

## **Stability of the 3-body problem Visualization in Java**

---

RGP I - Week 12

### **Java language features**

---

We will see a very quick overview over the most important features  
Then we will discuss several simple applets to illustrate the  
important features

The aim of this lecture and exercise is to give you a first taste of  
Java. You can learn more by using web-based tutorials

As a homework we ask you to

- change an applet for the Kepler problem to the 3-body problem

- use this applet to study

  - stable solutions of the 3-body problem

  - instabilities and chaotic orbits in the 3-body problems

  - discuss the moon, asteroids and rings of Saturn

## Stability of the 2-body problem

---

The Kepler problem is stable:

Degrees of freedom: 12

Integrals of motion:

Center of mass: 3

Total momentum: 3

Angular momentum: 3

Energy: 1

Lenz vector: 1

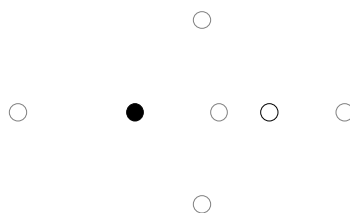
Thus there is only one degree of freedom (the angle of the particle), and the motion is stable, as you know from the exact solution

## Solutions of the 3-body problem

---

There is no stationary solution

There are stationary solutions in a rotating reference frame (the five Lagrange points)



The two equilateral triangles are linearly stable

The three co-linear solutions are unstable

## Stability of the 3-body problem

---

The full 3-body problem

18 degrees of freedom

10 Integrals of motion

8 degrees of freedom remaining: impossible to say anything

3-body problem in 2 dimensions

12 degrees of freedom

6 Integrals of motion

Center of mass 2

Total momentum 2

Angular momentum 1

Energy 1

still impossible to make any statements

## The restricted 3-body problem

---

is a

3-body problem in 2 dimensions

with the mass of the third body  $m_3 \rightarrow 0$

and a circular orbit for the first two bodies

this is an autonomous mechanical system for the third body with three degrees of freedom for the isoenergetic orbits

4 degrees of freedom

1 integral (energy)

Now the Kolmogorov-Arnold-Moser (KAM) theorem can be used to predict stability of moon-like orbits

## Resonances, asteroid belt, rings of Saturn

---

Let a medium weight body (Jupiter, moon of Saturn)  
run on a circular orbit around a heavy body (Sun, Saturn)

Investigate the orbit of a very light body with a fractional period  
(e.g.  $1/2$ )

Compare to an orbit with irrational period

Do you see effects of resonances that destabilize some orbits?

These resonances can explain the gaps in Saturn rings and the  
Kirkwood gaps in asteroid histograms

## The Java language

---

was developed by Sun as a new and better object oriented  
language

is more than a language. It contains

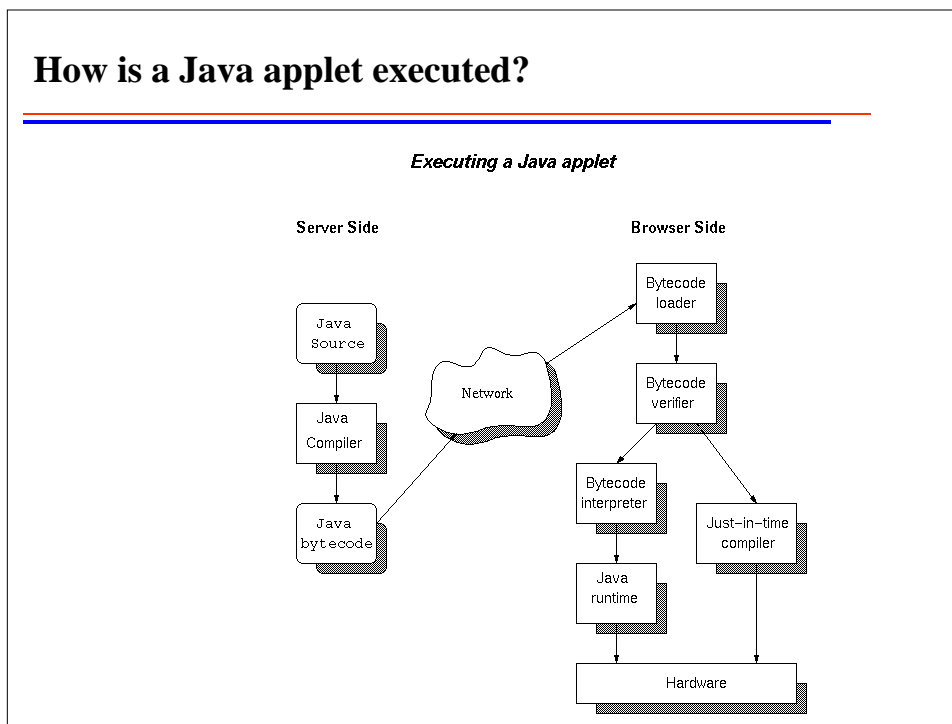
- a platform independent Java bytecode, like a machine language
- a Java virtual machine on each architecture to execute Java bytecode programs
- a huge class library useful for graphics and distributed computing

However unfortunately at the moment the motto

“Write once, run everywhere”

should rather read “Write once, debug everywhere”

## How is a Java applet executed?



## Differences between Java and C++

Java is an "improved" C++

Some features removed:

**No pointers**, all class variables are references

Automatic **garbage collection** removes need for **delete** statements and **destructors**

**No operator overloading**, as it is often misused

**No templates**

Improvements (?)

