | getX() == x
	 * @post | getY() == y
	 */
	public IntPoint(int x, int y) {
		this.x = x;
		this.y = y;
	}

	/** Returns an {@code IntVector} object representing the displacement from {@code other} to {@code this}.
	 *
	 * @pre | other != null
	 * @post | result != null
	 * @post | result.getX() == this.getX() - other.getX()
	 * @post | result.getY() == this.getY() - other.getY()
	 */
	public IntVector minus(IntPoint other) {
		return new IntVector(x - other.x, y - other.y);
	}

	/**
	 * Returns true iff this point is on open line segment {@code bc}.
	 * An open line segment does not include its endpoints.
	 *
	 * <p><b>Implementation hints:</b> Call this point {@code a}. First check if {@code ba} is collinear with {@code bc}. If not, return {@code false}.
	 * Then check that the dot product of {@code ba} and {@code bc} is between zero and the dot product of {@code bc} and {@code bc}.
	 *
	 * @pre | b != null
	 * @pre | 0 <= b.getX() && b.getX() <= 10000 && 0 <= b.getY() && b.getY() <= 10000
	 * @pre | c != null
	 * @pre | 0 <= c.getX() && c.getX() <= 10000 && 0 <= c.getY() && c.getY() <= 10000
	 * @post
	 *     | result == (
	 *     |   this.minus(b).isCollinearWith(c.minus(b)) &&
	 *     |   0 < this.minus(b).dotProduct(c.minus(b)) &&
	 *     |   this.minus(b).dotProduct(c.minus(b)) < c.minus(b).dotProduct(c.minus(b))
	 *     | )
	 */
	public boolean isOnLineSegment(IntPoint b, IntPoint c) {
		IntPoint a = this;
		// Is it on the carrier?
		IntVector bc = c.minus(b);
		IntVector ba = a.minus(b);
		if (!ba.isCollinearWith(bc))
			return false;
		long dotProduct = ba.dotProduct(bc);
		return 0 < dotProduct && dotProduct < bc.dotProduct(bc);
	}

	/**
	 * Returns a {@code DoublePoint} object that represents the same 2D point represented by this {@code IntPoint} object.
	 *
	 * @post | result != null
	 * @post | result.getX() == this.getX()
	 * @post | result.getY() == this.getY()
	 */
	public DoublePoint asDoublePoint() {
		return new DoublePoint(this.x, this.y);
	}

	/** Returns an {@code IntPoint} object representing the point obtained by displacing this point by the given vector.
	 *
	 * @pre | vector != null
	 * @post | result != null
	 * @post | result.getX() == this.getX() + vector.getX()
	 * @post | result.getY() == this.getY() + vector.getY()
	 */
	public IntPoint plus(IntVector vector) {
		return new IntPoint(this.x + vector.getX(), this.y + vector.getY());
	}

	/**
	 * Returns true iff the open line segment {@code ab} intersects the open line segment {@code cd}.
	 *
	 * <p><b>Implementation Hints:</b> Assume the precondition holds. Then {@code ab} intersects {@code cd} if and only if {@code ab} straddles the carrier of {@code cd} and
	 * {@code cd} straddles the carrier of {@code ab}. Two points straddle a line if they are on opposite sides of the line.
	 *
	 * <p>Specifically, {@code cd} straddles the carrier of {@code ab} iff (the signum of the cross product of {@code ac} and {@code ab}) times
	 * (the signum of the cross product of {@code ad} and {@code ab}) is negative.
	 *
	 * <p>The signum of a number {@code x} is -1 if {@code x} is negative, 0 if {@code x} is zero, and {@code 1} otherwise. See {@link Math#signum(double)}.
	 *
	 * @pre | a != null
	 * @pre | b != null
	 * @pre | c != null
	 * @pre | d != null
	 * @pre The line segments have at most one point in common.
	 */
	public static boolean lineSegmentsIntersect(IntPoint a, IntPoint b, IntPoint c, IntPoint d) {
		// Check if cd straddles the carrier of ab and ab straddles the carrier of cd
		IntVector ab = b.minus(a);
		if (Math.signum(c.minus(a).crossProduct(ab)) * Math.signum(d.minus(a).crossProduct(ab)) < 0) {
			// cd straddles the carrier of ab

			IntVector cd = d.minus(c);
			return Math.signum(a.minus(c).crossProduct(cd)) * Math.signum(b.minus(c).crossProduct(cd)) < 0;

		} else
			return false;
	}

}
```

```java
package drawit;

import java.util.Objects;

/**
 * An instance of this class represents a displacement in the two-dimensional plane.
 *
 * @immutable
 */
public class IntVector {

	private final int x;
	private final int y;

	public int getX() { return x; }
	public int getY() { return y; }

	/**
	 * @pre | other != null
	 * @post | result == (getX() == other.getX() && getY() == other.getY())
	 */
	public boolean equals(IntVector other) {
		return x == other.x && y == other.y;
	}

	/**
	 * Returns a number that depends only on this object's coordinates.
	 *
	 * @post | result == 31 * (31 + getX()) + getY()
	 */
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + x;
		result = prime * result + y;
		return result;
	}

	/**
	 * Returns a textual representation of this object.
	 *
	 * @post | Objects.equals(result, "IntVector [x=" + getX() + ", y=" + getY() + "]")
	 */
	@Override
	public String toString() {
		return "IntVector [x=" + x + ", y=" + y + "]";
	}

	/**
	 * Returns {@code true} if the given object's class is {@code IntPoint} and
	 * this object represents the same displacement as the given object.
	 *
	 * @post | result == (object != null && object.getClass() == this.getClass() && this.equals((IntVector)object))
	 */
	@Override
	public boolean equals(Object object) {
		if (this == object)
			return true;
		if (object == null)
			return false;
		if (getClass() != object.getClass())
			return false;
		IntVector other = (IntVector) object;
		if (x != other.x)
			return false;
		if (y != other.y)
			return false;
		return true;
	}

	/**
	 * @mutates | this
	 * @post | getX() == x
	 * @post | getY() == y
	 */
	public IntVector(int x, int y) {
		this.x = x;
		this.y = y;
	}

	/**
	 * Returns the cross product of this vector and the given vector.
	 *
	 * @pre | other != null
	 * @post | result == (long)getX() * other.getY() - (long)getY() * other.getX()
	 */
	public long crossProduct(IntVector other) {
		return (long)x * other.y - (long)y * other.x;
	}

	/**
	 * Returns the vector obtained by multiplying this vector's X coordinate by factor {@code xFactor}
	 * and its Y coordinate by factor {@code yFactor}.
	 *
	 * @mutates nothing |
	 * @post | result != null
	 * @post | result.getX() == (int)Math.round(this.getX() * xFactor)
	 * @post | result.getY() == (int)Math.round(this.getY() * yFactor)
	 */
	public IntVector scale(double xFactor, double yFactor) {
		return new IntVector((int)Math.round(this.x * xFactor), (int)Math.round(this.y * yFactor));
	}

	/**
	 * Returns whether this vector is collinear with the given vector.
	 *
	 * @pre | other != null
	 * @post | result == (this.crossProduct(other) == 0)
	 */
	public boolean isCollinearWith(IntVector other) {
		return crossProduct(other) == 0;
	}

	/**
	 * Returns the dot product of this vector and the given vector.
	 *
	 * @pre | other != null
	 * @post | result == (long)getX() * other.getX() + (long)getY() * other.getY()
	 */
	public long dotProduct(IntVector other) { return (long)x * other.x + (long)y * other.y; }

