Data och Informationsteknik / Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Erland Holmström Göteborg 14 dec 2012

Examination in

PROGRAMMERINGSTEKNIK F1 TIN211

DAY: Monday DATE: 2012-12-17 TIME: 8.30-13.30 (OBS 5 tim) ROOM: M

Responsible teacher:	Erland Holmström tel. 1007, home 0708-710 600
Results:	Are sent by mail from Ladok.
Solutions:	Are eventually posted on homepage.
Inspection of grading:	The exam can be found in our study expedition after posting of results.
	Time for complaints about grading are announced on homepage after the result are published or mail me and we find a time.
Grade limits:	CTH: 3=26p, 4=36p, 5= 46p, max 60p
Aids on the exam:	**Bravaco, Simonson: Java Programming From the Grounf Up**

Observe:

- Start by reading through all questions so you can ask questions when I come. I usually will come after appr. 2 hours.
- All answers must be motivated when applicable.
- Write legible! Draw figures. Solutions that are difficult to read are not evaluated!
- Answer concisely and to the point.
- The advice and directions given during course must be followed.
- Programs should be written in Java, indent properly, use comments and so on.
- Start every new problem on a new sheet of paper.

Good Luck!

Problem 1. Sant eller falskt? Motivera. Eller svara på frågorna.

a) En underklass ärver alla metoder från sin superklass utom konstruktorer.

b) Även statiska metoder kan överskuggas.

c) Interfacet Comparable ser ut som nedan. Hur måste man ändra en klass, tex Polygon nedan. för att den skall implementera interfacet?

```
interface Comparable {
  public int compareTo(Object o)
}
```

d) Finns det några positiva indata (≥ 0) till funktionen nedan som gör att den inte terminerar?

```
public static int f(int n) {
    if(n==0) {
        return 1;
    } else if (n==1) {
        return 2;
    } else {
        return f(n-1) * f(n-3);
    }
}
```

e) Antag att vi har följande klasser

```
public class Mammal { ... }
public class Dog extends Mammal { ... }
public class Cat extends Mammal { ... }
public class TestMammal {
    public static void main(String[] args) { ...
```

och att vi i main deklarerar följande variabler

```
Mammal m;
Dog d = new Dog();
Cat c = new Cat();
```

Vilka av följande satser är ok, vilka kommer att orsaka kompileringsfel och vilka kan orsaka exekveringsfel och varför?

```
m = d;          // 1
d = m;          // 2
d = (Dog)m;     // 3
d = c;          // 4
d = (Dog)c;     // 5
m = (Mammal)d;  // 6
```

(8p)

Problem 2. *Testar: operatorer, lab 2, loopar, villkor.*

Skriv ett program som läser ett tal j från användaren och hittar det närmaste primtalet till j (förutom j självt om j är primtal). Avståndet till ett primtal från j är abs(j-primtalet). Finns det två närmsta primtal så skall bägge skrivas ut. Primtalstesten skall ligga i ett underprogram `isPrime(n)`.

Programmet skall vara effektivt tex primtalstesten får inte vara onödigt ineffektiv.

Du behöver inte felhantera indata.

Exempel på 3 körningar:

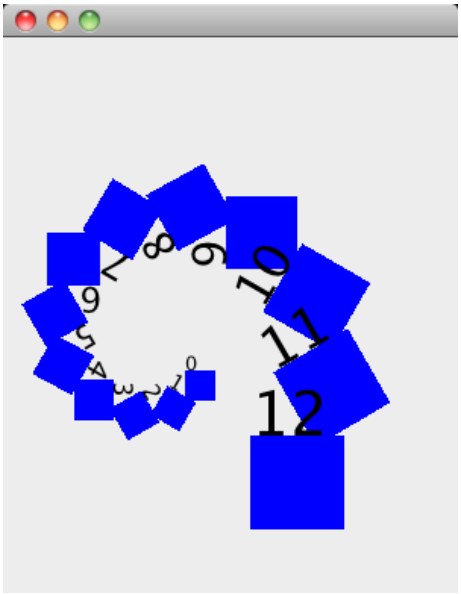
```
> ange ett heltal:
5
närmsta primtalet är 3 på avståndet 2
närmsta primtalet är 7 på avståndet 2

> ange ett heltal:
7
närmsta primtalet är 5 på avståndet 2

> ange ett heltal:
26
närmsta primtalet är 23 på avståndet 3
närmsta primtalet är 29 på avståndet 3
```

(10p)

Problem 3. *Testar: grafik, att läsa Javadoc.*



Skriv ett program som ritar figuren till höger.

När man får ett Graphics objekt som parameter till paintComponent så är Graphics den statiska typen. Den dynamiska typen för objektet man får är Graphics2D (som ärver Graphics). Utnyttja detta. Utdrag ur Javadoc för Graphics2D i slutet.

Rektanglarna är blå, texten svart, fonten skall vara SansSerif, PLAIN och 15 pixlar. Ritytan är 400x400 och den innersta figuren börjar ungefär i 100,200.

Figuren behöver inte ändra storlek när man ändrar fönsterstorlek.

Speciellt användbara metoder i Graphics2D

(origo är från början i övre vänstra hörnet som vanligt)

setColor, fillRect (fill2dRect), drawString

translate(dx, dy) - Flyttar origo dx, dy.

rotate(angle in radians) - roterar efterföljande ritningar i förhållande till origo.

scale(sx, sy) - Justerar storleken på alla följande ritade objekt. 1.0 är samma storlek.

(12p)

Problem 4. Testar: metoder, fält. loopar, matriser, olika typer av felhantering, exceptions, inläsning.

