

Christophe Cornu

Fun Science With Your Computer

Original Edition

Last modified: 2007/7/14

www.funsciencewithyourcomputer.org

Copyright 2007 by Christophe Cornu

Table of Content

<i>Welcome note.....</i>	<i>3</i>
<i>Introduction.....</i>	<i>7</i>
<i>How to run the programs in this book</i>	<i>9</i>
<i>Progress Card.....</i>	<i>11</i>
<i>Activity 1 – Pseudo Random.....</i>	<i>14</i>
<i>Activity 2 - Monte Carlo.....</i>	<i>23</i>
<i>Activity 3 - Random Walk.....</i>	<i>33</i>
<i>Activity 4 - Translator.....</i>	<i>43</i>
<i>Activity 5 - Launcher.....</i>	<i>55</i>
<i>Activity 6 – Path Finder</i>	<i>64</i>
<i>Activity 7 - Slow or Fast</i>	<i>70</i>
<i>Activity 8 – Space Simulation.....</i>	<i>82</i>
<i>Activity 9 - Web Browser</i>	<i>94</i>
<i>Activity 10 – Vector Graphics</i>	<i>101</i>
<i>Activity 11 - Chess</i>	<i>111</i>
<i>Epilogue.....</i>	<i>130</i>
<i>Index.....</i>	<i>131</i>

Welcome note

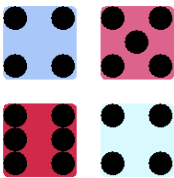
Science and technology are fun to learn when you play with them. It's all about imagination, discovery, trial and doing things with others.

How does a computer play chess? How can you create a space game that uses the laws of gravity? Create a maze and find a way out? How about a self-learning language translator? Throw a dice to calculate the digits of Pi 3.141592653... Yes, that's right. You will achieve all of the above. You will run, play with and improve computer programs designed specifically to help you understand the big picture.

This book is for teenagers (grade six and above), students, parents and teachers. If you have no programming experience you will need a friend or a mentor familiar with Java programming to get your computer setup and ready to run the programs. [How to run the programs in this book](#) will help you getting started.

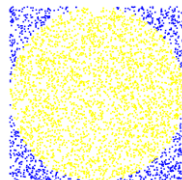
Each activity is like a cooking recipe producing a surprising and unique meal. Our eyes open up to a key scientific puzzle that once teased the best minds throughout the centuries. First we follow the recipe – program the computer. Then we enjoy the meal and share it with friends – we run the program and show it to our friends. And then we refine it – we take on the guided exercises proposed in the activity. We add our own improvements to the recipe - the program. Each activity highlights related major dates and famous people. We get a sense of the amazing evolution of ideas and inventions throughout our history – with more yet to come.

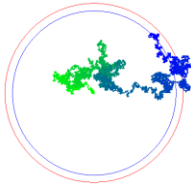
Activities can be explored in the order of your interest.



Simulate a dice with [Activity 1 – Pseudo Random](#). Did you know it is very hard to have truly random numbers? Once we know how to generate numbers that apparently look random, we will learn how to do very interesting things with them.

What can you do with one random number? Or with two random numbers? Not much. Be prepared to be surprised. You can do nearly everything with a million of them. [Activity 2 - Monte Carlo](#) illustrates how the digits of Pi can be calculated by randomly picking points inside a square.



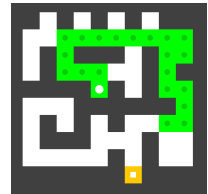


Once you manipulate millions of random numbers, you can predict average behaviors. For example, if you walk randomly changing direction after each step, you can estimate the distance you are from your starting point. Can you guess what that distance is? [Activity 3 - Random Walk](#) introduces you to something very dear to biologists and physicists.

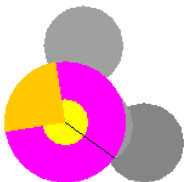
A newborn baby does not know much about life on Earth. What a human baby is incredibly good at is learning – learning how to see, how to move, how to communicate. [Activity 4 - Translator](#) is a program that initially knows nothing except how to ask users to complete information it is missing. You can teach it any language you can think of.

In 2006 Joshua Plotnik showed elephants pass the mirror test. Elephants, along with humans, dolphins and great apes, can recognize themselves in a mirror. Well, [Activity 5 - Launcher](#) is a program that knows how to look upon itself to discover what it can do. You can use it to start any of the programs from this book. And your own.

How does a postman decide the order in which mail is to be delivered? The postman doesn't. A computer searches the most efficient trip for the postman. Just the same way you plan your vacation road trip and find the best path from your home to your five star hotel. [Activity 6 – Path Finder](#) can find any path in and out of any map.



Consider a recipe that requires three hours to make a cake for four people. Can you improve the recipe so the cake is just as good but requires less than sixty minutes of preparation? [Activity 7 - Slow or Fast](#) investigates two ways to resolve a mathematical problem. One is constantly fast. The other is increasingly slow. Be a good cook – er, a good programmer.

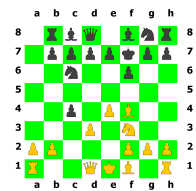


We all worry about weight or have relatives who worry about theirs. Well it is time to have a little recreation with mass, gravity and Mr. Isaac Newton. [Activity 8 – Space Simulation](#) is a multiplayer space simulation action game realistically based on the laws of Physics.

We spend more time on the Internet than we do watching

TV. Have you ever wondered how a web browser works? [Activity 9 - Web Browser](#) shows how we can create a very simple text based web browser by loading content and navigating through its hyperlinks.

Animation movies are for the most part computer generated. There are ways to create graphics that can be adjusted to any size and filled with any kind of color you need. We will learn how to draw a Chess board in [Activity 10 – Vector Graphics](#). You can, of course, change the knight so it looks like your favorite T-Rex dinosaur.



What about the Holy Grail of artificial intelligence? What greater challenge than teaching a computer how to perform something only intelligent human beings were thought to be capable of? Yes, we will program and be able to play Chess against our own program in [Activity 11 - Chess](#). Our brain really does work differently than our Chess machine, so is it fair to oppose a human to a machine?

Capture your programming successes with the [Progress Card](#). You can be proud of your achievements. Every step you complete opens the door to Science a little bit wider...

Conventions

Important parts of the Java programs are explained and presented as follow. Line numbers show up on the left so you can easily look them up inside the whole program you get from the web site at www.funsciencewithyourcomputer.org. The Java code below is taken from [Activity 11 - Chess](#) and shows lines 14 to 15 of the Chess.java program. If the current chess game is over then the program will stop playing...

```
14 |         if (chess.gameOver())
15 |             break;
```

Each activity is followed by two hands on exercises – similar to two scientific experiments conducted with your computer. You modify the Java program of the current activity and then run it. Four symbols are used to describe each experimental step. They show to the right of the text in the exercises.

The magnifying lens signals a portion of code to search in the Java program. The text left to the lens will specify which lines of code and the actual code will often be shown in the book to facilitate the reading.



Modifying a program is about removing some of the existing code and replacing it with new code. The cross image asks you to remove a portion of code in the Java program. Text to the left will mention the line numbers to be removed e.g. lines 5 to 15.



The plus sign in a circle tells you to add some new code – often after you removed some of the old code. The text right to the plus sign specifies where the code should be added and the actual new code follows after the plus sign.



Run the Java program you are editing in the exercise when you reach the following start button. Text left to the button will describe what you should pay special attention for when the program executes on your machine.



I would like to thank my friends and coworkers at the IBM Ottawa Labs for their feedback and detailed reviews. I am particularly indebted to Jim Des Rivieres, Mike Wilson, Carolyn McLeod, Nick Edgar, Jean-Michel Lemieux, John Arthorne and Adrian Cho. The book is filled with many imperfections only because I did not always include all of their corrections and ideas in the current edition. I wish to thank Professor Martin Moskovits, Dean, Division of Mathematical, Life and Physical Sciences at University of California, Santa Barbara and Professor Pierre-Gilles de Gennes, Nobel laureate in physics. Their working style convinced me the conquest of knowledge is a fine art requiring an extraordinary blend of interdisciplinarity and collaborative skills. That also means *everyone* can enjoy research and engineering activities with mentors showing them the true human face of Science, its many doubts and trials.

It is now time for you to meet three new friends. Sarah, Mike and Liliane will travel with you through the gates of Science. Their guide is a retired professor familiarly referred to as Alex...

Have a good journey!

Chrix

Introduction

Sarah and Mike have arrived at their summer camp located in a small village in the French alps. Sarah is twelve years old and her brother Mike is eighteen. They both come from the province of Ontario in Canada. It is their first time outside of North America and they are very excited about this adventure in Europe. Now they are seated in front of an old chalet on a sunny day in the Alps. The chalet is filled with hiking equipment, but also computers and telescopes. Yes, they really are at a Science camp!

The two teenagers are meeting with Alex and Liliane who supervise their group. Alex takes care of the science and computer activities. He is a retired professor who worked in many different parts of the world. Liliane is helping Alex. She studies literature and history at the university La Sorbonne in Paris. Taking part into a summer camp is a great summer job, she thinks. Alex was repairing an old and dusty computer.

"What an antique! That has to be over fifty years old..." Mike said.

"Alex, what are you doing with that strange box?" Sarah asked.