	/**
	 * Returns a {@code DoubleVector} object that represents the same vector represented by this {@code IntVector} object.
	 *
	 * @post | result != null
	 */
	public DoubleVector asDoubleVector() {
		return new DoubleVector(this.x, this.y);
	}

}
```

```java
package drawit;

import java.util.Arrays;
import java.util.stream.IntStream;

/**
 * Declares a number of methods useful for working with arrays of {@code IntPoint} objects.
 */
public class PointArrays {

	private PointArrays() { throw new AssertionError("This class is not meant to be instantiated"); }

	/**
	 * Returns {@code null} if the given array of points defines a proper polygon; otherwise, returns a string describing wh
y it does not.
	 *
	 * <p>We interpret an array of N points as the polygon whose vertices are the given points and
	 * whose edges are the open line segments between point I and point (I + 1) % N, for I = 0, ..., N - 1.
	 *
	 * <p>A proper polygon is one where no two vertices coincide and no vertex is on any edge and no two edges intersect.
	 *
	 * <p>If there are exactly two points, the polygon is not proper. Notice that if there are more than two points and no t
wo vertices
	 * coincide and no vertex is on any edge, then no two edges intersect at more than one point.
	 *
	 * @pre | points != null
	 * @pre | Arrays.stream(points).allMatch(p -> p != null)
	 * @inspects | points
	 * @mutates nothing |
	 * @post
	 *     | (result == null) == (
	 *     |     points.length != 2 &&
	 *     |     IntStream.range(0, points.length).allMatch(i ->
	 *     |         IntStream.range(0, points.length).allMatch(j -> i == j || !points[i].equals(points[j]))) &&
	 *     |     IntStream.range(0, points.length).allMatch(i ->
	 *     |         IntStream.range(0, points.length).allMatch(j -> !points[i].isOnLineSegment(points[j], points[(j + 1) % p
oints.length]))) &&
	 *     |     IntStream.range(0, points.length).allMatch(i ->
	 *     |         IntStream.range(0, points.length).allMatch(j -> i == j ||
	 *     |             !IntPoint.lineSegmentsIntersect(points[i], points[(i + 1) % points.length], points[j], points[(j + 1
) % points.length])))
	 *     | )
	 */
	public static String checkDefinesProperPolygon(IntPoint[] points) {
		if (points.length == 2)
			return "Line segment 0 intersects with line segment 1";
		// If `points.length != 2`, then either some vertices are on some line segments or the line segments have at mos
t one point in common.
		for (int i = 0; i < points.length - 1; i++) {
			for (int j = i + 1; j < points.length; j++) {
				if (points[i].equals(points[j]))
					return "IntPoint " + i + " coincides with point " + j;
				if (points[i].isOnLineSegment(points[j], points[(j + 1) % points.length]))
					return "IntPoint " + i + " is on line segment " + j;
				if (points[j].isOnLineSegment(points[i], points[i + 1]))
					return "IntPoint " + j + " is on line segment " + i;
				if (IntPoint.lineSegmentsIntersect(points[i], points[i + 1], points[j], points[(j + 1) % points.
length]))
					return "Line segment " + i + " intersects with line segment " + j;
			}
		}
		return null;
	}

	/**
	 * Returns a new array with the same contents as the given array.
	 * @pre | points != null
	 * @pre | Arrays.stream(points).allMatch(p -> p != null)
	 * @inspects | points
	 * @mutates nothing |
	 * @creates | result
	 * @post | result != null
	 * @post | result.length == points.length
	 * @post | IntStream.range(0, points.length).allMatch(i -> result[i].equals(points[i]))
	 */
	public static IntPoint[] copy(IntPoint[] points) {
		IntPoint[] result = new IntPoint[points.length];
		for (int i = 0; i < points.length; i++)
			result[i] = points[i];
		return result;
	}

	/**
	 * Returns a new array whose elements are the elements of the given array with the given point inserted at the given ind
ex.
	 *
	 * @pre | points != null
	 * @pre | Arrays.stream(points).allMatch(p -> p != null)
	 * @pre | 0 <= index && index <= points.length
	 * @pre | point != null
	 * @inspects | points
	 * @mutates nothing |
	 * @creates | result
	 * @post | result != null
	 * @post | result.length == points.length + 1
	 * @post | IntStream.range(0, index).allMatch(i -> result[i].equals(points[i]))
	 * @post | result[index].equals(point)
	 * @post | IntStream.range(index, points.length).allMatch(i -> result[i + 1].equals(points[i]))
	 */
	// Javadoc notes:
	// - It's also acceptable to allow null elements, but then you have to use Objects.equals or == to compare elements
	// - It's also acceptable to compare elements using == in this case
	public static IntPoint[] insert(IntPoint[] points, int index, IntPoint point) {
		IntPoint[] result = new IntPoint[points.length + 1];
		for (int i = 0; i < index; i++)
			result[i] = points[i];
		result[index] = point;
		for (int i = index; i < points.length; i++)
			result[i + 1] = points[i];
		return result;
	}

	/**
	 * Returns a new array whose elements are the elements of the given array with the element at the given index removed.
	 *
	 * @pre | points != null
	 * @pre | Arrays.stream(points).allMatch(p -> p != null)
	 * @pre | 0 <= index && index < points.length
	 * @inspects | points
	 * @mutates nothing |
	 * @creates | result
	 * @post | result != null
	 * @post | result.length == points.length - 1
	 * @post | IntStream.range(0, index).allMatch(i -> result[i].equals(points[i]))
	 * @post | IntStream.range(index + 1, points.length).allMatch(i -> result[i - 1].equals(points[i]))
	 */
	public static IntPoint[] remove(IntPoint[] points, int index) {
		IntPoint[] result = new IntPoint[points.length - 1];
		for (int i = 0; i < index; i++)
			result[i] = points[i];
		for (int i = index; i < result.length; i++)
			result[i] = points[i + 1];
		return result;
	}

	/**
	 * Returns a new array whose elements are the elements of the given array with the element at the given index replaced b
y the given point.
	 *
	 * @pre | points != null
	 * @pre | 0 <= index && index < points.length
	 * @pre | value != null
	 * @inspects | points
	 * @mutates nothing |
	 * @creates | result
	 * @post | result != null
	 * @post | result.length == points.length
	 * @post | IntStream.range(0, points.length).allMatch(i -> result[i].equals(i == index ? value : points[i]))
	 */
	public static IntPoint[] update(IntPoint[] points, int index, IntPoint value) {
		IntPoint[] result = copy(points);
		result[index] = value;
		return result;
	}

	/**
	 * Returns a new array whose elements are the elements of the given array, translated along the given vector.
	 *
	 * @pre | points != null
	 * @pre | delta != null
	 * @inspects | points
	 * @mutates nothing |
	 * @creates | result
	 * @post | result != null
	 * @post | result.length == points.length
	 * @post | IntStream.range(0, points.length).allMatch(i -> result[i].equals(points[i].plus(delta)))
	 */
	public static IntPoint[] translate(IntPoint[] points, IntVector delta) {
		return Arrays.stream(points).map(p -> p.plus(delta)).toArray(n -> new IntPoint[n]);
	}
```

```java
package drawit;

import java.awt.Color;
import java.util.Arrays;

/**
 * An instance of this class is a mutable abstraction storing a rounded polygon defined by a set of 2D points with integer coord
inates
 * and a nonnegative corner radius.
 *
 * @invar | getVertices() != null
 * @invar | Arrays.stream(getVertices()).allMatch(v -> v != null)
 * @invar | PointArrays.checkDefinesProperPolygon(getVertices()) == null
 * @invar | 0 <= getRadius()
 * @invar | getColor() != null
 */
public class RoundedPolygon {