Pascals triangel är välkänd för dom flesta. En rad i triangeln bildas genom att man tar summan av de två talen ovanför. De första 5 raderna ser ut som figuren till vänster:

```
    1              1
   1 1            1 1
  1 2 1          1 2 1
 1 3 3 1        1 3 3 1
1 4 6 4 1      1 4 6 4 1
```

Nu skall du skapa, lagra i en matris och skriva ut, de n första raderna av Pascals triangel där n anges på kommandoraden. Matrisen skall ta så liten plats som möjligt så du lagrar värdena som i figuren till höger. Respektive rad i matrisen får inte vara längre än nödvändigt dvs matrisens första rad är bara ett element lång, den andra är 2 element lång, tredje 3 element lång osv. Denna typ av java matris kallas ”jagged” (jagged=ojämn, sågtandad), se not 1.

Du får olika många poäng (och olika maxpoäng) beroende på vilken variant av deluppgifterna nedan du löser. *Ange vad du väljer tex ”a) alt1: ”.* I alla deluppgifter kan du anta att dom andra finns tillgängliga

a) Skriv först en metod, `printx(...)`, som skriver ut en ojämn matris som den får som parameter. Den skall skriva ett mellanslag mellan talen och skall avsluta med en tom rad.
   Du skriver **en** av de två följande versionerna: (se även not 2 nedan)
   - Alt 1: Metoden, `print1`, skriver ut enligt figuren till höger ovan. (2p)
   - Alt 2: Metoden, `print2`, skriver ut enligt figuren till vänster ovan. (3p)

b) Matrisen skapas och returneras av en metod `fillInPascal` som bara tar antalet rader i triangeln som parameter. (5p)

c) Nu skall vi skriva ett huvudprogram (main). Det skall läsa n från *kommandoraden*, skapa en matris med n rader och skriva ut den med hjälp av metoderna ovan.
   Skriv **en** av metoderna nedan, ange vilket alternativ du gör. Fel resulterar i en felutskrift som beskriver problemet (olika för olika fel) och programmet avslutas på lämpligt sätt. (Du får fånga exceptions om du vill men inte kasta.)
   - Alt.1: ingen felhantering behövs (2p)
   - Alt.2: n skall vara mellan 2 och 20. (3p)(
   - Alt.3: som ovan + n skall vara heltal. (5p)
   - Alt.4: som ovan + inget argument givet på kommandoraden hanteras. (6p)

Not 1: Man kan konstruera en ojämn matris såhär

```
// ger en vektor med referenser till int[] vektorer
int[][] triangle = new int[antalet rader][];
// skapar vektor 0 som ett heltalsfält med längd 2
triangle[0] = new int[2];  ... osv.
```

Not 2: Det räcker med ett mellanslag mellan varje tal dvs du behöver inte ta hänsyn till att talen blir större än 10 från 6e raden utan vi accepterar att matrisen bli ”sned” då.

```
 1 4 6 4 1
1 5 10 10 5 1
```

(max 14p)

Problem 5. Testar: Programmera klasser, läsa javadoc, felhantering, fält.

Javadoc dokumentation för klassen Polygon finns i slutet. Implementera klassen enligt alla konstens regler.

Du skall implementera följande (se Javadoc'en)

- datatyper - speciellt: använd två fält för lagringen. Det är också ok att använda en arrayList (men du får inte ändra i specifikationen)

- `Polygon()`

- `Polygon(int[] xpoints, int[] ypoints, int npoints)` - Det är 4 unika möjliga fel på indatat som behöver hanteras om man inte räknar att en del av dom skall göras på bägge fälten, kasta en `IndexOutOfBoundsException`.

- `addPoint` – observera att fältet inte skall bli fullt, det måste du lösa genom att förstora det.

- `int[] getXPoints()`

- `reset`

- `boolean contains(int x, int y)` och `boolean contains(Point p)`

Du kan anta att `boolean contains(double x, double y)` redan är skriven (kod finns i slutet).

Du behöver *inte* skriva resten av metoderna och inte ha med variabeln bounds och behöver inte implementera Shape.

När man kör följande testprogram så får man (den korrekta) utskriften nedan:

```
public class TestPolygon {
  public static void main(String[] args) {
    Polygon p = new Polygon(); // ny polygon med 3 platser
    p.addPoint(3,5);           // lägg in 4 punkter
    p.addPoint(4,6);
    p.addPoint(5,7);
    p.addPoint(6,8);

    p.translate(2,3); // flytta alla punkter

    int[] xp = p.getXPoints(); // hämta x-kordinater
    int[] yp = p.getYPoints();
    System.out.println(p.getNPoints());
    for(int i = 0; i<p.getNPoints(); i++) {
      System.out.println(xp[i] + " " + yp[i]);
    }
  }
} // end TestPolygon
/*
4
5 8
6 9
7 10
8 11
*/
```

(16p)

```
/**Determines if the coordinates are inside this Polyg.*/
  public boolean contains(double x, double y) {
    if (npoints <= 2) { return false; }
    int hits = 0;
    int lastx = xpoints[npoints – 1];
    int lasty = ypoints[npoints - 1];
    int curx, cury;
    // Walk the edges of the polygon
    for (int i = 0; i < npoints;
                        lastx = curx, lasty = cury, i++) {
      curx = xpoints[i];   cury = ypoints[i];
      if (cury == lasty) { continue; }
      int leftx;
      if (curx < lastx) {
        if (x >= lastx) { continue; }
        leftx = curx;
      } else {
        if (x >= curx) { continue; }
        leftx = lastx;
      }
      double test1, test2;
      if (cury < lasty) {
        if (y < cury || y >= lasty) {
          continue;
        }
        if (x < leftx) {
          hits++;
          continue;
        }
        test1 = x – curx;   test2 = y - cury;
      } else {
        if (y < lasty || y >= cury) {
          continue;
        }
        if (x < leftx) {
          hits++;
          continue;
        }
        test1 = x - lastx;
        test2 = y - lasty;
      }
      if (test1 < (test2 / (lasty - cury) * (lastx - curx))) {
        hits++;
      }
    }
    return ((hits & 1) != 0);
  }
```

java.awt

Class Polygon

java.lang.Object
  └ **java.awt.Polygon**

All Implemented Interfaces:

Shape

```
public class Polygon
extends Object
implements Shape
```

The `Polygon` class encapsulates a description of a closed, two-dimensional region within a coordinate space. This region is bounded by an arbitrary number of line segments, each of which is one side of the polygon. Internally, a polygon comprises of a list of (*x*, *y*) coordinate pairs, where each pair defines a *vertex* of the polygon, and two successive pairs are the endpoints of a (imaginary) line that is a side of the polygon. The first and final pairs of (*x*, *y*) points are joined by a (imaginary) line segment that closes the polygon.

Since:

JDK1.0

Field Summary

protected Rectangle	**bounds** Bounds of the polygon.
int	**npoints** The total number of points.
int[]	**xpoints** The array of *x* coordinates.
int[]	**ypoints** The array of *y* coordinates.

Constructor Summary

Polygon() Creates an empty polygon with default size = 3.
Polygon(int[] xpoints, int[] ypoints, int npoints) Constructs and initializes a `Polygon` from the specified parameters.

Method Summary

void	**addPoint**(int x, int y) Appends the specified coordinates to this `Polygon`.
boolean	**contains**(double x, double y) Determines if the specified coordinates are inside this `Polygon`.
boolean	**contains**(int x, int y) Determines whether the specified coordinates are inside this `Polygon`.
boolean	**contains**(Point p) Determines whether the specified `Point` is inside this `Polygon`.
Rectangle	**getBounds**() Gets the bounding box of this Polygon.
int	**getNPoints**() Returns the total number of points.
int[]	**getXPoints**() Returns a copy of the array of y coordinates.
int[]	**getYPoints**() Returns a copy of the array of y coordinates.
void	**reset**() Resets this `Polygon` object to an empty polygon.
void	**translate**(int deltaX, int deltaY) Translates the vertices of the `Polygon` by `deltaX` along the x axis and by `deltaY` along the y axis.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

npoints

```
private int npoints
```

The total number of points. The value of `npoints` represents the number of valid points in this `Polygon` and might be less than the number of elements in `xpoints` or `ypoints`.

See Also:

`addPoint(int, int)`

---

xpoints

```
private int[] xpoints
```

The array of *x* coordinates. The number of elements in this array might be more than the number of *x* coordinates in this `Polygon`. The extra elements allow new points to be added to this `Polygon` without re-creating this array. The value of `npoints` is equal to the number of valid points in this `Polygon`.

See Also:
addPoint(int, int)

---

ypoints

```
private int[] ypoints
```

The array of *y* coordinates. The number of elements in this array might be more than the number of *y* coordinates in this `Polygon`. The extra elements allow new points to be added to this `Polygon` without re-creating this array. The value of `npoints` is equal to the number of valid points in this `Polygon`.

See Also:
addPoint(int, int)

---

bounds

```
private Rectangle bounds
```

Bounds of the polygon. This value can be NULL. Please see the javadoc comments getBounds().

See Also:
getBounds()

Constructor Detail

Polygon

```
public Polygon()
```

Creates an empty polygon with default size = 3.

Polygon

```
public Polygon(int[] xpoints,
               int[] ypoints,
               int npoints)
```

Constructs and initializes a `Polygon` from the specified parameters.

Parameters:
`xpoints` - an array of *x* coordinates
`ypoints` - an array of *y* coordinates
`npoints` - the total number of points in the `Polygon`
Throws:
NegativeArraySizeException - if the value of `npoints` is negative.
IndexOutOfBoundsException - if `npoints` is greater than the length of `xpoints` or the length of `ypoints`.
NullPointerException - if `xpoints` or `ypoints` is `null`.

Method Detail

---

addPoint

```
public void addPoint(int x,
                     int y)
```

Appends the specified coordinates to this `Polygon`.

If an operation that calculates the bounding box of this `Polygon` has already been performed, such as `getBounds` or `contains`, then this method updates the bounding box.

Parameters:
`x` - the specified x coordinate
`y` - the specified y coordinate
See Also:
getBounds(), contains(java.awt.Point)

---

contains

```
public boolean contains(Point p)
```

Determines whether the specified Point is inside this `Polygon`.

Parameters:
`p` - the specified `Point` to be tested
Returns:
`true` if the `Polygon` contains the `Point`; `false` otherwise.
See Also:
contains(double, double)

---

contains

```
public boolean contains(int x,
                        int y)
```

Determines whether the specified coordinates are inside this `Polygon`.

Parameters:
`x` - the specified x coordinate to be tested
`y` - the specified y coordinate to be tested
Returns:
`true` if this `Polygon` contains the specified coordinates, (*x*, *y*); `false` otherwise.
Since:
JDK1.1
See Also:
contains(double, double)

---

contains

```
public boolean contains(double x,
                        double y)
```

Determines if the specified coordinates are inside this `Polygon`. For the definition of *insideness*, see the class comments of `Shape`.

Specified by:
`contains` in interface `Shape`

Parameters:
`x` - the specified x coordinate
`y` - the specified y coordinate

Returns:
`true` if the `Polygon` contains the specified coordinates; `false` otherwise.

---

getBounds

```
public Rectangle getBounds()
```

Gets the bounding box of this `Polygon`. The bounding box is the smallest `Rectangle` whose sides are parallel to the x and y axes of the coordinate space, and can completely contain the `Polygon`.