Alex observed the two teens. He did not say a word and continued to type on the computer keyboard. Liliane was returning from the kitchen, bringing fresh croissants and milk to the table. Soon they were happily eating. Sarah was observing the professor with increasing curiosity and impatience. She asked again. "Alex, really what are you doing?"

Alex looked at the teens again. Sarah had her tiny music player, headphones and cell phone hooked to her belt. Mike was making himself comfortable playing video games on a small portable console.

- "Do you know how your cell phone works, Sarah?" Alex asked.

- "Of course I know." Sarah was feeling a little bit insulted and her brother was shrugging. She took the phone and showed it to Alex. "Press the green button located here. Turn this wheel to select your phone number. Press the wheel again. And when someone calls me, it plays my favorite tune."

- "I didn't ask if you know how to use it, Sarah. Do you know how it remembers phone numbers? Do you know how it plays your favorite song?"

Sarah was a little bit confused by these unusual questions. "Well, it has a computer inside, I suppose."

Alex turned toward Mike. "And you Mike, do you know how are games created on your console?" Mike felt a little bit annoyed. He had the best console, with the best animation. What kind of question was that? He could use a cartridge or download games from the web. And then he could play alone or with friends. Though in this remote village, he had not found anybody to play with. And Alex's antique certainly could not play his favorite game, could it?

Alex turned the screen so they could see it. It was filled with green characters. "See this? Can you guess what it is? It's my first program. I was your age, Sarah and that computer was the first computer at the school in our village. We didn't want to waste time and money on commercial games. So instead I created my own! Here, here is a Chess game."

Mike and Sarah sat near Alex. What could possibly be done on such an old machine?

"How do I move the pawn? There is no mouse." Mike said.

"Use the keyboard. Type in the coordinates of the piece and where it should go - e2e4 for example."

Mike wasn't convinced. This looked really horrible. "There's no music." he said. Sarah didn't know how to play Chess so she let Mike try it out. After a few moves, Sarah noticed a message on the screen "Black Wins". Her brother was looking very frustrated. Alex was smiling.

"How did you do that?" she asked.

"You simply learn how to talk to the machine so you can teach it everything that comes to your mind. It doesn't come all in one day. We can start with simple programs that do fun things. A program is made of text instructions telling the machine how to do different things. You enter the program into the machine. You don't have to understand every detail of the program, just copy it from a programming book or get it from the Internet. You'll see if it makes sense to the machine – it works or it does not work. When it works, modify the text here and there to see how it runs differently on the computer. Over time, it becomes quite natural, like learning how to swim or skate. It takes some practice and someone to guide you through the first steps."

Mike was looking at the green screen. "So I could also learn how to write a chess game and beat this old chess game?"

"You sure could." answered Alex. "By the end of summer, you could certainly do that."

Throughout the summer, Mike, Sarah and Liliane gathered with Alex a few times a week. With Alex's patient explanations, they started to get the feeling that it was really fun to be able to program a computer to do anything you could imagine!

How to run the programs in this book

The programs in this book can be run on personal computers, of the kind you most likely have at home, at school or at work. They are written in a very widely used language called Java.

The Java language offers many advantages. It is free to install – you don't need to spend money buying special software. It is very popular and appreciated at university and in the professional world. So you can be sure time spent studying these programs is bringing real and valuable skills. Finally, most computers can run Java programs.

Today's computers are used to send emails, browse the web and download digital photos from your camera. Programmers need an extra step. They need to install a few tools. Go to our web site www.funsciencewithyourcomputer.org and follow the instructions.

The book contains screenshots showing what the program looks like once executed on the machine. Seeing is not enough. Try running these programs on your computer, by yourself or assisted by a friend or a teacher who knows a little bit about programming. Only then is the experience complete. There is no trick. The machine does what you tell it to do. In a way it is magic, at times frustrating. Programmers can spend hours trying to understand why the computer does not do what they expect it to do. This book wants you to discover this feeling too.

Once you have installed the tool to edit and run Java programs, download the small Java programs used in this book from the www.funsciencewithyourcomputer.org site. After you successfully edit and run a program, take some time to congratulate yourself. Look back at the program. Try to modify it here and there. You don't have to understand everything before running it – nobody really understands everything in general. Experiment and see what the effect is on the computer when you run the program again after making some changes.

The activities are organized to encourage you to further modify the given programs. For example you can modify the graphics of the chess game, or change the values of certain arguments in the program to calculate the number Pi with more or less accuracy. Play. Try things out. Fail, try again, and succeed. That's it. You have become a programmer.

Although programs in this book are written in Java, this is not a book about the Java language. Excellent and in-depth books exist on this topic. It is about programming in general and about turning entertaining ideas into simple programs in particular. It is not about writing perfectly optimized Java programs – the fact is the code in this book will not please expert Java programmers. The

decision was made to write simple programs that are easy to read, using an extremely limited set of language features. All too often, there is a false sense of complexity resulting from the multiplication of choices available to programmers. There are so many ways to solve a given problem - this in itself creates problems. Imagine going to a restaurant with no daily specials. Everything is customizable with dozens of choices for mineral water, etc. In a few rare cases such as on your wedding day, that would be a good thing. In daily situations, deciding what to order takes more time than you can afford during your lunch break. Don't run away from programming and Science because you don't understand 'everything' immediately. You will be in control of the examples given in this book.

Progress Card

You can track every progress you make with the following progress card. Go through an activity. Run the program related to that activity. Praise yourself and mark that activity as completed in table 1 below (indicate the date). Your job title progresses with the number of activities you have completed.

Table 1 - Activities completed - progress

Activity	Completion date	Score	Job Title
1	.. / .. / ..	50	Developer
2	.. / .. / ..	100	
3	.. / .. / ..	150	Consultant
4	.. / .. / ..	200	
5	.. / .. / ..	250	Committer
6	.. / .. / ..	300	
7	.. / .. / ..	350	Senior Developer
8	.. / .. / ..	450	
9	.. / .. / ..	550	Senior Consultant
10	.. / .. / ..	650	
11	.. / .. / ..	1000	Architect

Each activity includes two exercises. An exercise teaches you how to modify the program and improve on a particular aspect of the activity. Once you have completed all the exercises for a particular activity, fill up the corresponding entry in table 2. You have demonstrated that you are now an expert in that field. Scientists frequently gain expertise in two or three fields. Some are famous worldwide because they have knowledge and skills in very important areas that no one else can match.

Table 2 - Exercises completed per activity – expertise

Activity	Exercise 1	Exercise 2	Expertise
1			Probabilities
2			Monte Carlo Simulation
3			Statistics
4			Natural Language Translation
5			Framework design and reflection
6			Graph Theory
7			Performance
8			Classical Physics and Laws of Gravity
9			Web
10			Scalar Vector Graphics
11			Game Theory

The purpose of this book is to encourage you to program by yourself - and with other students or friends. After you complete a couple of activities and exercises, think about programs you would like to create. Modify one of the programs from a completed activity. It is convenient to start from a program that works instead of starting everything from nothing. Give it a name and do something unique with it. Once your own program works well, write its name, the date you completed it and a short description. The more programs you write, the more experience you get. See the extra job title you get in table 3 below based on how many programs you wrote by yourself - or working with another person, which is another very good way to learn.

Table 3 - New programs created by yourself or with a friend

New program	Completion date	Description	Job Title
1.	.. / .. / ..		Inventor
2.	.. / .. / ..		
3.	.. / .. / ..		Chief Inventor
4.	.. / .. / ..		
5.	.. / .. / ..		Research Officer
6.	.. / .. / ..		
7.	.. / .. / ..		Program Manager
8.	.. / .. / ..		
9.	.. / .. / ..		Director of Research and Development
10.	.. / .. / ..		
11.	.. / .. / ..		Chief Technical Officer

The challenge is to become an architect – table 1 – with as many as eleven areas of expertise – table 2 – who works as Chief Technical Officer – table 3.

Special note to teachers and parents – course work

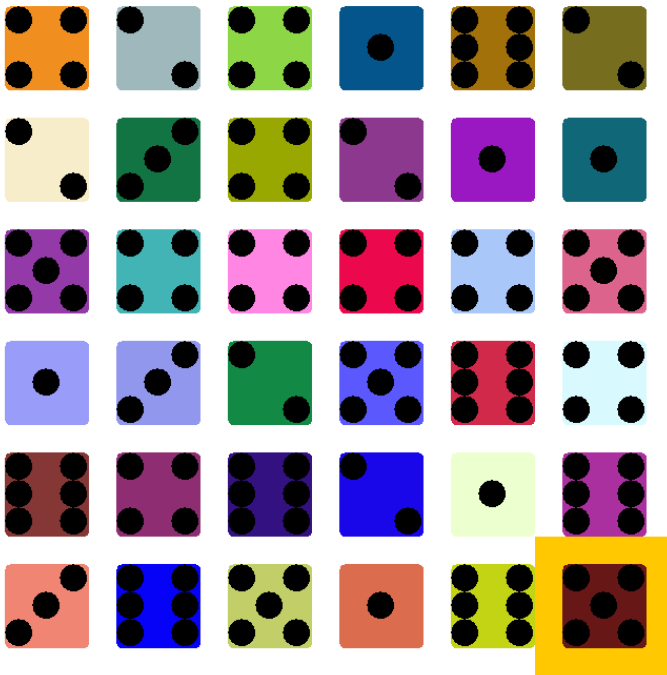
Progress cards can structure a course work. Teachers and parents have the difficult task of propagating knowledge from one generation to the next. I hope you will find the activities and exercises helpful in giving colorful faces to Science. Motivated students will challenge and master their materials rather than just strive to survive or skip science classes. Programming in this book plays with Mathematics (geometry, trigonometry and statistics) and Physics (classical mechanics). We show Science in motion with examples taken from recent discoveries. Students can see for themselves that what they learn at school is concrete and necessary to resolve real problems – even fun ones like the ones presented here. Science serves a purpose, and it goes beyond getting the best grades to get admitted into the best university or college. It is also about having fun challenging what we know and can do with our world.