	/**
	 * @representationObject
	 * @invar | vertices != null
	 * @invar | Arrays.stream(vertices).allMatch(v -> v != null)
	 * @invar | PointArrays.checkDefinesProperPolygon(vertices) == null
	 * @invar | 0 <= radius
	 */
	private IntPoint[] vertices = new IntPoint[0];
	private int radius;
	private Color color = Color.yellow;

	/**
	 * Returns a new array whose elements are the vertices of this rounded polygon.
	 *
	 * @creates | result
	 */
	public IntPoint[] getVertices() {
		return PointArrays.copy(vertices);
	}

	/**
	 * Returns the radius of the corners of this rounded polygon.
	 */
	public int getRadius() { return radius; }

	public Color getColor() { return color; }

	/**
	 * @mutates | this
	 * @post | getVertices().length == 0
	 * @post | getRadius() == 0
	 * @post | getColor().equals(Color.yellow)
	 */
	public RoundedPolygon() {}

	/**
	 * Sets the vertices of this rounded polygon to be equal to the elements of the given array.
	 *
	 * @inspects | newVertices
	 * @mutates | this
	 * @throws IllegalArgumentException | newVertices == null
	 * @throws IllegalArgumentException | Arrays.stream(newVertices).anyMatch(v -> v == null)
	 * @throws IllegalArgumentException if the given vertices do not define a proper polygon.
	 *     | PointArrays.checkDefinesProperPolygon(newVertices) != null
	 * @post | Arrays.equals(getVertices(), newVertices)
	 * @post | getRadius() == old(getRadius())
	 * @post | getColor().equals(old(getColor()))
	 */
	public void setVertices(IntPoint[] newVertices) {
		if (newVertices == null)
			throw new IllegalArgumentException("newVertices is null");
		if (Arrays.stream(newVertices).anyMatch(v -> v == null))
			throw new IllegalArgumentException("An element of newVertices is null");
		IntPoint[] copy = PointArrays.copy(newVertices);
		String msg = PointArrays.checkDefinesProperPolygon(copy);
		if (msg != null)
			throw new IllegalArgumentException(msg);
		vertices = copy;
	}

	/**
	 * Sets this rounded polygon's corner radius to the given value.
	 *
	 * @throws IllegalArgumentException if the given radius is negative.
	 *     | radius < 0
	 * @mutates | this
	 * @post | Arrays.equals(getVertices(), old(getVertices()))
	 * @post | getRadius() == radius
	 * @post | getColor().equals(old(getColor()))
	 */
	public void setRadius(int radius) {
		if (radius < 0)
			throw new IllegalArgumentException("The given radius is negative");
		this.radius = radius;
	}

	public void setColor(Color color) {
		this.color = color;
	}

	/**
	 * @throws IllegalArgumentException | !(0 <= index && index <= getVertices().length)
	 * @throws IllegalArgumentException | point == null
	 * @throws IllegalArgumentException | PointArrays.checkDefinesProperPolygon(PointArrays.insert(getVertices(), index, poi
nt)) != null
	 * @mutates | this
	 * @post | Arrays.equals(getVertices(), PointArrays.insert(old(getVertices()), index, point))
	 * @post | getRadius() == old(getRadius())
	 * @post | getColor().equals(old(getColor()))
	 */
	public void insert(int index, IntPoint point) {
		if (!(0 <= index && index <= getVertices().length))
			throw new IllegalArgumentException("index out of range");
		if (point == null)
			throw new IllegalArgumentException("point is null");
		setVertices(PointArrays.insert(vertices, index, point));
	}

	/**
	 * @throws IllegalArgumentException | !(0 <= index && index < getVertices().length)
	 * @throws IllegalArgumentException | PointArrays.checkDefinesProperPolygon(PointArrays.remove(getVertices(), index)) !=
 null
	 * @mutates | this
	 * @post | Arrays.equals(getVertices(), PointArrays.remove(old(getVertices()), index))
	 * @post | getRadius() == old(getRadius())
	 * @post | getColor().equals(old(getColor()))
	 */
	public void remove(int index) {
		if (!(0 <= index && index < getVertices().length))
			throw new IllegalArgumentException("index out of range");
		setVertices(PointArrays.remove(vertices, index));
	}

	/**
	 * @throws IllegalArgumentException | !(0 <= index && index < getVertices().length)
	 * @throws IllegalArgumentExeption | point == null
	 * @throws IllegalArgumentException | PointArrays.checkDefinesProperPolygon(PointArrays.update(getVertices(), index, poi
nt)) != null
	 * @mutates | this
	 * @post | Arrays.equals(getVertices(), PointArrays.update(old(getVertices()), index, point))
	 * @post | getRadius() == old(getRadius())
	 * @post | getColor().equals(old(getColor()))
	 */
	public void update(int index, IntPoint point) {
		if (!(0 <= index && index < getVertices().length))
			throw new IllegalArgumentException("index out of range");
		if (point == null)
			throw new IllegalArgumentException("point is null");
		setVertices(PointArrays.update(vertices, index, point));
	}

	/**
	 * Returns {@code true} iff the given point is contained by the (non-rounded) polygon defined by this rounded polygon's
vertices.
	 * This method does not take into account this rounded polygon's corner radius; it assumes a corner radius of zero.
	 *
	 * <p>A point is contained by a polygon if it coincides with one of its vertices, or if it is on one of its edges, or if
 it is in the polygon's interior.
	 *
	 * <p><b>Implementation hints:</b> A point is in the interior of a polygon if
	 * a line from that point to infinity in any direction (an "exit path" for the point) intersects with the polygon's peri
meter an odd number of times.
	 *
	 * <p>Use the line from the given point in the positive X direction as the exit path.
	 *
	 * <p>First, find the first vertex that is not on the exit path.
	 * If you don't find such a vertex, return {@code false}.
	 *
	 * <p>Now, starting with this vertex, V, find the next vertex, V', that is not on the exit path.
	 * If there are no vertices on the exit path between V and V', then check if the edge VV' intersects with the exit path:
	 * check that VV' straddles the carrier of the exit path and that (the signum of the cross product of VP and VV') times
(the signum of the cross
	 * product of +X and VV') is negative.
	 * If there are vertices on the exit path between V and V', then count one intersection between the exit path and the po
lygon's perimeter iff
	 * VV' straddles the carrier of the exit path.
	 *
	 * <p>Repeat this until you again reach the first vertex that is not on the exit path.
```