Specified by:
`getBounds` in interface `Shape`

Returns:
a `Rectangle` that defines the bounds of this `Polygon`.

Since:
JDK1.1

See Also:
`Shape.getBounds2D()`

---

getNPoints

```
public int getNPoints()
```

Returns the total number of points.

Returns:
Returns the total number of points.

---

getXPoints

```
public int[] getXPoints()
```

Returns a copy of the array of x coordinates.

Returns:
Returns a copy of the array of x coordinates.

---

getYPoints

```
public int[] getYPoints()
```

Returns a copy of the array of y coordinates.

Returns:
Returns a copy of the array of y coordinates.

---

reset

```
public void reset()
```

Resets this `Polygon` object to an empty polygon. The coordinate arrays and the data in them are left untouched but the number of points is reset to zero to mark the old vertex data as invalid and to start accumulating new vertex data at the beginning. All internally-cached data relating to the old vertices are discarded. Note that since the coordinate arrays from before the reset are reused, creating a new empty `Polygon` might be more memory efficient than resetting the current one if the number of vertices in the new polygon data is significantly smaller than the number of vertices in the data from before the reset.

Since:
1.4

---

translate

```
public void translate(int deltaX,
                      int deltaY)
```

Translates the vertices of the `Polygon` by `deltaX` along the x axis and by `deltaY` along the y axis.

Parameters:
`deltaX` - the amount to translate along the *x* axis
`deltaY` - the amount to translate along the *y* axis

Since:
JDK1.1

---

java.awt

Class Graphics2D

java.lang.Object
  └ java.awt.Graphics
    └ **java.awt.Graphics2D**

public abstract class **Graphics2D**
extends Graphics

This Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text and images on the Java(tm) platform.

Coordinate Spaces

All coordinates passed to a Graphics2D object are specified in a device-independent coordinate system called User Space, which is used by applications. The Graphics2D object contains an AffineTransform object as part of its rendering state that defines how to convert coordinates from user space to device-dependent coordinates in Device Space.

Coordinates in device space usually refer to individual device pixels and are aligned on the infinitely thin gaps between these pixels. Some Graphics2D objects can be used to capture rendering operations for storage into a graphics metafile for playback on a concrete device of unknown physical resolution at a later time. Since the resolution might not be known when the rendering operations are captured, the Graphics2D Transform is set up to transform user coordinates to a virtual device space that approximates the expected resolution of the target device. Further transformations might need to be applied at playback time if the estimate is incorrect.

Some of the operations performed by the rendering attribute objects occur in the device space, but all Graphics2D methods take user space coordinates.

Every Graphics2D object is associated with a target that defines where rendering takes place. A GraphicsConfiguration object defines the characteristics of the rendering target, such as pixel format and resolution. The same rendering target is used throughout the life of a Graphics2D object.

When creating a Graphics2D object, the GraphicsConfiguration specifies the default transform for the target of the Graphics2D (a Component or Image). This default transform maps the user space coordinate system to screen and printer device coordinates such that the origin maps to the upper left hand corner of the target region of the device with increasing X coordinates extending to the right and increasing Y coordinates extending downward. The scaling of the default transform is set to identity for those devices that are close to 72 dpi, such as screen devices. The scaling of the default transform is set to approximately 72 user space coordinates per square inch for high resolution devices, such as printers. For image buffers, the default transform is the Identity transform.

Rendering Process

The Rendering Process can be broken down into four phases that are controlled by the Graphics2D rendering attributes. The renderer can optimize many of these steps, either by caching the results for future calls, by collapsing multiple virtual steps into a single operation, or by recognizing various attributes as common simple cases that can be eliminated by modifying other parts of the operation.

The steps in the rendering process are:

1. Determine what to render.
2. Constrain the rendering operation to the current Clip. The Clip is specified by a Shape in user space and is controlled by the program using the various clip manipulation methods of Graphics and Graphics2D. This *user clip* is transformed into device space by the current Transform and combined with the *device clip*, which is defined by the visibility of windows and device extents. The combination of the user clip and device clip defines the *composite clip*, which determines the final clipping region. The user clip is not modified by the rendering system to reflect the resulting composite clip.
3. Determine what colors to render.
4. Apply the colors to the destination drawing surface using the current Composite attribute in the Graphics2D context.

The three types of rendering operations, along with details of each of their particular rendering processes are:

1. **Shape operations**
    1. If the operation is a draw(Shape) operation, then the createStrokedShape method on the current Stroke attribute in the Graphics2D context is used to construct a new Shape object that contains the outline of the specified Shape.
    2. The Shape is transformed from user space to device space using the current Transform in the Graphics2D context.
    3. The outline of the Shape is extracted using the getPathIterator method of Shape, which returns a PathIterator object that iterates along the boundary of the Shape.
    4. If the Graphics2D object cannot handle the curved segments that the PathIterator object returns then it can call the alternate getPathIterator method of Shape, which flattens the Shape.
    5. The current Paint in the Graphics2D context is queried for a PaintContext, which specifies the colors to render in device space.
2. **Text operations**
    1. The following steps are used to determine the set of glyphs required to render the indicated String:
        1. If the argument is a String, then the current Font in the Graphics2D context is asked to convert the Unicode characters in the String into a set of glyphs for presentation with whatever basic layout and shaping algorithms the font implements.
        2. If the argument is an AttributedCharacterIterator, the iterator is asked to convert itself to a TextLayout using its embedded font attributes. The TextLayout implements more sophisticated glyph layout algorithms that perform Unicode bi-directional layout adjustments automatically for multiple fonts of differing writing directions.
        3. If the argument is a GlyphVector, then the GlyphVector object already contains the appropriate font-specific glyph codes with explicit coordinates for the position of each glyph.
    2. The current Font is queried to obtain outlines for the indicated glyphs. These outlines

are treated as shapes in user space relative to the position of each glyph that was determined in step 1.
3. The character outlines are filled as indicated above under Shape operations.
4. The current `Paint` is queried for a `PaintContext`, which specifies the colors to render in device space.