Activity 1 – Pseudo Random

Alex was listening to the results of the 6-49 lottery on the radio. Then he sighed and turned off the radio. He had obviously not won. "Mike, Sarah, come here", Alex said. "How would you generate random numbers with a computer?"

Computer executes the program `PseudoRandom.java`

Dice: 5 (from calculated pseudo random value 6756118)



Mike and Sarah were certain it ought to be really easy. Alex asked for ways to create random numbers without a computer.

"Throw a dice and you look at the result" said Sarah.

"Yes, a dice is good. It generates a random number between 1 and 6."

"Use the letters in the Scrabble game, put them in a bag and grab one" Mike suggested.

"True. That's good for the Scrabble game, but if you want any letter to have the same chance to come as any other, then you need to make sure you only put one of each letter in the bag, right?"

"Oh I see. Scrabble has rare letters only once, but many times very common letters. So you'd have many more chances to pick those. Yes you're right."

"If you want numbers between 1 and 49, you can use the results of the lottery," pointed out Sarah. She had followed the draw on the radio carefully. Imagine if Alex had won!

"Correct again" said Alex.

"You can say numbers as they go through your mind."

"Do you really think so? Let's verify that one. Mike, write down one hundred times the letters H or T in the order that comes to your mind. Sarah, flip this coin, throw it a hundred times and record the results - heads or tails."

Mike handed out the two pieces of paper. Alex circled the portions in the list where the letter H was repeated - like the coin had given heads two or three times in a row for example. He also circled the portions where T was repeated. And he compared the two series from Mike and Sarah. "Sarah, this is yours, Mike that's yours."

Mike and Sarah's curiosity jumped up. "That's right! How did you know?" they asked.

"In Sarah's list, there are many more repetitions - sequences where the same result happened consecutively. That's one indicator of true randomness - there is a significant probability of having the same side of the coin occurring five or six times when you flip the coin a hundred times. When a human tries to produce random series of H or T choices, we tend to think it is not random to give the same result five or six consecutive times - it feels artificial. This is a known psychological behavior which can be used to detect true random versus human random results."

"That is so weird... So there is random, and random?"

"Yes. Random has its laws too. They can be applied to find people who lie about their taxes."

"How is that possible?"

"People who make up numbers about how much income they declare in different parts of their business often do a very poor job at bringing up random numbers or numbers that have the same statistical properties from data originating from real taxes. When the government detects such deviation, they target these individuals for further inquiries."

The teenagers were a little bit shaken. They had thought something was either random or not. And now they were starting to realize things were a little bit more varied than they had imagined.

"So now, how would you generate random numbers with a computer?" Alex asked again. "Think about it for a while. It's time for my national news on TV."

Half an hour later, Alex came back. "How did it go?" he enquired.

“Well, we thought it would be easy. The computer knows how to calculate, so we thought of a formula to generate random numbers.” Sarah explained.

“We have defined a very complicated formula. It's pretty smart. It's doing a bunch of cosine, sinus and logarithmic computations based on what digits are in different positions in the number.” proudly presented Mike.

Alex looked at it. “And does it work?”

They showed him some numbers the program had generated. The numbers looked quite different.

“What initial number did you use?”

“We just picked one with different digits.”

“And what happens if you restart the program?”

Of course, the exact same series of numbers were reproduced. “Do you see where I'm getting at?” asked Alex. “Your numbers look different. Yet, the next number is entirely defined from the previous number. So if two machines run your program, starting with the same number, they print the same results from there.”

“A random number has no memory of the numbers that come before it. When you flip a coin, it does not matter if just before the coin had been flipped on tail or on face. If you throw a dice, it does not matter if you previously had a pair of six. You are still just as likely to get a one as you are from getting any of the other sides. So your program does not generate random numbers.”

Mike and Sarah were disappointed. But they agreed with Alex. Yes, Alex was right.

“So how do you do it, Alex? What is the solution?”

Alex frowned, and smiled at them.

“You can't. A computer cannot generate random numbers.” And he waited for their reaction.

Now the teenagers did not want to believe him. He had to explain.

“You are very close to it. What your program does is generate pseudo random numbers. Numbers that have some statistical properties of true random numbers. Yet there is a hidden order in your result which you can use to predict the next number. It's determined by the very mathematical formula that you picked. It will always yield the same result for a given input.”

“That's exactly what computers were made for. To calculate and be reliable, so you can count your money, count votes, make simulations and always get the same results for a given set of data. Imagine you go to your bank to deposit a one hundred dollar check and sometimes the bank adds eighty dollars, sometimes it adds one hundred and five dollars (you could see Alex was not very kind with his banker as he assumed the banker's computer made mistakes that profited to the bank, not Alex...). That would be a difficult world to live in, isn't it?”

“The good news is that for many uses, pseudo random numbers are just as useful as true random numbers. And they are used extensively for many

simulations, games. Many mathematicians and programmers work very hard to bring up increasingly reliable pseudo random number generators.”

Liliane's notes

Alex's program uses a pseudo-random number generator invented by a person who is very famous for his work on computers last century. John Von Neumann invented that method to produce apparently random numbers in 1946. He knew they were not truly random. He knew you could predict the next number based on what current number was used. He knew the numbers repeated themselves after a certain period. But the method was simple and it could be run by the computer very efficiently - much better than what Mike and Sarah had originally invented.

Take a number with eight digits. Multiply this number by itself. You get a number of sixteen digits. Remove the first four and last four digits and you now have a new eight digits number. Repeat as many times as you need eight digit numbers. You are generating what looks like a series of random eight digit numbers.

How do we select the initial eight digits number? That is a very good question indeed. If we run the method starting from the same initial eight digits number, then we obviously get the exact same series of numbers. So they are not random. They are pseudo random. An important step is to randomly select the initial eight digits number so that we don't run the same results every time the program is started. And it looks like we are pretty much forced into a corner here. That's precisely what we were trying to solve initially... Chicken and egg story. In practice, a program can use the amount of milliseconds in the computer's clock. It is unlikely that running the same program on two machines or on the same machine at different times will happen at the exact same millisecond. So for many applications, we can live with that kind of technique.

There is one more thing to perform so we can truly use our pseudo random numbers. We have eight digit numbers. But what if we want to use our program to replace a dice? So we want numbers between one and six. We could extract the last digit in our eight digit number but that would be a number between 0 and 9. There are different ways to achieve this. A common technique is to remove six from our eight digit number. And repeat again as many times is needed, till the resulting number is lower than six - i.e. between zero and five. This is so common of an operation that all computers actually know how to calculate it very quickly. It's called modulo. Its symbol is %. For example $13 \% 6$ is read thirteen modulo six. That's equal to 1. $13 - 6$ equal 7 which is still greater than 6. $13 - 6 - 6$ equals 1 which is now lower than 6. So we get our eight digit number modulo six, and we obtain a number between zero and five. We want a number between one and six. We simply add one and we are done. We now can simulate a dice using our eight digit pseudo random number generator.

Here is how to actually teach the computer how to do it.

This defines the name of our program to create and display pseudo random numbers. Giving a name to our program allows other programs to use it (that's why we say it is 'public'. We can choose any name, so it's best to pick one that reminds us of what the program is for.

```
9 | public class PseudoRandom {
```

Our program starts here. This is the main part of the program that gets first executed by the computer.

```
11 | public static void main(String[] args) throws  
    Exception {
```

First we initialize our program with a number. Here we chose the number 12345678.

```
12 | PseudoRandom random = new PseudoRandom();  
13 | long value = 12345678;
```

We request a portion of the computer's screen so we can draw images of dices later.

```
14 |     Screen screen = new Screen();  
15 |     screen.setVisible(true);
```

We want the program to create and display random numbers. So we tell it to repeat the instructions that follow this line for ever.

```
16 |     while (true) {
```

We ask the computer to give us a pseudo random number using the previous pseudo random number.

```
17 |         value = random.calculateNext(value);
```

What we really want is simulating a dice. We need to transform our pseudo random number so that its value is between one and six. That's taking care of by our dice function.

```
18 |         int dice = random.dice(value);
```

We now simply need to ask the computer to display it on the screen. For fun, this will draw an actual dice, not just a plain number. And the color of the dice will be based on the value itself (screen can show millions of different colors, it will use the one that corresponds to our pseudo random number).

```
19 |      screen.showDice(dice, value);
```

We pause the program for one second (one thousand milliseconds) so we have some time to read the screen. If this line is removed, things would go really fast.

```
20 |      Thread.sleep(1000);
```

We tell the computer where the part of the program that must be repeated indefinitely ends. And then we say we have also reached the end of the main part of the program.

```
21 |      }  
22 |  }
```

The main method asks the computer to create and display pseudo random numbers. It does not actually teach the computer how to create a pseudo random number. That is done elsewhere in the program in the function named `calculateNext`.