```
         *
         * @pre | point != null
         * @inspects | this
         * @mutates nothing |
         */
        public boolean contains(IntPoint point) {
                // We call the half-line extending from `point` to the right the "exit path"
                // Find first vertex that is not on the exit path
                int firstVertex;
                {
                        int i = 0;
                        for (;;) {
                                if (i == vertices.length) // Zero or one vertices
                                        return false;
                                if (vertices[i].equals(point))
                                        return true;
                                if (!(vertices[i].getY() == point.getY() && vertices[i].getX() > point.getX()))
                                        break;
                                i++;
                        }
                        firstVertex = i;
                }
                IntVector exitVector = new IntVector(1, 0);
                // Count how many times the exit path crosses the polygon
                int nbEdgeCrossings = 0;
                for (int index = firstVertex; ; ) {
                        IntPoint a = vertices[index];
                        // Find the next vertex that is not on the exit path
                        boolean onExitPath = false;
                        int nextIndex = index;
                        IntPoint b;
                        for (;;) {
                                int nextNextIndex = (nextIndex + 1) % vertices.length;
                                if (point.isOnLineSegment(vertices[nextIndex], vertices[nextNextIndex]))
                                        return true;
                                nextIndex = nextNextIndex;
                                b = vertices[nextIndex];
                                if (b.equals(point))
                                        return true;
                                if (b.getY() == point.getY() && b.getX() > point.getX()) {
                                        onExitPath = true;
                                        continue;
                                }
                                break;
                        }
                        if (onExitPath) {
                                if ((b.getY() < point.getY()) != (a.getY() < point.getY()))
                                        nbEdgeCrossings++;
                        } else {
                                // Does `ab` straddle the exit path's carrier?
                                if (Math.signum(a.getY() - point.getY()) * Math.signum(b.getY() - point.getY()) < 0) {
                                        // Does the exit path straddle `ab`'s carrier?
                                        IntVector ab = b.minus(a);
                                        if (Math.signum(point.minus(a).crossProduct(ab)) * Math.signum(exitVector.crossProduct(a
b)) < 0)
                                                nbEdgeCrossings++;
                                }
                        }
                        if (nextIndex == firstVertex)
                                break;
                        index = nextIndex;
                }
                return nbEdgeCrossings % 2 == 1;
        }

        /**
         * Returns a textual representation of a set of drawing commands for drawing this rounded polygon.
         *
         * <p>The returned text consists of a sequence of drawing operators and arguments, separated by spaces. The drawing oper
ators are
         * {@code line} and {@code arc}. Each argument is a decimal representation
         * of a floating-point number.
         *
         * <p>Operator {@code line} takes four arguments: X1 Y1 X2 Y2; it draws a line between (X1, Y1) and (X2, Y2).
         * {@code arc} takes five: X Y R S E.
         * It draws a part of a circle. The circle is defined by its center (X, Y) and its radius R. The part to draw is defined
 by the start angle A
         * and angle extent E, both in radians. Positive X is angle zero; positive Y is angle {@code Math.PI / 2}; negative Y is
 angle {@code -Math.PI / 2}.
         *
         * <p>For example, the following commands draw a rounded square with corner radius 10:
         * <pre>
     * line 110 100 190 100
         * arc 190 110 10 -1.5707963267948966 1.5707963267948966
         * line 200 110 200 190
         * arc 190 190 10 0 1.5707963267948966
         * line 190 200 110 200
         * arc 110 190 10 1.5707963267948966 1.5707963267948966
         * line 100 190 100 110
         * arc 110 110 10 3.141592653589793 1.5707963267948966
         * </pre>
         *
         * <p>By rounding a corner, the adjacent edges are cut short by some amount.
         * The corner radius to be used for a particular corner is the largest radius that is not greater than this rounded poly
gon's corner radius
         * and that is such that no more than half of each adjacent edge is cut off by it.
         *
         * <p><b>Implementation hints:</b>
         * First, if this rounded polygon has less than three vertices, return an empty string.
         *
         * <p>Then, draw each corner, including half of each adjacent edge. Let B be the vertex for which we are drawing the cor
ner; let A be the preceding
         * vertex and C the succeding one. Let BAC be the center of the line segment BA, and BCC be the center of the line segme
nt BC.
         *
         * <p>If BA and BC are collinear, just draw the lines BAC-B and B-BCC.
         *
         * <p>Otherwise, let BAU be the unit vector from B to A, and BCU the unit vector from B to C.
         * Then BAU + BCU points in the direction of the bisector. Let BSU be the unit vector in that direction.
         *
         * <p>First, suppose the center of the corner would be at B + BSU. Compute how much is cut off from BA by projecting BSU
 onto BA:
         * BAUcutoff = dot product of BAU and BSU
         * (By symmetry, the same amount is cut off from BC.)
         * The radius of the corner would then be unitRadius = absolute value of cross product of BSU and BAU.
         * Now, determine the scale factor to apply: this is the minimum of the scale factor that would scale unitRadius to this
.getRadius() and the scale
         * factor that would scale BAUcutoff to half of the minimum of the size of BA and BC.
         * From this, we can easily determine the actual center and actual radius of the corner.
         *
         * <p>To determine the start angle, use the cutoff length to compute the point on BA where the arc starts,
         * and turn the vector from the corner's center to that point into
         * an angle. Similarly, compute the end angle. The angle extent is the difference between the two, after adding or subtr
acting 2PI as necessary
         * to obtain a value between -PI and PI.
         *
         * @inspects | this
         * @mutates nothing |
         * @post | result != null
         */
        public String getDrawingCommands() {
                if (vertices.length < 3)
                        return "";
                StringBuilder commands = new StringBuilder();
                for (int index = 0; index < vertices.length; index++) {
                        IntPoint a = vertices[(index + vertices.length - 1) % vertices.length];
                        IntPoint b = vertices[index];
                        IntPoint c = vertices[(index + 1) % vertices.length];
                        DoubleVector ba = a.minus(b).asDoubleVector();
                        DoubleVector bc = c.minus(b).asDoubleVector();
                        DoublePoint baCenter = b.asDoublePoint().plus(ba.scale(0.5));
                        DoublePoint bcCenter = b.asDoublePoint().plus(bc.scale(0.5));
                        double baSize = ba.getSize();
                        double bcSize = bc.getSize();
                        if (ba.crossProduct(bc) == 0) {
                                commands.append("line " + bcCenter.getX() + " " + bcCenter.getY() + " " + b.getX() + " " + b.get
Y() + "\n");
                                commands.append("line " + b.getX() + " " + b.getY() + " " + baCenter.getX() + " " + baCenter.get
Y() + "\n");
                        } else {
                                DoubleVector baUnit = ba.scale(1/baSize);
                                DoubleVector bcUnit = bc.scale(1/bcSize);
                                DoubleVector bisector = baUnit.plus(bcUnit);
                                bisector = bisector.scale(1/bisector.getSize());
                                double unitEdgeDistance = baUnit.dotProduct(bisector);
                                double unitRadius = Math.abs(bisector.crossProduct(baUnit));
                                double scaleFactor = Math.min(this.radius / unitRadius, Math.min(baSize, bcSize) / 2 / unitEdgeD
istance);
                                DoublePoint center = b.asDoublePoint().plus(bisector.scale(scaleFactor));
                                double radius = unitRadius * scaleFactor;
                                DoublePoint bcCornerStart = b.asDoublePoint().plus(bcUnit.scale(unitEdgeDistance * scaleFactor))
;
                                DoublePoint baCornerStart = b.asDoublePoint().plus(baUnit.scale(unitEdgeDistance * scaleFactor))
;
                                double baAngle = baCornerStart.minus(center).asAngle();
                                double bcAngle = bcCornerStart.minus(center).asAngle();
                                double angleExtent = bcAngle - baAngle;
                                if (angleExtent < -Math.PI)
                                        angleExtent += 2 * Math.PI;
                                else if (Math.PI < angleExtent)
                                        angleExtent -= 2 * Math.PI;
                                commands.append("line " + baCenter.getX() + " " + baCenter.getY() + " " + baCornerStart.getX() +
 " " + baCornerStart.getY() + "\n");
                                commands.append("arc " + center.getX() + " " + center.getY() + " " + radius + " " + baAngle + "
" + angleExtent + "\n");
```

```java
package drawit.shapegroups1;

import drawit.IntPoint;