3. **`Image` Operations**
   1. The region of interest is defined by the bounding box of the source `Image`. This bounding box is specified in Image Space, which is the `Image` object's local coordinate system.
   2. If an `AffineTransform` is passed to `drawImage(Image, AffineTransform, ImageObserver)`, the `AffineTransform` is used to transform the bounding box from image space to user space. If no `AffineTransform` is supplied, the bounding box is treated as if it is already in user space.
   3. The bounding box of the source `Image` is transformed from user space into device space using the current `Transform`. Note that the result of transforming the bounding box does not necessarily result in a rectangular region in device space.
   4. The `Image` object determines what colors to render, sampled according to the source to destination coordinate mapping specified by the current `Transform` and the optional image transform.

Default Rendering Attributes

The default values for the `Graphics2D` rendering attributes are:

`Paint`
: The color of the `Component`.

`Font`
: The `Font` of the `Component`.

`Stroke`
: A square pen with a linewidth of 1, no dashing, miter segment joins and square end caps.

`Transform`
: The `getDefaultTransform` for the `GraphicsConfiguration` of the `Component`.

`Composite`
: The `AlphaComposite.SRC_OVER` rule.

`Clip`
: No rendering `Clip`, the output is clipped to the `Component`.

Rendering Compatibility Issues

The JDK(tm) 1.1 rendering model is based on a pixelization model that specifies that coordinates are infinitely thin, lying between the pixels. Drawing operations are performed using a one-pixel wide pen that fills the pixel below and to the right of the anchor point on the path. The JDK 1.1 rendering model is consistent with the capabilities of most of the existing class of platform renderers that need to resolve integer coordinates to a discrete pen that must fall completely on a specified number of pixels.

The Java 2D(tm) (Java(tm) 2 platform) API supports antialiasing renderers. A pen with a width of one pixel does not need to fall completely on pixel N as opposed to pixel N+1. The pen can fall partially on both pixels. It is not necessary to choose a bias direction for a wide pen since the blending that occurs along the pen traversal edges makes the sub-pixel position of the pen visible to the user. On the other hand, when antialiasing is turned off by setting the `KEY_ANTIALIASING` hint

key to the `VALUE_ANTIALIAS_OFF` hint value, the renderer might need to apply a bias to determine which pixel to modify when the pen is straddling a pixel boundary, such as when it is drawn along an integer coordinate in device space. While the capabilities of an antialiasing renderer make it no longer necessary for the rendering model to specify a bias for the pen, it is desirable for the antialiasing and non-antialiasing renderers to perform similarly for the common cases of drawing one-pixel wide horizontal and vertical lines on the screen. To ensure that turning on antialiasing by setting the `KEY_ANTIALIASING` hint key to `VALUE_ANTIALIAS_ON` does not cause such lines to suddenly become twice as wide and half as opaque, it is desirable to have the model specify a path for such lines so that they completely cover a particular set of pixels to help increase their crispness.

Java 2D API maintains compatibility with JDK 1.1 rendering behavior, such that legacy operations and existing renderer behavior is unchanged under Java 2D API. Legacy methods that map onto general `draw` and `fill` methods are defined, which clearly indicates how `Graphics2D` extends `Graphics` based on settings of `Stroke` and `Transform` attributes and rendering hints. The definition performs identically under default attribute settings. For example, the default `Stroke` is a `BasicStroke` with a width of 1 and no dashing and the default Transform for screen drawing is an Identity transform.

The following two rules provide predictable rendering behavior whether aliasing or antialiasing is being used.

- Device coordinates are defined to be between device pixels which avoids any inconsistent results between aliased and antaliased rendering. If coordinates were defined to be at a pixel's center, some of the pixels covered by a shape, such as a rectangle, would only be half covered. With aliased rendering, the half covered pixels would either be rendered inside the shape or outside the shape. With anti-aliased rendering, the pixels on the entire edge of the shape would be half covered. On the other hand, since coordinates are defined to be between pixels, a shape like a rectangle would have no half covered pixels, whether or not it is rendered using antialiasing.
- Lines and paths stroked using the `BasicStroke` object may be "normalized" to provide consistent rendering of the outlines when positioned at various points on the drawable and whether drawn with aliased or antialiased rendering. This normalization process is controlled by the `KEY_STROKE_CONTROL` hint. The exact normalization algorithm is not specified, but the goals of this normalization are to ensure that lines are rendered with consistent visual appearance regardless of how they fall on the pixel grid and to promote more solid horizontal and vertical lines in antialiased mode so that they resemble their non-antialiased counterparts more closely. A typical normalization step might promote antialiased line endpoints to pixel centers to reduce the amount of blending or adjust the subpixel positioning of non-antialiased lines so that the floating point line widths round to even or odd pixel counts with equal likelihood. This process can move endpoints by up to half a pixel (usually towards positive infinity along both axes) to promote these consistent results.

The following definitions of general legacy methods perform identically to previously specified behavior under default attribute settings:

- For `fill` operations, including `fillRect`, `fillRoundRect`, `fillOval`, `fillArc`, `fillPolygon`, and `clearRect`, `fill` can now be called with the desired `Shape`. For example, when filling a rectangle:

  ```
  fill(new Rectangle(x, y, w, h));
  ```

  is called.

- Similarly, for draw operations, including `drawLine`, `drawRect`, `drawRoundRect`, `drawOval`, `drawArc`, `drawPolyline`, and `drawPolygon`, `draw` can now be called with the desired `Shape`. For example, when drawing a rectangle:

```
draw(new Rectangle(x, y, w, h));
```

  is called.