This function uses a mathematical formula to create a different value from the given value. It returns that new value. That new value looks apparently very different and unrelated to the previous value – it looks random.

```
24 |  public long calculateNext(long value) {
```

That's how the next value is calculated – by multiplying the previous value by itself.

```
25 |      value = value * value;
```

And the function ends by returning the new value it has just calculated.

```
31 |      return value;  
32 |  }
```

We calculated a large number that looks random. We are going to use some Mathematics to turn it into a small number that is between one and six. We create the function named `dice`. Given our large value the function returns a

small number between one and six. As we have seen earlier, ‘value % 6’ is read ‘value modulo six’. That looks complicated but all it does is removing six to the value as many times till the result is lower than six – between zero and five. After that we add one, and voila.

```
34 | public int dice(long value) {  
35 |     return (int) (value % 6) + 1;  
36 | }
```

We have simulated a dice that has six faces – one to six dots.

See also the following related activities

- Monte Carlo
- Random Walk

To go further

- 1654 - Blaise Pascal and Pierre de Fermat study gambling problems and develop the mathematical theory of probabilities. Prior to their work, most people played dice and card games without really being aware of the odds of winning or losing.
- 1687 – Isaac Newton publishes the foundations of classical mechanics. The world is mechanical and as long as one knows accurately its state at a given moment, one can predict any future state. There is no room for true random behavior.
- 1925 - Heisenberg and Schrödinger develop the foundation of modern quantum mechanics, along with Albert Einstein and Niels Bohr. Unlike what Mr. Einstein would like to believe in, it now appears that God does play dice with the universe. Certain natural phenomenas are truly random such as the emission of neutrons by radioactive materials. The emission of a neutron at a particular time from the radioactive source can happen at any interval of time independently of when was the last time the source emitted a neutron. Yet at large scales, the average emission of neutrons can be measured. But at the scale of a single neutron, it is completely random. Such fundamental random phenomenas are now used to manufacture random number generators. These devices can be connected to a computer so that a program can ask the device for true random numbers.
- 1946 - John Von Neumann develops the Middle Square method to rapidly produce series of pseudo random numbers on the ENIAC computer (to run Monte Carlo simulations). Mr. Neumann is a

mathematician who contributed to the development of the atomic and hydrogen bombs at Los Alamos in the United States.

Exercise 1: Create different pseudo random numbers every time the program is executed

Run the program twice. Notice each time the same series of numbers are displayed. To remedy to that, replace the line

```
13 | long value = 12345678;
```

With that line

```
13 | long value = System.currentTimeMillis() % 100000000;
```

Run the program twice again. What do you observe?

Now the computer is getting the exact current time (given in milliseconds) at the moment that part of the program is executed. It computes the first pseudo random number with eight digits. Every time you run the program the time is a little bit different so that number is also different. Each time you get a series of pseudo random numbers that look different from the previous execution.

Exercise 2: Modify the program to simulate Lotto 6/49

Modify the program so that it calculates and displays six numbers between one and forty nine.

Computer executes the program PseudoRandom2.java

Lotto: 3 (from calculated pseudo random value 72868392)



First edit the function dice.

```
34 | public int dice(long value) {  
35 |     return (int)(value % 6) + 1;  
36 | }
```

Replace the number 6 by the number 49.

```
34 | public int dice(long value) {  
    21
```

```

35 |     return (int) (value % 49) + 1;
36 | }

```

Second, edit the function paint. Replace the line below



```

79 |         drawDice(g2, i, j, dice, value,
        isCurrentDice);

```

With the following line.



```

79 |         drawLotto(g2, i, j, dice, value,
        isCurrentDice);

```

Find the last three lines of the function drawDice.



```

116 |         g.fillArc(x + 2 * r, y + r, r, r, 0, 360);
117 |     }
118 | }

```

Add the function drawLotto after the function drawDice.



```

119 |
120 |     void drawLotto(Graphics g, int i, int j, int
dice, long value,
121 |         boolean isCurrentDice) {
122 |         int width = 80, height = width, r = width / 3;
123 |         int x = (width + r) * i + width;
124 |         int y = (height + r) * j + height;
125 |         g.setFont(new Font("Tahoma", Font.PLAIN, 32));
126 |         if (isCurrentDice) {
127 |             g.drawString("Lotto: " + dice + " (from
calculated pseudo random value "
128 |                 + value + ")", 0, 30);
129 |             g.setColor(Color.ORANGE);
130 |             g.fillRect(x - r, y - r, width + 2 * r, height
+ 2 * r);
131 |         }
132 |         g.setColor(new Color((int) value));
133 |         g.fillOval(x, y, width, height);
134 |         g.setColor(Color.BLACK);
135 |         g.drawString("" + dice, x + r, y + 2 * r);
136 |     }

```

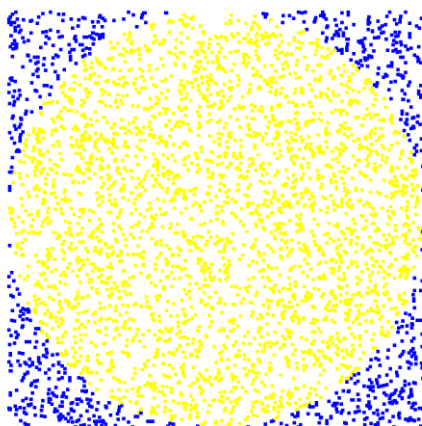
Run the program. It should display series of six numbers between one and forty nine. If you have completed Exercise 1, you will get different Lotto results every time you run the program.



Activity 2 - Monte Carlo

Alex had been showing a photo album with his family in the city of Monaco. “That was the one time we went to the grand casino of Monte Carlo, we had a really great time” He said. “We had won that trip because my boss was so happy about my new method to predict financial markets based on a Monte Carlo simulation. In one way, you could say he had a certain sense of humor.” Sarah didn't see the point. “Why is it funny? What do you mean?”

Computer executes the program MonteCarlo.java



Pi: 3.15

Points in square: 4000

Points in circle: 3150

“Monaco is a city state along the French Riviera. It was the first time we were visiting that area. Monte Carlo is a part of Monaco and is notorious for its casino and exuberant life style. At the time I was working for a large swiss insurance company. My boss was really happy because I had adapted recent mathematical techniques to estimate the risk of certain accidents to happen. It had given good results and the insurance had been able to price accurately its customers to cover certain risks and had made good numbers out of it. I had used something related to Monte Carlo simulations. So my boss sent me to Monte Carlo. Simple.”

“A Polish mathematician named Stanislaw Marcin Ulam had promoted a mathematical technique to compute surfaces, volumes (or integrals if you prefer). It was making extensive use of random numbers. His uncle was a big

gambler, so that technique was named the Monte Carlo simulation. In reference to the famous casino in that city.”

“Now that probably all sound scary but it's extraordinarily easy to get a feeling about what it is about. We will use random numbers to compute the number Pi!”

“No way!” Mike was incredulous. “Pi has nothing to do with random numbers.”

“So what is Pi about, Mike?” Alex asked.

“Well it gives the length of the circumference of a circle of a particular radius. The circumference is Pi multiplied by the radius, multiplied by two.”

“Very well. You knew that too Sarah isn't it? Otherwise I can have you measure the circumference of this camembert box - which as anyone knows is circular - and evaluate Pi by dividing the circumference by the diameter of the box.”

“I knew about it Alex. We did that experience at school. Only we did not use camembert boxes but bagels.”

“I see. And what about the surface of a circle?”

“The surface of a circle is equal to Pi multiplied by the square of the radius.”

“Good Sarah. So if we take a circle whose radius is one meter, then what is its surface?”

“Pi square meters, I suppose. Because it is Pi multiplied by one multiplied by one. So it is Pi itself.”

“Indeed. Let's draw that.”

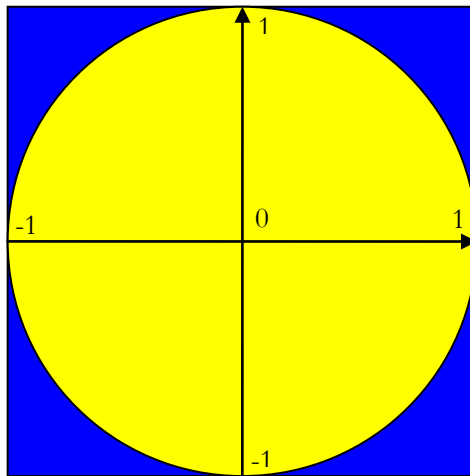


Figure 1 - Circle inside a square

“Sarah, please divide the circle in four equal parts. Now consider the upper right portion. It is included in a square whose side's length is equal to the radius isn't it?”

“Yes it is. The whole circle is inside a square whose side's length is twice the radius. That portion of the circle is inside a square four times smaller.”

“How far is a point on that portion of the circle from the origin of the circle?” Alex asked.

“That's easy. It's equal to the radius, obviously. It's a circle, every point on the circle is distant from the center by a distance equal to the radius.” promptly answered Sarah.

“And every corner of our square that comprises our portion of the circle?”

“The first corner is on the center. Distance is zero. The upper corner and right corner are both on the circle, so distance is one meter. The upper right corner, it's a little bit more difficult.”