/**
 * Each instance of this class represents a rectangular area in a 2D coordinate
 * system, whose edges are parallel to the coordinate axes.
 *
 * Note: the "top" and "bottom" terminology used by this class assumes
 * that the Y axis points down, as is common in computer graphics.
 *
 * This class must deal with illegal arguments defensively.
 *
 * @immutable
 *
 * @invar | getLeft() <= getRight()
 * @invar | getTop() <= getBottom()
 * @invar | getWidthAsLong() == (long)getRight() - getLeft()
 * @invar | getHeightAsLong() == (long)getBottom() - getTop()
 */
public class Extent {

	/**
	 * @invar | left <= right
	 * @invar | top <= bottom
	 */
	private final int left;
	private final int top;
	private final int right;
	private final int bottom;

	/**
	 * Returns the X coordinate of the edge parallel to the Y axis
	 * with the smallest X coordinate.
	 */
	public int getLeft() { return left; }
	/**
	 * Returns the Y coordinate of the edge parallel to the X axis
	 * with the smallest Y coordinate.
	 */
	public int getTop() { return top; }
	/**
	 * Returns the X coordinate of the edge parallel to the Y axis
	 * with the largest X coordinate.
	 */
	public int getRight() { return right; }
	/**
	 * Returns the Y coordinate of the edge parallel to the X axis
	 * with the largest Y coordinate.
	 */
	public int getBottom() { return bottom; }
	/**
	 * Returns the distance between the edges that are parallel to the Y axis.
	 */
	public long getWidthAsLong() { return (long)right - left; }
	/**
	 * Returns the distance between the edges that are parallel to the Y axis.
	 *
	 * @throws UnsupportedOperationException if the width cannot be represented as an {@code int}
	 *     | getWidthAsLong() > Integer.MAX_VALUE
	 * @post | result == getWidthAsLong()
	 */
	public int getWidth() {
		long width = (long)right - left;
		if (width > Integer.MAX_VALUE)
			throw new UnsupportedOperationException("width too big");
		return (int)width;
	}
	/**
	 * Returns the distance between the edges that are parallel to the X axis.
	 */
	public long getHeightAsLong() { return (long)bottom - top; }
	/**
	 * Returns the distance between the edges that are parallel to the X axis.
	 *
	 * @throws UnsupportedOperationException if the height cannot be represented as an {@code int}
	 *     | getHeightAsLong() > Integer.MAX_VALUE
	 * @post | result == getHeightAsLong()
	 */
	public int getHeight() {
		long height = (long)bottom - top;
		if (height > Integer.MAX_VALUE)
			throw new UnsupportedOperationException("height too big");
		return (int)height;
	}

	/**
	 * Returns the top-left corner of this extent.
	 *
	 * @post | result != null
	 * @post | result.equals(new IntPoint(getLeft(), getTop()))
	 */
	public IntPoint getTopLeft() { return new IntPoint(left, top); }

	/**
	 * Returns the bottom-right corner of this extent.
	 *
	 * @post | result != null
	 * @post | result.equals(new IntPoint(getRight(), getBottom()))
	 */
	public IntPoint getBottomRight() { return new IntPoint(right, bottom); }

	/**
	 * Returns whether this extent, considered as a closed set of points (i.e.
	 * including its edges and its vertices), contains the given point.
	 *
	 * @throws IllegalArgumentException if {@code point} is null
	 *     | point == null
	 * @post
	 *     | result == (
	 *     |     getLeft() <= point.getX() && point.getX() <= getRight() &&
	 *     |     getTop() <= point.getY() && point.getY() <= getBottom()
	 *     | )
	 */
	public boolean contains(IntPoint point) {
		return
				getLeft() <= point.getX() && point.getX() <= getRight() &&
				getTop() <= point.getY() && point.getY() <= getBottom();
	}

	/**
	 * Returns whether this extent equals the given extent.
	 *
	 * @post | result == (
	 *       |     other != null &&
	 *       |     getTopLeft().equals(other.getTopLeft()) &&
	 *       |     getBottomRight().equals(other.getBottomRight())
	 *       | )
	 */
	public boolean equals(Extent other) {
		return other != null && left == other.left && top == other.top && right == other.right && bottom == other.bottom
;
	}

	/**
	 * Returns whether this extent equals the given object.
	 *
	 * @post | result == (other instanceof Extent && this.equals((Extent)other))
	 */
	@Override
	public boolean equals(Object other) {
		return other instanceof Extent && equals((Extent)other);
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + bottom;
		result = prime * result + left;
		result = prime * result + right;
		result = prime * result + top;
		return result;
	}

	@Override
	public String toString() {
		return "drawit.shapegroups1.Extent[left="+left+", top="+top+", right="+right+", bottom="+bottom+"]";
	}

	private Extent(int left, int top, int right, int bottom) {
		this.left = left;
		this.top = top;
		this.right = right;
		this.bottom = bottom;
	}

	/**
	 * Returns an object representing the extent defined by the given left, top, width, and height.
	 *
	 * @throws IllegalArgumentException if the given width is negative
	 *     | width < 0
	 * @throws IllegalArgumentException if the given height is negative
	 *     | height < 0
	 * @throws IllegalArgumentException if the sum of {@code left} and {@code width} is greater than {@code Integer.MAX_VALU
E}
```

```
     *    | Integer.MAX_VALUE - width < left
     * @throws IllegalArgumentException if the sum of {@code top} and {@code height} is greater than {@code Integer.MAX_VALU
E}
     *    | Integer.MAX_VALUE - height < top
     * @post | result != null
     * @post | result.getLeft() == left
     * @post | result.getTop() == top
     * @post | result.getWidth() == width
     * @post | result.getHeight() == height
     */
    public static Extent ofLeftTopWidthHeight(int left, int top, int width, int height) {
            if (width < 0)
                    throw new IllegalArgumentException("width is negative");
            if (height < 0)
                    throw new IllegalArgumentException("height is negative");
            if (Integer.MAX_VALUE - width < left)
                    throw new IllegalArgumentException("left + width too large");
            if (Integer.MAX_VALUE - height < top)
                    throw new IllegalArgumentException("top + height too large");
            return new Extent(left, top, left + width, top + height);
    }

    /**
     * Returns an object representing the extent defined by the given left, top, right, and bottom.
     *
     * @throws IllegalArgumentException if the given right less than the given left
     *    | right < left
     * @throws IllegalArgumentException if the given bottom is less than the given top
     *    | bottom < top
     * @post | result != null
     * @post | result.getLeft() == left
     * @post | result.getTop() == top
     * @post | result.getRight() == right
     * @post | result.getBottom() == bottom
     */
    public static Extent ofLeftTopRightBottom(int left, int top, int right, int bottom) {
            if (right < left)
                    throw new IllegalArgumentException("right less than left");
            if (bottom < top)
                    throw new IllegalArgumentException("bottom less than top");
            return new Extent(left, top, right, bottom);
    }

    /**
     * Returns an object that has the given left coordinate and the same
     * right, top, and bottom coordinate as this object.
     *
     * @throws IllegalArgumentException if the given left coordinate is greater than this extent's right coordinate
     *    | getRight() < newLeft
     * @post | result != null
     * @post | result.getLeft() == newLeft
     * @post | result.getTop() == getTop()
     * @post | result.getRight() == getRight()
     * @post | result.getBottom() == getBottom()
     */
    public Extent withLeft(int newLeft) {
            if (getRight() < newLeft)
                    throw new IllegalArgumentException("newLeft greater than getRight()");
            return new Extent(newLeft, top, right, bottom);
    }