Garbage collection (makes programs safer but slower)

Improved multiple inheritance by using "**interfaces**"

Big standardized **class library**

Good support for **network computing**

Real standard, same on all machines

No separation between declaration and definition

## A first applet

---

The file „HelloWorld.java“

```
public class HelloWorld
  extends java.applet.Applet
  {
  public void
    paint(java.awt.Graphics g)
    {
    g.drawString(
      "Hello, world!", 100 , 25);
    }
  }
```

The file „HelloWorld.html“

```
<applet
  code="HelloWorld.class"
  width=250 height=50>
</applet>
```

Applets are derived from the class `java.applet.Applet`

The `paint` function draws the contents of the applet

`drawString` draws a string into a graphics object at a given place

The source file is compiled using the **javac** compiler.

This produces a `.class` file.

The applet is started from the HTML file using the applet tag

Make sure that the `.class` file is in the same directory or use a `codebase=... .` argument.

## Java Packages

---

are like C++ namespaces are, but better done

```
java.applet.Applet
```

is the Applet class in the package “`java.applet`”

Instead of the C++ `using` statement, `import` is used:

```
import java.applet.*;
imports everything from the package
import java.applet.Applet;
imports only the Applet class
```

After importing the `java.applet.` qualifier can be omitted

A huge class library exists. Look at the documentation at

<http://www.java.sun.com/docs/index.html>

especially at the JDK API documentation

## An overview over packages

---

`java.lang`

essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads.

`java.io`

`java.util`

`java.net`

`java.awt`

Package that provides an integrated set of classes to manage user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields. (AWT = Abstract

`java.applet`

Package that enables the creation of applets through the Applet class.

## Class Definitions

---

```
public class HelloWorld
extends
java.applet.Applet {
...
}
```

`public` here means the class is visible from outside the file  
`extends` is the derivation syntax  
In C++ this would be:

```
class HelloWorld :
public java::applet::Applet
{
};
```

## Class member functions

---

```
public class HelloWorld
extends java.applet.Applet {
    public void
    paint(java.awt.Graphics g)
    {
        g.drawString(
        "Hello, world!", 100 , 25);
    }
}
```

Member functions are defined in the class definition, unlike in C++ where they are only declared in the class

Keywords are written before each member:

```
public
protected
private (default)
final (cannot be changed by
derived classes)
synchronized
```

Data members can also have such modifiers

```
public, protected,
private
final (means const in C++)
```

## A "real" applet to draw shapes

---

We want to write an applet that allows you to

- choose a color (red, green or blue)
- choose a shape (circle or square)
- draw the shapes at some place

For that we need

- a shape class
  - a square class derived from it
  - a circle class derived from it
- a container for all shapes
- pop-up menus to select the shape and color
- an event handler to react to mouse clicks

## The shape classes and virtual functions

```

abstract class Shape {
    static public final int
    shapeRadius = 20;
    Color color;
    int x;
    int y;

    abstract void draw(Graphics g);
    // drawing is always done in a
    // Graphics object
}

// this is a reference to the class
// and not a pointer to the class as
// in C++

class Circle extends Shape {
    void draw(Graphics g) {
        g.setColor(this.color);
        g.fillOval(this.x -
        shapeRadius, this.y -
        shapeRadius, shapeRadius * 2,
        shapeRadius * 2);
    }
}

class Square extends Shape {
    void draw(Graphics g) {
        g.setColor(this.color);
        g.fillRect(this.x -
        shapeRadius, this.y -
        shapeRadius, shapeRadius * 2,
        shapeRadius * 2);
    }
}

```

## The SimpleDraw Applet

```

import java.applet.Applet;
import java.util.*;
import java.awt.*;

public class SimpleDraw extends
    Applet {
    Vector drawnShapes;
    Choice shapeChoice;
    Choice colorChoice;

    // all these variables are only
    // references to objects!!!

    /** Create the GUI. */
    public void init() {
    }

    /** Draw all the shapes. */
    public void paint(Graphics g) {
        ...
    }

    /** Create a new shape. */
    public boolean mouseUp(Event e, int x, int y) {
        ...
    }
}

```

## Setting up the GUI in init()

---

```
public void init() {
    drawnShapes = new Vector();

    shapeChoice = new Choice();

    shapeChoice.addItem("Circle");
    shapeChoice.addItem("Square");
    add(shapeChoice);

    colorChoice = new Choice();
    colorChoice.addItem("Red");
    colorChoice.addItem("Green");
    colorChoice.addItem("Blue");
    add(colorChoice);
}
```