Mike was fumbling. “Yes, I remember this. It's Pythagoras, right? It's like the diagonal of an isosceles rectangular triangle, so the length is given by the square root of the sum of the sides, so the square root of one plus one. Square root of two. It's about one point four meter.”

“Excellent, Mike. So the area of that square is one square meter. The area of that portion of the circle is ...?”

“It's a quarter of the full circle. So it would be Pi divided by four.”

“But Alex, where are you going with all this?”

“We are almost there, a little bit more patience. Now imagine I randomly pick a point within that square. I.e. its distance to the left corner is less than one. Its distance to the bottom corner is less than one. Then it can be anywhere on the square isn't it? We will call these distances x and y from now on.”

“Yes. Every point in the square verifies that.” observed Sarah.

“Now some of these points may or may not be also inside the circle. Can you tell me how I could verify?”

Alex paused for a moment. The teens were thinking this through carefully.

“The distance between that point and the center of the circle has to be lower than one. Otherwise it is outside of the circle. And that distance, we have it. As Mike explained earlier, it is given by Pythagoras. Compute the square of x , the square of y , add them, and take the square root. That's it. That gives you the distance. If it is lower than one, then it is within our portion of the circle. Otherwise it belongs exclusively to the square.”

“And why do we care about it?” asked Mike.

“You take one hundred random points both on the square. It's random, they are anywhere on the square. You check which points are inside the circle as well. Not all of them are. In average, how many points do you think will also be on the circle?”

“Well it's random. It depends.”

“Yes it depends. On the size of the surface of the circle. If the circle had the same size as the square, every point on the square would also be on the circle. If the circle was covering exactly half the surface of the square, it would contain half of the points. But you can even see from that figure, the circle's surface seems a little bit bigger than half of the square. As a matter of fact, that quarter of circle covers a little bit less than 80% of the square.”

“We said the surface was Pi divided by four. So it is roughly 3.14 divided by four, about 0.785 - 0.8 square meters. And the square itself is exactly one square meter. So it is 78.5 percents of the square. Oh I see how you got the 80% then.” said Mike, helped with a little calculator.

“That feels about right on the figure” said Sarah.

“I still don't know where we are going with this?” asked again Mike.

Alex was almost done.

“You take one hundred random numbers. You count how many of them are on the circle. That count is given to you by the simulation - our own Monte Carlo simulation. That count divided by one hundred is equal to Pi divided by four. The number of points on the circle divided by the number of points on the square gives the ratio of the surface of the circle with the surface of the square.”

“Oh!” “Neat!”

“Now Mike and Sarah, I need to take a rest. Make a program that uses random numbers between zero and one to compute Pi.” On this, Alex left to read his newspaper.

Liliane's notes

The program is named MonteCarlo.

```
9 | public class MonteCarlo {
```

Program always starts from the main part.

```
11 | public static void main(String[] args) {
```

We create our new Monte Carlo simulation.

```
12 | MonteCarlo monteCarlo = new MonteCarlo();
```

We request a portion of the screen to draw our simulation.

```
13 | Screen screen = new Screen();  
14 | screen.setVisible(true);
```

The lines that follow will be repeated for ever.

```
15 | while (true) {
```

A point is randomly picked anywhere in a square of width equal to two.

```
16 | monteCarlo.addRandomPointInSquare();
```

That point is drawn onto the screen. It is shown in yellow if it is inside the circle comprised in the square and in blue otherwise.

```
17 | screen.draw(monteCarlo);
```

Simulation repeats itself for ever. It picks up another random point, draws it, etc.

Let's review in more details certain portions of the program.

The function `isPointInsideCircle` answers 'true' or 'false' if a point of coordinates `x` and `y` belongs to a circle of radius one. It calculates the distance of the point to the center of the circle and verifies it is less than one. If it is greater than one the point is outside of the circle and the relation is false. That allows us to decide if the point is yellow or blue.

```
34 | boolean isPointInsideCircle() {  
35 |     return x * x + y * y < 1;  
36 | }
```

The simulation picks random points inside the square of width two i.e. with a surface equals to four. It counts how many of these points also fall inside the circle comprised in that square. This gives us enough information to estimate the surface of the circle using the formula below. Since the surface of a circle of radius one is equal to π our simulation allows us to measure π . We slowly get a better approximation of π as we try with more and more random points.

```
54 | double getPi() {  
55 |     return 4 * nCircle / nSquare;  
56 | }
```

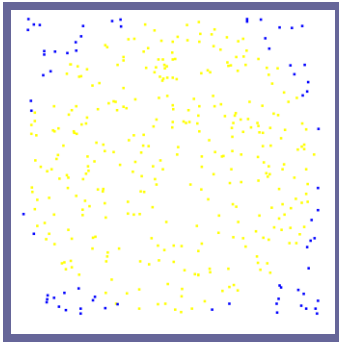


Figure 2 - 400 random points

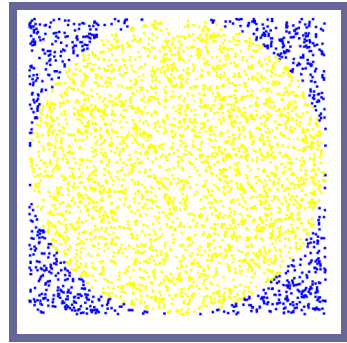


Figure 3 - 4000 random points

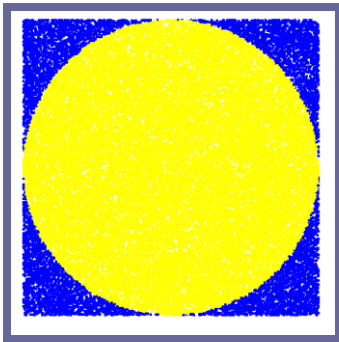


Figure 4 - 40,000 random points

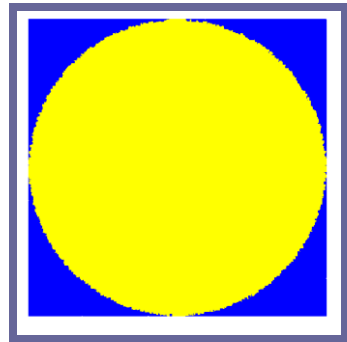


Figure 5 - 400,000 random points

We observed the simulation finds different values for π . Simulation is increasingly accurate with large quantities of random points. On Figure 1, we barely recognize the shape of a circle. The circle is far clearer on Figure 2, 3 and 4. With one hundred random points inside the blue square, eighty one points were also part of the yellow circle. That corresponds to a value of π of 3.24. With more random points, our estimate of π gets more and more accurate but you can notice it is slowly converging – with our last simulation with one hundred millions points we have three decimals of π correct.

Table 1 - Simulation results

Random points inside square	Random points inside circle	π (π)
400	303	3.03

4,000	3150	3.15
40,000	31,220	3.122
400,000	314,067	3.14067
4,000,000	3,142,321	3.142321
40,000,000	31,415,292	3.1415292
400,000,000	314,158,451	3.14158451

Later in the evening Alex explained a few more things about Monte Carlo simulation. “In this case, really what we did was use random numbers to estimate the surface of a circle. It is a simple problem, because it is very simple to calculate the surface of such a geometric shape. Now imagine you have a much complicated shape, very much more irregular. Imagine it is not a surface, but maybe a volume. Or maybe even more than a volume, it needs more than three parameters to be represented.”

Sarah had a little trouble imagining what could be more than a volume.

“Well you can still use the same technique. Get random points - with three parameters or as many parameters as needed - and check if they belong to that shape or not. The amazing thing is this technique gets really efficient when there are very complicated shapes and many parameters. Because it is still just about computing random numbers and counting how many belong to the shape. Monte Carlo simulations are widely used for much more complicated problems than the one we discussed here tonight. Thanks to it, we got to see Monte Carlo city. I'm glad Mr. Ulam and his friends picked this name. And now, it is time to go to bed.”

See also the following related activities

- Pseudo Random
- Random Walk

To go further

- 1866 – Monte Carlo is named after Prince Charles III of Monaco. Monte Carlo means Charles’ Hill in Italian. It is a small and luxurious resort area in Monaco located near the French Riviera in the south of France. Its casino attracts many tourists worldwide.
- 1901 – Mario Lazzarini throws a needle 3408 times on a floor covered by colored strips. He reported the needle crossed a strip 1808 times.

The length of the needle was $5/6$ times the distance between two strips. The needle can fall along any direction on the floor and π is equal to $5/3$ times the number of needles dropped on the floor by the number of times they crossed a strip. This is an example of Monte Carlo simulation without a computer, giving an excellent estimate of $\pi = 355/113 = 3.141592$. Value is actually so theoretically ideal it is believed Mr. Lazzarini lied about doing the experiment or that he dropped the needle as long as it was needed to obtain a result matching the already known value of π .

- 1938 – Enrico Fermi wins the Nobel Prize in Physics. Mr. Fermi used ingenious statistical random methods to calculate neutron properties. He later invented the Fermiac analog computer to study the evolution of nuclear systems with random numbers.
- 1940 – John Von Neumann develops the Monte Carlo method that applies large sets of random numbers to predict the average behavior of complex systems. These methods were used for the development of the first atomic bomb during the Manhattan Project and later for the hydrogen bomb. Rapid progresses in electronic computers were made after 1945 to support large and precise Monte Carlo simulations. Today Monte Carlo simulations are applied to many scientific areas and can use billions of random or pseudo random numbers.