    /**
     * Returns an object that has the given top coordinate and the same
     * left, right, and bottom coordinate as this object.
     *
     * @throws IllegalArgumentException if the given left coordinate is greater than this extent's right coordinate
     *    | getBottom() < newTop
     * @post | result != null
     * @post | result.getLeft() == getLeft()
     * @post | result.getTop() == newTop
     * @post | result.getRight() == getRight()
     * @post | result.getBottom() == getBottom()
     */
    public Extent withTop(int newTop) {
            if (getBottom() < newTop)
                    throw new IllegalArgumentException("newLeft greater than getRight()");
            return new Extent(left, newTop, right, bottom);
    }

    /**
     * Returns an object that has the given right coordinate and the same
     * left, top, and bottom coordinate as this object.
     *
     * @throws IllegalArgumentException if the given left coordinate is greater than this extent's right coordinate
     *    | newRight < getLeft()
     * @post | result != null
     * @post | result.getLeft() == getLeft()
     * @post | result.getTop() == getTop()
     * @post | result.getRight() == newRight
     * @post | result.getBottom() == getBottom()
     */
    public Extent withRight(int newRight) {
            if (newRight < getLeft())
                    throw new IllegalArgumentException("newLeft greater than getRight()");
            return new Extent(left, top, newRight, bottom);
    }

    /**
     * Returns an object that has the given bottom coordinate and the same
     * left, top, and right coordinate as this object.
     *
     * @throws IllegalArgumentException if the given left coordinate is greater than this extent's right coordinate
     *    | newBottom < getTop()
     * @post | result != null
     * @post | result.getLeft() == getLeft()
     * @post | result.getTop() == getTop()
     * @post | result.getRight() == getRight()
     * @post | result.getBottom() == newBottom
     */
    public Extent withBottom(int newBottom) {
            if (newBottom < getTop())
                    throw new IllegalArgumentException("newLeft greater than getRight()");
            return new Extent(left, top, right, newBottom);
    }

    /**
     * Returns an object that has the given width and the same left, top,
     * and bottom coordinate as this object.
     *
     * @throws IllegalArgumentException if the given width is negative
     *    | newWidth < 0
     * @throws IllegalArgumentException if the new right coordinate would be greater than {@code Integer.MAX_VALUE}
     *    | Integer.MAX_VALUE - newWidth < getLeft()
     * @post | result != null
     * @post | result.getLeft() == getLeft()
     * @post | result.getTop() == getTop()
     * @post | result.getWidth() == newWidth
     * @post | result.getHeight() == getHeight()
     */
    public Extent withWidth(int newWidth) {
            if (newWidth < 0)
                    throw new IllegalArgumentException("newWidth negative");
            if (Integer.MAX_VALUE - newWidth < getLeft())
                    throw new IllegalArgumentException("new right coordinate would be greater than Integer.MAX_VALUE");
            return new Extent(left, top, left + newWidth, bottom);
    }

    /**
     * Returns an object that has the given height and the same left, top,
     * and right coordinate as this object.
     *
     * @throws IllegalArgumentException if the given height is negative
     *    | newHeight < 0
     * @throws IllegalArgumentException if the new bottom coordinate would be greater than {@code Integer.MAX_VALUE}
     *    | Integer.MAX_VALUE - newHeight < getTop()
     * @post | result != null
     * @post | result.getLeft() == getLeft()
     * @post | result.getTop() == getTop()
     * @post | result.getWidth() == getWidth()
     * @post | result.getHeight() == newHeight
     */
    public Extent withHeight(int newHeight) {
            if (newHeight < 0)
                    throw new IllegalArgumentException("newHeight negative");
            if (Integer.MAX_VALUE - newHeight < getTop())
                    throw new IllegalArgumentException("new bottom coordinate would be greater than Integer.MAX_VALUE");
            return new Extent(left, top, right, top + newHeight);
    }
}
```

```java
package drawit.shapegroups1;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Set;
import java.util.stream.Collectors;

import drawit.IntPoint;
import drawit.IntVector;
import drawit.PointArrays;
import drawit.RoundedPolygon;

import logicalcollections.LogicalList;
import logicalcollections.LogicalSet;

/**
 * Each instance of this class represents a shape group. A shape group is either a leaf group,
 * in which case it directly contains a single shape, or it is a non-leaf group, in which case it directly contains
 * two or more subgroups, which are themselves shape groups.
 *
 * @invar | getParentGroup() == null ||
 *        | getParentGroup().getSubgroups() != null && getParentGroup().getSubgroups().contains(this)
 * @invar | !getAncestors().contains(this)
 */
public abstract class ShapeGroup {

	/**
	 * @invar | parent == null || parent.subgroups.contains(this)
	 * @invar | !getAncestorsPrivate().contains(this)
	 *
	 * @peerObject
	 */
	NonleafShapeGroup parent;

	/**
	 * Returns the set of the ancestors of this shape group.
	 *
	 * @post | result != null
	 * @post | result.equals(LogicalSet.<ShapeGroup>matching(ancestors ->
	 *       |     getParentGroup() == null || ancestors.contains(getParentGroup()) &&
	 *       |     ancestors.allMatch(ancestor ->
	 *       |         ancestor.getParentGroup() == null || ancestors.contains(ancestor.getParentGroup()))))
	 */
	public Set<ShapeGroup> getAncestors() {
		return getAncestorsPrivate();
	}

	Set<ShapeGroup> getAncestorsPrivate() {
		return LogicalSet.<ShapeGroup>matching(ancestors ->
			parent == null || ancestors.contains(parent) &&
			ancestors.allMatch(ancestor -> ancestor.parent == null || ancestors.contains(ancestor.parent))
		);
	}

	/**
	 * Returns the shape group that directly contains this shape group, or {@code null}
	 * if no shape group directly contains this shape group.
	 *
	 * @basic
	 * @peerObject
	 */
	public NonleafShapeGroup getParentGroup() { return parent; }

	/**
	 * Returns the list of all RoundedPolygon objects contained directly or
	 * indirectly by this shape group, in depth-first order.
	 *
	 * @post The result is not null.
	 *     | result != null
	 * @post The elements of the result are not null.
	 *       (More details are given in the subclasses.)
	 *     | result.stream().allMatch(s -> s != null)
	 */
	public abstract List<RoundedPolygon> getAllShapes();

	/**
	 * Returns a map that maps each RoundedPolygon object contained directly or
	 * indirectly by this shape group to its current list of vertices.
	 *
	 * @inspects | this, ...getAllShapes()
	 * @post | result != null
	 * @post | result.keySet().equals(Set.copyOf(getAllShapes()))
	 * @post | getAllShapes().stream().allMatch(s -> Arrays.equals(result.get(s), s.getVertices()))
	 */
	public Map<RoundedPolygon, IntPoint[]> getAllVertices() {
		return getAllShapes().stream().collect(Collectors.toMap(s -> s, s -> s.getVertices()));
	}

	/**
	 * Returns the smallest extent that contains all of the shapes contained directly or indirectly by this shape group.
	 *
	 * @inspects | this, ...getAllShapes()
	 * @post | result != null
	 * @post | result.getLeft() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getX()).min().getAsInt()
	 * @post | result.getTop() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getY()).min().getAsInt()
	 * @post | result.getRight() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getX()).max().getAsInt()
	 * @post | result.getBottom() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getY()).max().getAsInt()
	 */
	public abstract Extent getBoundingBox();