Now the new operator is used to create the objects

Delete is not needed as the objects are automatically deleted when no longer referenced

A **Vector** can hold any object

**Choice** is a pop-up menu to which entries are added by **add**

Then the pop-up menu is added to the applet by the **add** function of the applet

## Drawing the shapes

---

```
public void paint(Graphics g) {
    Shape s;
    int numShapes;
    // get the number of shapes
    numShapes = drawnShapes.size();
    for (int i = 0; i < numShapes; i++) {
        // convert each element to a shape
        s = (Shape)drawnShapes.elementAt(i);
        // draw the shape
        s.draw(g);
    }
}
```

## Creating new shapes by mouse clicks

---

```

public boolean                               // get the color
mouseUp(Event e, int x, int y) {
    Shape s;
    // get the menu selections
    String shapeString =
        shapeChoice.getSelectedItemAt();
    String colorString =
        colorChoice.getSelectedItemAt();

    // create the shape
    if
        (shapeString.equals("Circle"))
        s = new Circle();
    else
        s = new Square();

    if (colorString.equals("Red"))
        s.color = Color.red;
    else if
        (colorString.equals("Green"))
        s.color = Color.green;
    else
        s.color = Color.blue;

    s.x = x;
    s.y = y;

    // add the shape to the vector
    drawnShapes.addElement(s);
    // force a redraw
    repaint();
    return true; // we handled the click
}

```

## The Kepler applet

---

An applet to draw the motion of a planet around the sun

We want to

- enter the starting coordinate and velocity
- reset/change these coordinates
- let the planet move

The planet shall move while we watch or type new coordinates

We need a separate thread (lightweight process) to propagate the planet

New issues to discuss

- threads
- interfaces
- exceptions

## Interfaces and Threads

---

are a better implementation of multiple inheritance  
 are like abstract C++ classes with only pure virtual functions  
 For threads the **Runnable** interface is needed

```
class KeplerPanel extends Canvas implements Runnable {
public void start() {...}
public void run() {...}
public void stop() {...}
...
}
```

These three member functions are needed for threads and declared in the **Runnable** interface

**Canvas** is a graphics element to draw on

## Exceptions

---

are the best way to deal with errors  
 we want to implement a field to enter numbers:

```
class NumberField extends TextField {

public NumberField(String s, int w) {
super(s,w); // calls the constructor of the base class
}

public double doubleValue () throws
java.lang.NumberFormatException {
Double val = new Double(getText()); // this can throw the exception
return val.doubleValue(); // Double is a class for a double
}
}
```

The **Double** constructor does not know how to deal with the incorrect input. The exception is a way to let the caller deal with the problem/error

## Catching exceptions

---

```
synchronized void read_params()
{
    try {
        pos1.x=x1.doubleValue();
        pos1.y=y1.doubleValue();
        vel1.x=vx1.doubleValue();
        vel1.y=vy1.doubleValue();
    }
    catch (Exception e) {
        running.setState(false);
    }
    return;
}
```

Synchronized means only one thread is allowed to execute such a function at any time

Here we try to read the values of text fields

If there is an exception execution will jump to the catch clause

There we stop the running of the program

## Further information

---

can be found on Sun's Java web page  
documentation  
examples  
tutorials and text books

we will now discuss the Kepler applet

## Next exercise

---

extend the Kepler applet for a 3-body problem

- enter masses, positions and velocities for two bodies
- the sun shall be a body with infinite mass resting at the center
- implement the possibility of viewing the system in a reference frame rotating with one of the bodies (use a CheckBox)

take care of

- adaptive time steps
  - larger time step for small forces
  - smaller time step for large forces

then study the following problems:

- stability of the Lagrange points
- stability of the moon orbit
- gaps in Saturn's rings

## Is OOP useful?

---

Object oriented programming (OOP) uses **inheritance**

Useful for graphical user interfaces

but did you ever use inheritance for simulations?

For scientific computing generic programming with templates is more useful

OOP is not well designed for scientific programming:

```
class Animal {
    public: virtual Animal mates (const Animal&);
};
class Horse : public Animal { ... };
class Cow: public Animal { ... };
Horse horse; Cow cow;
Animal child = cow.mates(horse);
```

Do you see the problem? It is very widespread!

## Java for scientific computing?

---

While Java is useful for network-based business applications and for applets it is missing several features important for scientific computing:

- operator overloading

- complex numbers

  - either write always explicitly as real arithmetic

  - or use a (slow) complex class, but without operator overloading!

- fast floating point arithmetic

  - all bits of the result have to be equal on all machines in Java!

  - this means: `x=b*c; y=a*b*c;`

  - cannot be replaced by: `x=b*c; y=a*x;`

  - This makes Java 2 to 10 times slower than C, Fortran or C++!

- No templates

The Java Grande Forum wants to push Sun to change the language to make it useful for scientific computing

Are succeeding in changing floating point arithmetic