Exercise 1: Draw different shapes

Program draws the random points inside the circle in yellow. It draws the other points in blue. Let's draw a shape that isn't a circle.

We need to modify the function `isPointInsideCircle()` which currently contains the following line.



```
35 | return x * x + y * y < 1;
```

This line tests if the random point of coordinates x and y is within a distance of one from the center, i.e. inside a circle of radius one. If the test is positive then the program will draw the point in yellow. If it fails the point appears in blue.

Let's modify this line with a different test. Then we will run the program and see how the new yellow shape looks like.

We start with a very basic shape. Coordinates x and y randomly vary between minus one and plus one. We want to test if a point belongs to a square whose points vary between zero and one.



```
35 | return x > 0 && y > 0;
```

Run the program. Can you confirm the yellow shape looks like a square?



Of course we no longer calculate Pi. We estimate the surface of the new shape. A square whose points vary between zero and one has a surface of one multiplied by one, i.e. one. Verify the program now displays a value close to 1 and not 3.14.

We can try more elaborate tests and draw complicated shapes. Edit and run the program for each mystery shape given below. Are you surprised by the appearance of the mystery shape three?

Mystery shape 1



```
35 | return x * x + y * y * y < 0.1;
```

Mystery shape 2



```
35 | return Math.abs(y) < Math.abs(x);
```

Mystery shape 3



```
35 | return Math.cos(10 * x) * Math.sin(10 * y) < 0.1;
```

Exercise 2: Draw your own shapes

Now you have seen enough shapes in the previous exercise. Try defining your own shape. It can be as simple or as complicated as you wish. Remember x and y vary between minus one and plus one. Store your two favorite ones below.

My own shape 1



```
35 | return _____ ;
```

My own shape 2



```
35 | return _____ ;
```

|

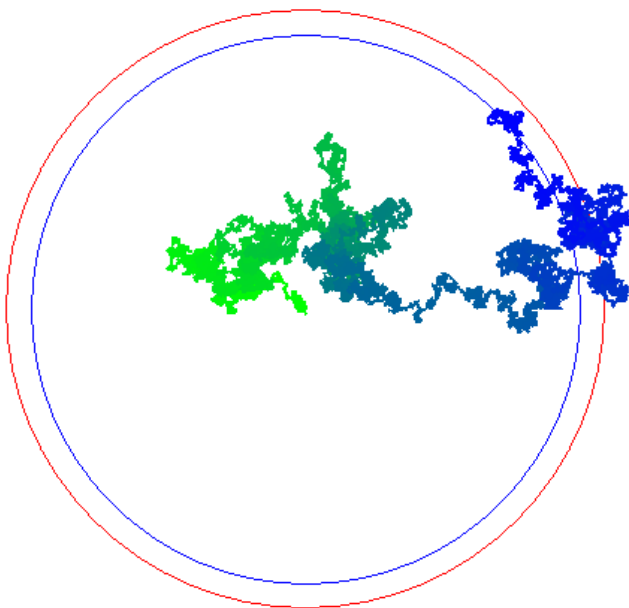
Mathematicians can spend a lot of their time defining and studying new shapes. They give them names. Sometimes hundreds of years later physicists or biologists discover a natural phenomenon that is well described by such shape previously invented by Mathematicians. Maybe your shape will become famous one day too.

Activity 3 - Random Walk

Alex, finally we are back!” shouted Sarah. “We got lost. Mike had no idea where he was going. He was just walking in every possible direction and we kept going into circles. Luckily, Liliane saw us and drove us back.” “Well I’m relieved the two of you are fine. We were getting seriously worried” responded Alex. “Drink some milk and take a rest. Then we will discuss your brother’s brownian motion.” “Brownian what?” uttered Mike and Sarah puzzled.

Computer executes the program RandomWalk.java

Random walk 39 (1000000 steps)



The two teenagers were sipping their chocolate milk after having a refreshing shower. They were ready to listen to Alex.

“Mike, imagine you are in a place that is flat and infinitely large. You can go in any direction. East, north, south or west. Or any angle in between.”

“Mike could be in the Prairies in Manitoba. It’s certainly large enough.” interrupted Sarah.

“Yes for sure Canada has the right size. Actually just imagine being in the center of a large skating rink like the one for the Olympics in Vancouver for 2010, Sarah, since I know you are very found of figure skating and Mike plays hockey.”

“What do I do in the rink?” volunteered Mike.

“You do one hundred steps. After every step, you turn into any direction anywhere around you. And you do another step. And so on till you have walked one hundred steps.”

“Can I run?”

“No. Every step you do must have the exact same length. Now Mike, after a hundred steps, how far are you from your starting point? I.e., how much distance have you actually moved away from?” asked Alex.

“Let’s say the length of a single step is one meter. So I go one meter in any direction. I have as much chance to go in one direction as in the other. So I’d say I stay pretty close to where I started from. In average I return to where I was originally.”

“No matter how many steps you do?” asked again Alex.

“I don’t think so. I don’t see it makes any difference.”

Alex paused. He wanted to make sure he had the attention of the two teenagers.

“What if I tell you that in average, after one hundred steps you’d be about ten meters away from your starting point? After one thousand steps, you’d be about thirty two meters away.”

The teens kept silence for a few seconds. Sarah was thinking for herself. “A thousand steps, that’s one kilometer. If Mike is only thirty two meters away from where he started after walking a full kilometer, that’s really tiring. Like this afternoon’s walk with Mike.”

“I don’t understand” said Mike. “I go backward or forward. So in average, it’s zero.”

“I didn’t say that in average you went thirty two meters forward. Or thirty two meters backward. I’m saying if you repeat that random walk many times, doing a thousand steps every time, you’ll find that in average you are thirty two meters away from your starting point. For a given walk, the result will unlikely be thirty two. It will vary greatly - maybe some times you’ll be three hundred meters away from your starting point or just five meters. In average, for many random walks, you’ll find the average distance you covered isn’t zero. It’s thirty two meters.”

"The actual average distance for a large amount of steps is given by the square root of the number of steps", Alex was visibly thrilled. "And we can write a simple program to see that."

"Will we use our pseudo random generator to get random directions?" asked Sarah.

"Good point Sarah. We could. We can try. Now our random generator is a little bit too limited for our needs here. We want to simulate a really large number of random steps. So instead we will use a more sophisticated pseudo random generator that any computer program can use. Then we will see what distance Mike actually covers."

"Let's program this. It draws beautiful figures on the screen of the computer." continued Alex. "You randomly pick up an angle between 0 and 360 degrees to decide where your next step will be. So you need to pick up a random number between 0 and 360. After a little bit of work, the program gives you these lovely animations of particles randomly walking on a screen. Let's walk a million steps each, to get some nicely chaotic images."

Sarah blinked her eyes rapidly. Alex had lost her but Mike seemed to be fine with Alex's explanations. And the figures were pretty regardless.

They started the program.

Mike frowned. "I knew it. It's not even close to thirty two meters at all. It's random."

"Correct Mike", told Alex. "That was one walk. Notice the distance isn't zero either. To be able to find the average distance we computed mathematically, you'll need to do many walks and average each distance."

Sarah changed the program setting it to perform a hundred random walks. Each random walk comprised one thousand random steps. That meant a million steps - or a million meters - had been walked by Mike. Quite a distance.

She had stored the sum of each distance actually covered from the initial to the final point for each walk. She divided that sum by the number of random walks.

"Thirty two meters" Mike read with a faint voice. "Amazing."

"Alex, you mentioned earlier something about brownian motion. What is it?"

"I nearly forgot about that part. That's a neat story. In 1827, a botanist named Robert Brown observed pollen particles floating in water under a microscope. He observed what looked like random movements of the particles in the fluid. He tried with different kinds of particles - dust - and noticed the same thing. It was found that the random walk model could be used to explain how far these particles could move in average. Of course Brown could not follow every little movement of the particle - it's happening way too fast and the steps are way too small to be observed with an optical microscope. But he could measure every minute by how much distance the particle had moved."

"But in water, the particle does not just move like a skater on a rink." pointed Sarah.

“However if you examine a thin layer of water, then it is like a flat surface. The particle can go to the north, east, south or west - or any angle in between - with the same probability. That can be described with a random walk in two dimensions. It’s like having Mike walk on a giant skating rink randomly.”