	/**
	 * Returns a textual representation of a sequence of drawing commands for drawing
	 * the shapes contained directly or indirectly by this shape group.
	 *
	 * For the syntax of the drawing commands, see {@code RoundedPolygon.getDrawingCommands()}.
	 *
	 * @inspects | this, ...getAllShapes()
	 * @post | result != null
	 */
	public abstract String getDrawingCommands();

	/**
	 * Moves this shape group to the front of its parent's list of subgroups.
	 *
	 * @throws UnsupportedOperationException if this shape group has no parent
	 *     | getParentGroup() == null
	 * @mutates_properties | getParentGroup().getSubgroups()
	 * @post | getParentGroup().getSubgroups().equals(
	 *       |     LogicalList.plusAt(LogicalList.minus(old(getParentGroup().getSubgroups()), this), 0, this))
	 */
	public void bringToFront() {
		if (parent == null)
			throw new UnsupportedOperationException("no parent");

		parent.subgroups.remove(this);
		parent.subgroups.add(0, this);
	}

	/**
	 * Moves this shape group to the back of its parent's list of subgroups.
	 *
	 * @throws UnsupportedOperationException if this shape group has no parent
	 *     | getParentGroup() == null
	 * @mutates_properties | getParentGroup().getSubgroups()
	 * @post | getParentGroup().getSubgroups().equals(
	 *       |     LogicalList.plus(LogicalList.minus(old(getParentGroup().getSubgroups()), this), this))
	 */
	public void sendToBack() {
		if (parent == null)
			throw new UnsupportedOperationException("no parent");

		parent.subgroups.remove(this);
		parent.subgroups.add(this);
	}

	/**
	 * Translate (= displace) the shapes contained directly or indirectly by this shape group along
	 * the given vector.
	 *
	 * @throws IllegalArgumentException if {@code delta} is null
	 *     | delta == null
	 * @inspects | this
	 * @mutates_properties | (...getAllShapes()).getVertices()
	 * @post
	 *     | getAllShapes().stream().allMatch(s ->
	 *     |     Arrays.equals(s.getVertices(), PointArrays.translate(old(getAllVertices()).get(s), delta)))
	 */
	public void translate(IntVector delta) {
		if (delta == null)
			throw new IllegalArgumentException("delta is null");

		for (RoundedPolygon shape : getAllShapes())
			shape.setVertices(PointArrays.translate(shape.getVertices(), delta));
	}
```

```java
package drawit.shapegroups1;

import java.util.Arrays;
import java.util.List;
import java.util.Objects;

import drawit.IntPoint;
import drawit.RoundedPolygon;

/**
 * Each instance of this class represents a leaf shape group in a shape group graph.
 */
public class LeafShapeGroup extends ShapeGroup {

	/**
	 * @invar | shape != null
	 */
	final RoundedPolygon shape;

	/**
	 * Returns the shape directly contained by this leaf shape group.
	 *
	 * @immutable
	 *
	 * @post | result != null
	 */
	public RoundedPolygon getShape() { return shape; }

	/**
	 * {@inheritDoc}
	 *
	 * @post | Objects.equals(result, List.of(getShape()))
	 */
	public List<RoundedPolygon> getAllShapes() { return List.of(shape); }

	@Override
	public String getDrawingCommands() {
		return shape.getDrawingCommands();
	}

	/**
	 * Returns the smallest extent that contains all of the shapes contained directly or indirectly by this shape group.
	 *
	 * @inspects | this, ...getAllShapes()
	 * @post | result != null
	 * @post | result.getLeft() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getX
()).min().getAsInt()
	 * @post | result.getTop() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getY(
)).min().getAsInt()
	 * @post | result.getRight() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.get
X()).max().getAsInt()
	 * @post | result.getBottom() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.ge
tY()).max().getAsInt()
	 */
	@Override
	public Extent getBoundingBox() {
		IntPoint[] vertices = shape.getVertices();
		if (vertices.length == 0)
			throw new IllegalStateException("no vertices");
		int minX = Integer.MAX_VALUE;
		int maxX = Integer.MIN_VALUE;
		int minY = Integer.MAX_VALUE;
		int maxY = Integer.MIN_VALUE;
		for (IntPoint p : vertices) {
			minX = Math.min(minX, p.getX());
			maxX = Math.max(maxX, p.getX());
			minY = Math.min(minY, p.getY());
			maxY = Math.max(maxY, p.getY());
		}
		return Extent.ofLeftTopRightBottom(minX, minY, maxX, maxY);
	}

	/**
	 * Initializes this object to represent a leaf shape group that directly contains the given shape.
	 *
	 * @throws IllegalArgumentException if {@code shape} is null
	 *     | shape == null
	 * @throws IllegalArgumentException if {@code shape} has less than three vertices
	 *     | shape.getVertices().length < 3
	 * @mutates | this
	 * @post | getShape() == shape
	 * @post | getParentGroup() == null
	 */
	public LeafShapeGroup(RoundedPolygon shape) {
		if (shape == null)
			throw new IllegalArgumentException("shape is null");
		if (shape.getVertices().length < 3)
			throw new IllegalArgumentException("shape has less than three vertices");
		this.shape = shape;
	}

}
```

```java
package drawit.shapegroups1;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Objects;
import java.util.Set;
import java.util.stream.Collectors;

import drawit.IntPoint;
import drawit.RoundedPolygon;
import logicalcollections.LogicalList;

/**
 * Each instance of this class represents a non-leaf shape group in a shape group graph.
 *
 * @invar | getSubgroups() != null
 * @invar | LogicalList.distinct(getSubgroups())
 * @invar | getSubgroups().stream().allMatch(g -> g != null && g.getParentGroup() == this)
 */
public class NonleafShapeGroup extends ShapeGroup {

	/**
	 * @invar | LogicalList.distinct(subgroups)
	 * @invar | subgroups.stream().allMatch(g -> g != null && g.parent == this)
	 *
	 * @representationObject
	 * @peerObjects
	 */
	ArrayList<ShapeGroup> subgroups;

	/**
	 * Returns the list of subgroups of this shape group.
	 *
	 * @basic
	 * @peerObjects
	 */
	public List<ShapeGroup> getSubgroups() { return List.copyOf(subgroups); }

	/**
	 * Returns the number of subgroups of this non-leaf shape group.
	 *
	 * @post | result == getSubgroups().size()
	 */
	public int getSubgroupCount() {
		return subgroups.size();
	}

	/**
	 * Returns the subgroup at the given (zero-based) index in this non-leaf shape group's list of subgroups.
	 *
	 * @throws IllegalArgumentException if the given index is out of bounds
	 *     | index < 0 || getSubgroups().size() <= index
	 * @post | result == getSubgroups().get(index)
	 */
	public ShapeGroup getSubgroup(int index) {
		if (index < 0 || getSubgroups().size() <= index)
			throw new IllegalArgumentException("index out of bounds");
		return subgroups.get(index);
	}

	@Override
	public String getDrawingCommands() {
		StringBuilder builder = new StringBuilder();
		for (int i = subgroups.size() - 1; 0 <= i; i--)
			builder.append(subgroups.get(i).getDrawingCommands());
		return builder.toString();
	}