- The `draw3DRect` and `fill3DRect` methods were implemented in terms of the `drawLine` and `fillRect` methods in the `Graphics` class which would predicate their behavior upon the current `Stroke` and `Paint` objects in a `Graphics2D` context. This class overrides those implementations with versions that use the current `Color` exclusively, overriding the current `Paint` and which uses `fillRect` to describe the exact same behavior as the preexisting methods regardless of the setting of the current `Stroke`.

The `Graphics` class defines only the `setColor` method to control the color to be painted. Since the Java 2D API extends the `Color` object to implement the new `Paint` interface, the existing `setColor` method is now a convenience method for setting the current `Paint` attribute to a `Color` object. `setColor(c)` is equivalent to `setPaint(c)`.

The `Graphics` class defines two methods for controlling how colors are applied to the destination.

1. The `setPaintMode` method is implemented as a convenience method to set the default `Composite`, equivalent to `setComposite(new AlphaComposite.SrcOver)`.
2. The `setXORMode(Color xorcolor)` method is implemented as a convenience method to set a special `Composite` object that ignores the `Alpha` components of source colors and sets the destination color to the value:

```
dstpixel = (PixelOf(srccolor) ^ PixelOf(xorcolor) ^ dstpixel);
```

See Also:

`RenderingHints`

Constructor Summary

protected	**Graphics2D**() Constructs a new `Graphics2D` object.

Method Summary

abstract void	**addRenderingHints**(Map<?,?> hints) Sets the values of an arbitrary number of preferences for the rendering algorithms.
abstract void	**clip**(Shape s) Intersects the current `Clip` with the interior of the specified `Shape` and sets the `Clip` to the resulting intersection.
abstract void	**draw**(Shape s) Strokes the outline of a `Shape` using the settings of the current `Graphics2D` context.

void	**draw3DRect**(int x, int y, int width, int height, boolean raised) Draws a 3-D highlighted outline of the specified rectangle.
abstract void	**drawGlyphVector**(GlyphVector g, float x, float y) Renders the text of the specified `GlyphVector` using the `Graphics2D` context's rendering attributes.
abstract void	**drawImage**(BufferedImage img, BufferedImageOp op, int x, int y) Renders a `BufferedImage` that is filtered with a `BufferedImageOp`.
abstract boolean	**drawImage**(Image img, AffineTransform xform, ImageObserver obs) Renders an image, applying a transform from image space into user space before drawing.
abstract void	**drawRenderableImage**(RenderableImage img, AffineTransform xform) Renders a `RenderableImage`, applying a transform from image space into user space before drawing.
abstract void	**drawRenderedImage**(RenderedImage img, AffineTransform xform) Renders a `RenderedImage`, applying a transform from image space into user space before drawing.
abstract void	**drawString**(AttributedCharacterIterator iterator, float x, float y) Renders the text of the specified iterator, using the `Graphics2D` context's current `Paint`.
abstract void	**drawString**(AttributedCharacterIterator iterator, int x, int y) Renders the text of the specified iterator, using the `Graphics2D` context's current `Paint`.
abstract void	**drawString**(String s, float x, float y) Renders the text specified by the specified `String`, using the current text attribute state in the `Graphics2D` context.
abstract void	**drawString**(String str, int x, int y) Renders the text of the specified `String`, using the current text attribute state in the `Graphics2D` context.
abstract void	**fill**(Shape s) Fills the interior of a `Shape` using the settings of the `Graphics2D` context.
void	**fill3DRect**(int x, int y, int width, int height, boolean raised) Paints a 3-D highlighted rectangle filled with the current color.
abstract Color	**getBackground**() Returns the background color used for clearing a region.
abstract Composite	**getComposite**() Returns the current `Composite` in the `Graphics2D` context.
abstract GraphicsConfiguration	**getDeviceConfiguration**() Returns the device configuration associated with this `Graphics2D`.
abstract FontRenderContext	**getFontRenderContext**() Get the rendering context of the `Font` within this `Graphics2D`

	context.
abstract Paint	**getPaint**() Returns the current Paint of the Graphics2D context.
abstract Object	**getRenderingHint**(RenderingHints.Key hintKey) Returns the value of a single preference for the rendering algorithms.
abstract RenderingHints	**getRenderingHints**() Gets the preferences for the rendering algorithms.
abstract Stroke	**getStroke**() Returns the current Stroke in the Graphics2D context.
abstract AffineTransform	**getTransform**() Returns a copy of the current Transform in the Graphics2D context.
abstract boolean	**hit**(Rectangle rect, Shape s, boolean onStroke) Checks whether or not the specified Shape intersects the specified Rectangle, which is in device space.
abstract void	**rotate**(double theta) Concatenates the current Graphics2D Transform with a rotation transform.
abstract void	**rotate**(double theta, double x, double y) Concatenates the current Graphics2D Transform with a translated rotation transform.
abstract void	**scale**(double sx, double sy) Concatenates the current Graphics2D Transform with a scaling transformation Subsequent rendering is resized according to the specified scaling factors relative to the previous scaling.
abstract void	**setBackground**(Color color) Sets the background color for the Graphics2D context.
abstract void	**setComposite**(Composite comp) Sets the Composite for the Graphics2D context.
abstract void	**setPaint**(Paint paint) Sets the Paint attribute for the Graphics2D context.
abstract void	**setRenderingHint**(RenderingHints.Key hintKey, Object hintValue) Sets the value of a single preference for the rendering algorithms.
abstract void	**setRenderingHints**(Map<?,?> hints) Replaces the values of all preferences for the rendering algorithms with the specified hints.
abstract void	**setStroke**(Stroke s) Sets the Stroke for the Graphics2D context.
abstract void	**setTransform**(AffineTransform Tx) Overwrites the Transform in the Graphics2D context.
abstract void	**shear**(double shx, double shy) Concatenates the current Graphics2D Transform with a shearing transform.

abstract void	**transform**(AffineTransform Tx) Composes an AffineTransform object with the Transform in this Graphics2D according to the rule last-specified-first-applied.
abstract void	**translate**(double tx, double ty) Concatenates the current Graphics2D Transform with a translation transform.
abstract void	**translate**(int x, int y) Translates the origin of the Graphics2D context to the point (*x*, *y*) in the current coordinate system.