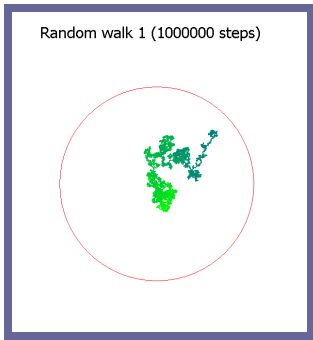


Figure 1 – First random walk has started

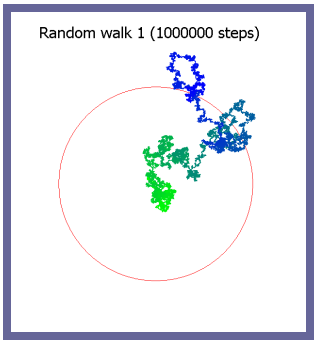


Figure 2 – First random walk completed

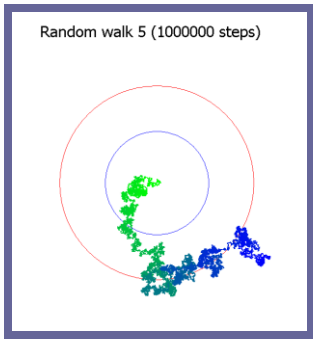


Figure 3 – After five random walks

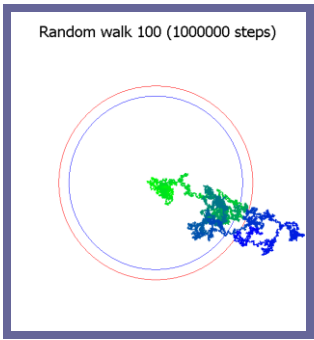


Figure 4 – After one hundred random walks

Mike and Sarah run the program many times. They noted how varied each random walk was on the screen. Yet you could precisely extract the average distance actually covered after many such random walks. That was really mind puzzling. The circle representing the theoretical average distance and the circle representing the distance from the simulation were often close to each other.

“When the microscopic borders with the macroscopic” whispered Sarah.

“What did you just say?” enquired her brother. “Nothing. Just thinking how Mr. Brown must have felt when he was observing pollen under his microscope...”

Liliane's notes

Program starts in the main part. Our simulation will run one million steps for every random walk.

```
11 | public static void main(String[] args) throws  
    | Exception {  
12 |     int steps = 1000000;
```

We are asking the computer to give us a portion of the screen to draw any graphics we want to it. It comes initially blank on the computer's screen. We will draw the path of a person randomly walking on that surface.

```
13 |     Screen screen = new Screen(steps);  
14 |     screen.setVisible(true);
```

The expected average distance from the starting point after walking a million steps is equal to the square root of the number of steps.

```
15 |     double theoreticalDistance = Math.sqrt(steps);
```

We will store the actual distances after we complete each random walk. So we can compare the theory with our simulation.

```
16 |     double sum = 0;
```

We will do one hundred walks. We therefore want to repeat the lines coming after this line a hundred times.

```
17 |     for (int i = 1; i <= 100; i++) {
```

Prepare a new random walk.

```
18 |         RandomWalk randomWalk = new RandomWalk();
```

Calculate the average distance based on all the random walks we have simulated so far.

```
19 |         double measuredDistance = sum / (i - 1);
```

Draw on the screen the walk number and two circles represented the theoretical average distance and the current average distance from our simulation.

```
20 |     screen.showWalk(i, theoreticalDistance,  
    | measuredDistance
```

Walk the requested number of steps (one million here) and draw them on the given screen. Return the actual distance from the initial point after all the steps were completed.

```
21 |     double distance = randomWalk.walk(steps,  
    | screen);
```

Add this result to our sum so we can compute a better average distance (because it includes a larger number of random walks so it is more accurate).

```
22 |     sum = sum + distance;
```

Pause the program for a second (one thousand milliseconds). Give some time to the user to appreciate the graphics on the screen before moving on to the next random walk. That's what the main part of the program does.

```
23 |     Thread.sleep(1000);
```

See also the following related activities

- Pseudo Random

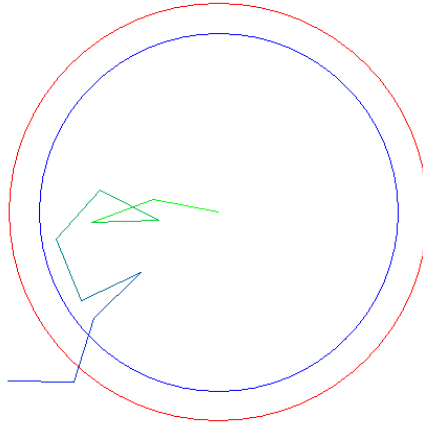
To go further

- 1827 – Botanist Robert Brown observes the random movement of pollen grains floating in water under a microscope. In 1905 Albert Einstein explains each grain is bombarded by the much smaller and faster moving molecules of water. As a result grains keep on moving into random directions.
- 1991 – Pierre-Gilles de Gennes is awarded the Nobel Prize in Physics for his work on polymers and crystal liquids. A polymer is a very long molecule. Its long chain of atoms can be described by a random walk.

Exercise 1: Modify the program to do a small random walk of ten steps

Modify the program so that each random walk ends after ten steps. The initial program does a very long walk of one million steps. A smaller walk allows us to see each step clearly on the screen.

Computer executes the program RandomWalk1.java



In the main part of the program, modify the value assigned to steps at the line below.

```
12 |    int steps = 1000000;
```



Use the value 10.

```
12 |    int steps = 10;
```



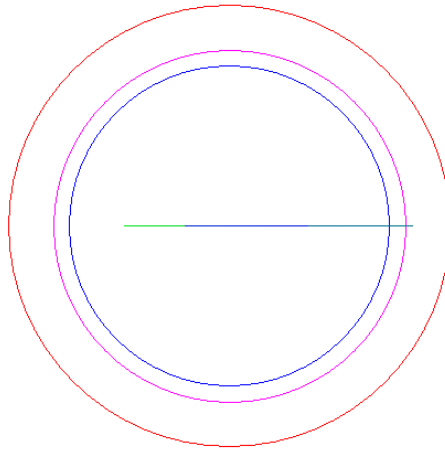
Run the program. The walk now looks clearly segmented – each step is large enough to appear on the screen. Remember the walk starts from the center. Walker picks up a direction randomly then does one step in that direction. After ten steps we measure the distance we are from the center of the screen (it is usually far less than the total length of the walk itself since we walk in any direction). In average, we are about three straight steps away from the center (square root of ten is about 3.16). The red circle shows that theoretical average. Blue circle shows the average distance considering all the random walks we have done so far. It doesn't matter if we walk ten steps or a million steps – the average distance from the center is still given by the same formula, the square root of the number of steps.



Exercise 2: Modify the program to do a random walk along a straight road

Modify the program so that the walker randomly does either one step forward or backward – along the same path or line. This simpler kind of random walk comes with an intriguing surprise...

Computer executes the program RandomWalk2.java



Edit the walk function. Replace the lines below



```
31 | double angle = random.nextDouble() * 2 * Math.PI;
32 | double dx = Math.cos(angle);
33 | double dy = Math.sin(angle);
34 | screen.showStep(i, x, y, x + dx, y + dy);
35 | x = x + dx;
36 | y = y + dy;
```

With the following lines.



```
31 | double dx;
32 | boolean goForward = random.nextBoolean();
33 | if (goForward)
34 |     dx = 1;
35 | else
36 |     dx = -1;
37 | screen.showStep(i, x, y, x + dx, y);
38 | x = x + dx;
```

We get a value that is set randomly set to either 'true' or 'false'. It is like flipping a coin, tail or face. If it is set to true, we move forward by one step. Otherwise we go backward one step. This time we only move along the horizontal axis.

Run the program. Do you notice that after a large number of these random walks the average distance (in blue) seems to be consistently lower than the theoretical distance?



Yes, there is a little secret. The theoretical average distance for random walks along a line is smaller than for the random walk in any direction on a plane surface. It is still proportional to the square root of the number of steps. But there is a coefficient. Square root of 2 divided by Pi. That's right. Pi is showing up in this problem. $\sqrt{2/\pi}$ is roughly equal to 0.8 which means we are in average 20% closer to the center than with the random walk on a plane surface.

Modify the program so that it shows a circle at that distance.

In the main part of the program, after the line below,



```
15 |    double theoreticalDistance = Math.sqrt(steps);
```

Add the following line.



```
16 |    double theoreticalDistance2 = theoreticalDistance  
    * Math.sqrt(2 / Math.PI);
```

That's the correct average distance for our random walk along a line.

Always in the main part, edit this line



```
21 |    screen.showWalk(i, theoreticalDistance,  
    measuredDistance);
```

So it looks like that line.



```
21 |    screen.showWalk(i, theoreticalDistance,  
    measuredDistance, theoreticalDistance2);
```

Edit the function showWalk below.



```
62 |    public void showWalk(int count, double  
    theoretical, double measured) {
```

So it takes an extra argument theoretical2.



```
62 |    public void showWalk(int count, double  
    theoretical, double measured, double theoretical2) {
```

And finally inside the showWalk function, after the following line.



```
67 | showCircleAtDistance(theoretical, Color.RED);
```

Add an extra line to show the circle with the correct theoretical distance (magenta color).



```
68 | showCircleAtDistance(theoretical2,  
   | Color.MAGENTA);
```

Run the program. Observe how the average distance (magenta circle) is a little bit smaller than the circle for a walk on a plane surface. Look if the random walks actually match better this magenta circle (as they should).



Don't you find this curious? Walking randomly forward or backward takes you in average at a distance related to the number Pi. Mathematics is filled with many of such surprises.

Activity 4 - Translator

How many languages can you speak, Alex?” Asked Sarah. “Quite a few, Sarah.” “How did you learn?” “Well, I did not learn them at school, so to say. Rather, what we call the school of life. It's good to know your neighbors, and what better way than learning how to communicate with them, right?” Alex paused. “You can teach a machine how to translate, you know...”