	/**
	 * Returns the smallest extent that contains all of the shapes contained directly or indirectly by this shape group.
	 *
	 * @inspects | this, ...getAllShapes()
	 * @post | result != null
	 * @post | result.getLeft() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getX
()).min().getAsInt()
	 * @post | result.getTop() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.getY(
)).min().getAsInt()
	 * @post | result.getRight() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.get
X()).max().getAsInt()
	 * @post | result.getBottom() == getAllShapes().stream().flatMap(s -> Arrays.stream(s.getVertices())).mapToInt(p -> p.ge
tY()).max().getAsInt()
	 */
	@Override
	public Extent getBoundingBox() {
		int minX = Integer.MAX_VALUE;
		int maxX = Integer.MIN_VALUE;
		int minY = Integer.MAX_VALUE;
		int maxY = Integer.MIN_VALUE;
		for (ShapeGroup subgroup : subgroups) {
			Extent boundingBox = subgroup.getBoundingBox();
			minX = Math.min(minX, boundingBox.getLeft());
			maxX = Math.max(maxX, boundingBox.getRight());
			minY = Math.min(minY, boundingBox.getTop());
			maxY = Math.max(maxY, boundingBox.getBottom());
		}
		return Extent.ofLeftTopRightBottom(minX, minY, maxX, maxY);
	}

	/**
	 * Return the first subgroup in this non-leaf shape group's list of subgroups whose
	 * bounding box contains the given point.
	 *
	 * @throws IllegalArgumentException if {@code point} is null
	 *     | point == null
	 * @post
	 *     | Objects.equals(result,
	 *     |     getSubgroups().stream().filter(g -> g.getBoundingBox().contains(point))
	 *     |         .findFirst().orElse(null))
	 */
	public ShapeGroup getSubgroupAt(IntPoint point) {
		if (point == null)
			throw new IllegalArgumentException("point is null");

		for (ShapeGroup group : subgroups)
			if (group.getBoundingBox().contains(point))
				return group;
		return null;
	}

	/**
	 * {@inheritDoc}
	 *
	 * @post | Objects.equals(result, getSubgroups().stream().flatMap(g -> g.getAllShapes().stream()).collect(Collectors.toL
ist()))
	 */
	public List<RoundedPolygon> getAllShapes() {
		return subgroups.stream().flatMap(subgroup -> subgroup.getAllShapes().stream()).collect(Collectors.toList());
	}

	/**
	 * Initializes this object to represent a non-leaf shape group that directly contains the given
	 * subgroups, in the given order.
	 *
	 * @mutates | this
	 * @mutates_properties | (...subgroups).getParentGroup()
	 * @inspects | subgroups
	 *
	 * @throws IllegalArgumentException if {@code subgroups} is null
	 *     | subgroups == null
	 * @throws IllegalArgumentException if {@code subgroups} has less than two elements
	 *     | subgroups.length < 2
	 * @throws IllegalArgumentException if any element of {@code subgroups} is null
	 *     | Arrays.stream(subgroups).anyMatch(g -> g == null)
	 * @throws IllegalArgumentException if the given subgroups are not distinct
	 *     | !LogicalList.distinct(List.of(subgroups))
	 * @throws IllegalArgumentException if any of the given subgroups already has a parent
	 *     | Arrays.stream(subgroups).anyMatch(g -> g.getParentGroup() != null)
	 *
	 * @post | Objects.equals(getSubgroups(), List.of(subgroups))
	 * @post | Arrays.stream(subgroups).allMatch(g -> g.getParentGroup() == this)
	 * @post | getParentGroup() == null
	 */
	public NonleafShapeGroup(ShapeGroup[] subgroups) {
		if (subgroups == null)
			throw new IllegalArgumentException("subgroups is null");
		if (subgroups.length < 2)
			throw new IllegalArgumentException("subgroups has less than two elements");
		if (Arrays.stream(subgroups).anyMatch(g -> g == null))
			throw new IllegalArgumentException("subgroups has null elements");
		if (!LogicalList.distinct(List.of(subgroups)))
			throw new IllegalArgumentException("subgroups has duplicate elements");
		if (Arrays.stream(subgroups).anyMatch(g -> g.getParentGroup() != null))
			throw new IllegalArgumentException("some of the given groups already have a parent");

		this.subgroups = new ArrayList<>(Arrays.asList(subgroups));
		for (ShapeGroup group : subgroups) {
			assert group.getParentGroup() == null;
			group.parent = this;
		}
	}

}
```

```java
package drawit.shapegroups1.exporter;

import java.awt.Color;
import java.util.Arrays;
import java.util.Map;
import java.util.stream.Collectors;

import drawit.IntPoint;
import drawit.RoundedPolygon;
import drawit.shapegroups1.LeafShapeGroup;
import drawit.shapegroups1.NonleafShapeGroup;
import drawit.shapegroups1.ShapeGroup;

public class ShapeGroupExporter {

	public static Object toPlainData(IntPoint point) {
		return Map.of("x", point.getX(), "y", point.getY());
	}

	public static Object toPlainData(Color color) {
		return Map.of("red", color.getRed(), "green", color.getGreen(), "blue", color.getBlue());
	}

	public static Object toPlainData(RoundedPolygon polygon) {
		return Map.of(
				"vertices", Arrays.stream(polygon.getVertices()).map(p -> toPlainData(p)).collect(Collectors.toL
ist()),
				"radius", polygon.getRadius(),
				"color", toPlainData(polygon.getColor()));
	}

	public static Object toPlainData(ShapeGroup shapeGroup) {
		if (shapeGroup instanceof LeafShapeGroup)
			return Map.of("shape", toPlainData(((LeafShapeGroup)shapeGroup).getShape()));
		else
			return Map.of("subgroups", ((NonleafShapeGroup)shapeGroup).getSubgroups().stream().map(g -> toPlainData(
g)).collect(Collectors.toList()));
	}

}
```

```java
package drawit.tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import drawit.IntPoint;
import drawit.IntVector;

class IntPointTest {

	IntPoint p = new IntPoint(5, 7);

	@Test
	void testConstructorAndGetters() {
		assert 5 == p.getX();
		assert 7 == p.getY();
	}

	@Test
	void testAsDoublePoint() {
		assert 5.0 == p.asDoublePoint().getX();
		assert 7.0 == p.asDoublePoint().getY();
	}

	@Test
	void testEquals() {
		assert p.equals(new IntPoint(5, 7));
		assert !p.equals(new IntPoint(7, 5));
	}

	@Test
	void testIsOnLineSegment() {
		assert p.isOnLineSegment(new IntPoint(5, 3), new IntPoint(5, 9));
		assert p.isOnLineSegment(new IntPoint(2, 4), new IntPoint(8, 10));
		assert p.isOnLineSegment(new IntPoint(8, 4), new IntPoint(2, 10));
		assert !p.isOnLineSegment(new IntPoint(7, 4), new IntPoint(1, 10));
	}

	@Test
	void testMinus() {
		IntVector v = p.minus(new IntPoint(6, 6));
		assert v.getX() == -1;
		assert v.getY() == 1;
		assert p.getX() == 5;
		assert p.getY() == 7;
	}

	@Test
	void testPlus() {
		assert new IntPoint(55, 77).equals(p.plus(new IntVector(50, 70)));
		assert p.getX() == 5;
		assert p.getY() == 7;
	}

	@Test
	void testLineSegmentsIntersect() {
		IntPoint q = new IntPoint(7, 5);
		IntPoint r = new IntPoint(5, 5);
		IntPoint s = new IntPoint(7, 7);
		assert IntPoint.lineSegmentsIntersect(p, q, r, s);
		assert IntPoint.lineSegmentsIntersect(s, r, q, p);
		assert IntPoint.lineSegmentsIntersect(r, s, p, q);
		assert !IntPoint.lineSegmentsIntersect(p, r, q, s);
	}

}
```