Methods inherited from class java.awt.Graphics

clearRect, clipRect, copyArea, create, create, dispose, drawArc, drawBytes, drawChars, drawImage, drawImage, drawImage, drawImage, drawImage, drawImage, drawLine, drawOval, drawPolygon, drawPolygon, drawPolyline, drawRect, drawRoundRect, fillArc, fillOval, fillPolygon, fillPolygon, fillRect, fillRoundRect, finalize, getClip, getClipBounds, getClipBounds, getClipRect, getColor, getFont, getFontMetrics, getFontMetrics, hitClip, setClip, setClip, setColor, setFont, setPaintMode, setXORMode, toString

Methods inherited from class java.lang.Object

clone, equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Graphics2D

```
protected Graphics2D()
```

Constructs a new Graphics2D object. Since Graphics2D is an abstract class, and since it must be customized by subclasses for different output devices, Graphics2D objects cannot be created directly. Instead, Graphics2D objects must be obtained from another Graphics2D object, created by a Component, or obtained from images such as BufferedImage objects.

See Also:

Component.getGraphics(), Graphics.create()

Method Detail

draw3DRect

```
public void draw3DRect(int x,
                       int y,
                       int width,
                       int height,
                       boolean raised)
```

Draws a 3-D highlighted outline of the specified rectangle. The edges of the rectangle are highlighted so that they appear to be beveled and lit from the upper left corner.

The colors used for the highlighting effect are determined based on the current color. The

```
1
2
3  ============================================================
4  Uppg 1.
5  a)  En underklass ärver alla metoder från sin superklass utom konstruktorer.
6  Falskt. En subklass ärver inte de privata metoderna.
7
8  b)  Kan statiska metoder överskuggas?
9  Nej då vore dom ju inte klass metoder längre.
10
11 c) ...implements Comparable ...
12 samt lägg till metoden public int compareTo(Object o)
13
14 d) Den terminerar bara för 0 och 1.
15
16 e)
17     m = d;          // 1 ok
18     d = m;          // 2 ger komp.fel, incompatible types
19     d = (Dog)m;     // 3 kan ge exekveringsfel beroende på vad m blivit initierad till
20     d = c;          // 4 ger komp.fel, incompatible types
21     d = (Dog)c;     // 5 ger komp.fel, inconvertible types
22     m = (Mammal)d; // 6 ok
23
24 Ny sida
```

```
25 ============================================================
26 Uppg 2.
27 import java.util.Scanner;
28 public class ClosestPrime {
29
30     static public boolean isPrime(int p) {
31         // testar om p är ett primtal
32         int rotenUrP, delare;
33         if (p == 2 || p == 3) return true;
34         if (p % 2 == 0 || p % 3 == 0 || p <=1)  return false;
35         rotenUrP = (int)(Math.sqrt((double)p)+0.5);
36         delare = 5;
37         while (delare <= rotenUrP) {
38             if (p % delare == 0) {
39                 return false;
40             } else {
41                 delare = delare + 2;
42             }
43         } // end loop
44         return true;
45     } // end isPrime
46     //============================================================================
47
48     public static void main(String[] args) {
49         int j = 0;
50         int k = 1;
51         boolean over = false;
52         Scanner in = new Scanner(System.in);
53     //  while (true) {
54             over = false;
55             System.out.println("ange ett heltal (sluta: <=2): ");
56             j = in.nextInt();
57             System.out.println();
58             if (j <= 2) {
59                 System.out.println(" avslutar ");
60                 System.exit(0);
61             }
62             if (j % 2 == 0) { // if even(j)
63                 k=1;
64             } else {
65                 k=2;
66             }
67             while (!over) { // vi kommer garanterat att hitta nåt
68                 if ( isPrime(j-k) ) {
69                     System.out.print( "närmsta primtalet är " + (j-k) );
70                     System.out.println( " på avståndet " + k );
71                     over = true; // System.exit(0);
72                 }
73                 if ( isPrime(j+k) ) {
74                     System.out.print( "närmsta primtalet är " + (j+k) );
75                     System.out.println( " på avståndet " + k );
76                     over = true; // System.exit(0);
77                 }
78                 k = k + 2;
79             }
80     //  }
81     } // end main
82 } // end Closest
83
84 Ny sida
```

```
 85 ============================================================
 86 Uppg 3.
 87 import java.awt.*;
 88 import javax.swing.*;
 89
 90 public class RotatingSquares extends JPanel {
 91     Font f;
 92     public RotatingSquares() {
 93         setPreferredSize(new Dimension(400, 400));
 94         f = new Font("SansSerif", Font.PLAIN, 15);
 95     }
 96
 97     public void paintComponent(Graphics g) {
 98         Graphics2D g2 = (Graphics2D)g;
 99         g2.translate(100, 200);
100         g2.setFont(f);
101         for(int i=0; i<=12; i++) {
102             g2.setColor(Color.BLUE);
103             g2.fillRect(20, 20, 20, 20);
104             g2.setColor(Color.BLACK);
105             g2.drawString("" + i, 20, 20);
106             g2.rotate(Math.toRadians(30));
107             g2.scale(1.1, 1.1);
108         }
109     }
110
111     public static void main(String[] args) {
112         JFrame jf = new JFrame();
113         jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
114         RotatingSquares r = new RotatingSquares();
115         jf.add(r);
116         jf.pack();
117         jf.setVisible(true);
118     }
119 }
120
121
122 Ny sida
```

```
============================================================
Uppg 4.
import java.util.Scanner;
public class PascalsTriangle  {
    // print1: skriver en triangle som ser ut som matrisen
    public static void print1(int[][] matrix) {
        for(int i=0; i<matrix.length; i++) {
            for(int j=0; j<matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
    // print2: skriver en "riktig" pascal triangel
    public static void print2(int[][] matrix) {
        for(int i=0; i<matrix.length; i++) {
            // skriv en sträng med blanka av lämplig längd
            for(int k=1; k<matrix.length-i; k++) {
                System.out.print(" ");
            }
            for(int j=0; j<matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
    public static int[][] fillInPascal(int nbrOfRows) {
        int[][] triangle = new int[nbrOfRows][];
        for(int i=0; i<triangle.length; i++) {
            triangle[i] = new int[i+1];
            triangle[i][0] = 1;
            triangle[i][i] = 1;
            for(int j=1; j<i; j++) {
                triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j];
            }
        }
        return triangle;
    }

    public static void main(String[] args) {
        // c) start // alt 1
        //int n = Integer.parseInt(args[0]);
        //print2(fillInPascal(n));
        if(args.length<1){  // alt 4
            System.out.println("No argument given");
        } else {
            try {  // alt 3
                int n = Integer.parseInt(args[0]);
                if(n<2 || n>20) { // alt 2
                    System.out.println("n must be between 2 and 20");
                } else {
                    print2(fillInPascal(n));
                }
            }
            catch (NumberFormatException e) {  // alt 3
                System.out.println("N must be an integer");
            }
        } // end if(args.length<1)
    }
} // end Pascal
Ny sida
```

```
184 ============================================================
185 Uppg 5.
186
187 import java.awt.Point;
188 import java.awt.Rectangle;
189 public class PolygonShort {
190
191     /** The total number of points.*/                                    //1p
192     private int npoints = 0;
193     /**The array of x coordinates.*/
194     private int[] xpoints = null;
195     /**The array of y coordinates.*/
196     private int[] ypoints = null;
197
198     // Constructors
199     /**Creates an empty polygon with default size = 3.*/
200     public PolygonShort() {                                              //1p
201         npoints = 0;
202         xpoints = new int[3];
203         ypoints = new int[3];
204     }
205     /**Constructs and initializes a Polygon from the specified parameters.*/
206     public PolygonShort(int[] xpoints, int[] ypoints, int npoints) {    //5p
207         // ok att kasta IndexOutOfBoundsException för alla fel
208         if ( xpoints==null || ypoints== null ) {
209             throw new NullPointerException(
210                 "xpoints || ypoints== null");
211         }
212         if ( xpoints.length != ypoints.length || npoints > xpoints.length ) {
213             throw new IndexOutOfBoundsException(
214                 "arrays must have equal length and npoints <= xpoints.length");
215         }
216         if ( npoints < 0 ) {
217             throw new NegativeArraySizeException("npoints < 0 ");
218         }
219         /* eller
220         if ( xpoints==null || ypoints== null
221             || xpoints.length != ypoints.length
222             || npoints > xpoints.length
223             || npoints < 0 ) {
224                 throw new IndexOutOfBoundsException(
225                     "...lång fel text...");
226         }
227         */
228
229         this.npoints = npoints;
230         this.xpoints = new int[npoints];
231         this.ypoints = new int[npoints];
232         // copy the array
233         for(int i=0; i<this.xpoints.length; i++) {
234             this.xpoints[i] = xpoints[i];
235             this.ypoints[i] = ypoints[i];
236         }
237         // can use arrayCopy instead
238         //System.arraycopy(xpoints, 0, this.xpoints, 0, npoints);
239         //System.arraycopy(ypoints, 0, this.ypoints, 0, npoints);
240     }
241     // Methods
242     /**Appends the specified coordinates to this Polygon.*/
243     public void addPoint(int x, int y) {                                //5p
244         if (npoints == xpoints.length) { // fälten fulla
```

```
245             int tmp[];                  // förstora
246             // eller använd en loop enligt ovan
247             tmp = new int[npoints * 2];
248             for(int i=0; i<xpoints.length; i++) {
249                 tmp[i] = xpoints[i];
250             }
251             xpoints = tmp;
252 
253             tmp = new int[npoints * 2];
254             System.arraycopy(ypoints, 0, tmp, 0, npoints);
255             ypoints = tmp;
256         }   // nu finns det plats
257         xpoints[npoints] = x;
258         ypoints[npoints] = y;
259         npoints++;
260     }
261     /**Determines whether the specified coordinates are inside this Polygon.*/
262     public boolean contains(int x, int y) {                              //1p
263         return contains((double) x, (double) y);
264     }
265         /**Determines whether the specified Point is inside this Polygon.*/ //1p
266     public boolean contains(Point p) {
267         return contains(p.x, p.y);
268     }
269     /** Returns a copy of the array of y coordinates.*/
270     public int[] getXPoints() {                                          //1p
271         int[] tmp = new int[xpoints.length];
272         for(int i=0; i<xpoints.length; i++) {
273             tmp[i] = xpoints[i];
274         }
275         return tmp;
276     }
277         /**Resets this Polygon object to an empty polygon.*/            //1p
278     public void reset() {
279         npoints = 0;
280     }
281 
282 } // end Polygon
283 
284 
285 
286 
287 
288 
289 
```