Computer executes the program A.java

```
Please enter sentence:
ca va bien
Teach me: 'ca va bien'
it's going ok
Teach me: 'ca'
that
Teach me: 'va'
goes
Teach me: 'bien'
well
I now know 4 expressions
it's going ok
Please enter sentence:
salut
Teach me: 'salut'
hello
I now know 5 expressions
hello
Please enter sentence:
salut ca va bien
hello it's going ok
```

Alex was fluent in about four languages. He didn't think grammar was necessary to learn a new language, and Sarah heart fully agreed with Alex on this. He thoughts words and sentences were part of something broader. There was the body language. You could go really far in a foreign place just by smiling appropriately. After all, words could differ, but we all needed to eat, drink, go to work and sleep, enjoy time with friends. So we had much in common in what we liked to do, if not the words to express it.

That's also the way Alex was teaching programming at the summer camp. He didn't open a big book explaining minute details on computers. That's just tedious. Instead, he showed them how a few lines of code could get a computer to do something concrete and fun. It was not so important at first that Mike and Sarah understood every part of the programs. The same way a child repeat a dirty word he heard at school from someone else. And see what effect it has on his parents... Action, reaction. That's how we all learnt, for the pleasant and less pleasant things in our lives.

"At school, we use computers to learn French." explained Sarah.

"How does it work?"

"The computer reads a sentence. I have to repeat it and it asks me to repeat till I pronounce right."

"Sounds like fun?"

"At first it was. But after a while, it got boring - the computer always asked the same sentences."

"So you can't teach the computer new sentences?"

"No."

Alex told Sarah it is possible to teach computers how to do new things. That's what a program does. It is also possible to write a program that teaches the computer how to learn about something. First the computer does not know much about it. Then it is given more and more information, which it can use to do new things. A computer is really good at remembering large quantities of information. It can display the content of many books, books it would take a full school year to Sarah to learn about. Of course, that did not mean the computer understood what it was remembering. After all, the book had been written for humans, and computers don't necessarily care about human stories, why would they?

Yet, Alex explained, many scientists had tried to teach computers how to learn - something called artificial intelligence. Nobody really knew what intelligence was about. On the other hand, Alex knew what a translator needed to do to be useful. So he proposed Sarah to teach the computer how to translate English to French - or any language to any other language she wanted to. As a matter of fact, he was going to teach the machine how to learn what Sarah would teach him. Sarah said yes, but she did not immediately understand what Alex really meant there. They were now both sitting in front of the computer, and unusually, Alex was the one using the keyboard.

"Sarah, we want to create a program that turns our computer into a translator. What does a translator do? Have you ever seen a real translator when a president of a country meets the president of another country?"

"Yes, each president is sitting by a person who knows how to speak the president's language and the other president's language. When one president speaks, the other president's translator listens to it and whispered the words to the ear of the other president using that president's native language."

"In big meetings, each person has their own translator because they don't always trust the other translator."

"And because a translator is best at translating a foreign language back to that translator's native language."

"And when your daddy speaks one language, and your mother another one, what's your preference?"

"Indeed, that's getting complicated. So to get back to our translator program, what do we need it to do?"

"User needs to speak in the language to be translated into. The user types in a sentence he needs translated. Then the machine would translate and display the sentence in the other language."

"Correct. And what do you do when you hear a word you don't understand?"

"Well I ask for its meaning, I suppose."

"So our program will ask the user to provide the translation, if the program does not understand the sentence. The program will be able to learn new sentences, like you."

"So it's like discussing with the computer?"

"In some ways, it is. Like you are the teacher, and the computer is a student. If you are a good teacher who knows about the student's abilities, then the student can deliver its full potential."

Alex was now typing pretty fast on the keyboard.

"There is an important thing we have not discussed yet. You know what it is?"

"No."

"When you had a course with your teacher, you remember it, isn't it?"

"Well not all of it, but yes some..."

"So what we teach the program, the program must remember it. Even after the computer is shutdown. Otherwise, without remembering, the program will never really learn - it will always be starting from nothing every time."

"How do we do that?"

"Your brain remembers past conversations. The computer can store information in places where it knows how to get back to it later. This old computer uses these pieces of magnetic and plastic discs. Others send the information to other machines that have a lot of memory. Every computer knows how to do that, so the details for now are not very important. What we need is to remember all the sentences we taught the program in one language and their translation into the other language."

"So the program can search through that to find the question and its answer."

"Exactly."

"Can this be big?"

"How big is your brain?"

"Pretty big."

"We're not sure how your brain works, but a computer can remember a lot of words. Now, there is another important thing to help the computer be a little bit

smart about the translation. Do you know how many words you need to communicate with other people?"

"A million words?"

"No! Actually, five hundred words get you going in regular situations like booking a hotel, going to a restaurant, buying some food. You're fluent if you know about five thousand words. And if you are fairly cultured, it is frequent to use about fifteen thousand words as part of your vocabulary."

"Is that too much for our program?"

"It's going to take a lot of time for you to teach the program all these words, but no that's not so much for a machine. The problem is elsewhere. Five thousand words is one thing. But how many sentences can you make with these five thousand words?"

"Oh, well... An infinity of sentences I imagine. There are lots of books out there in book stores, isn't it?"

"Precisely. So we just can't teach the program all the sentences in the world, because there are an infinite number of sentences..."

Yet Alex was continuing to type the program.

"So we can do what we frequently do when a problem is too complicated to be resolved directly. We simplify it. We divide it into smaller problems. If the smaller problem is still too big, we break it into even smaller problems itself. And so on. Till the problem is simple enough that we know how to resolve it. Then we assemble the whole solution from all the very small problems we identified. And we conquered the big problem!"

"What does that mean here?"

"We can't tell the program how each sentence ought to be translated into another sentence. But a sentence is constituted of words..."

"We teach the program how to translate a single word, and then the program will be able to translate every word of a sentence entered by the user?"

"Yes, that's the idea. And that's how the very first translators were working."

Alex chuckled. "Needless to say, these translators were not working too well."

"Why is that?" Sarah asked.

"Consider how such program would translate into French: "what you say is music to my ears?". It would be something like "quoi tu dire est musique vers mes oreilles". To a french speaker who is not familiar with this english saying, this sounds very poor french and quite puzzling. Maybe that person would guess you want to listen to music and would offer to turn on the radio. That's a faux-pas..."

"A real translator would instead say something like "C'est exactement ce que je voulais t'entendre dire" i.e. "It is exactly what I wanted you to say"."

"Yes. And if you pick up languages spoken by civilizations very remote from one another, the harder it is to adapt the original meaning to the new environment. You have to get close to it, but there is no such thing as right. The translation will evolve along with the cultural exchanges between the two

communities. Such situations occurred during the silk road when occidentals and orientals started to trade with one another over a thousand years ago.”

“So to give a chance to our translator to handle idioms that should not be translated word by word, here is what I propose. The program can be taught individual words. It can also be taught how to translate a group of words - call it a sub sentence or an idiom.”

“A sentence to be translated is made of words. The program checks if it knows how to translate the whole sentence. If it does, then the translation should be very good - it was probably an idiom previously taught. If we don't know how to translate the sentence as a whole, we break it into words. We try to translate large groups of words together - in case the sentence is made of expressions that we learnt about. And we continue till down to single words. If there are words we don't know about, or series of words, then the program shall ask the user to teach it their translation. The program will store that new information so that next time it does not have to ask again. And the program puts back together all the different words and idioms that it finds translations for and returns it in the right order.”

“This sounds complicated?”

“So let's play with it. I should have it working by now.”

Sarah started the program Alex had finished writing.

“Please enter sentence” the program asked. Sarah enters 'ca va bien'.

“Teach me 'ca va bien’” the program said. Sarah translated 'it's going ok'.

“Teach me 'ca'.” Sarah entered 'that'

“Teach me 'va'.” Sarah entered 'goes'

“Teach me 'bien'.” Sarah entered 'well'.

The program was then ready for another translation.

“Please enter sentence.” Sarah reentered 'ca va bien'

“it's going ok” said the machine. Note it did not do a word by word translation which would have yield 'that goes well'. It first tried to match the largest group of words it knew about.

“Please enter sentence.” Sarah entered 'salut'.

Teach me 'salut’” the program said. Sarah entered 'hello'.

At this moment, there was a power outage. Alex went for the candles he was leaving on the kitchen table. They looked through the windows and observed the whole village was in the dark. Alex said “there's probably a storm hitting the transformer on the other side of the mountain. There were some pretty dark clouds at the summit of the mountain this afternoon. I hope it won't take too long, it's about my news time on the TV.” Just as he finished speaking, electricity returned and the computer restarted.

Sarah restarted the program.

“Please enter sentence.” She entered 'salut ca va bien'

“hello it's going ok.”

Alex was pleased by what he was seeing on the screen. “I think I'm done here. Have fun, Sarah, your student is waiting for you.” The program had stored every word and group of words Sarah had taught him before the power outage. It had broken a complex sentence into smaller chunks it knew how to individually translate. Now Sarah had to patiently teach the program words and idioms to improve the translator.

Liliane’s notes

The program starts in the main part. First thing the translator program does is remember all that it previously learnt in past execution. Now the translator is ready to translate or learn new sentences.

```
10 | public static void main(String[] args) {  
11 |     Translator translator = new Translator();  
12 |     translator.read();
```

The program keeps on asking the user to enter a sentence.

```
14 | while (true) {  
15 |     System.out.println("Please enter sentence: ");  
16 |     String sentence = translator.input();
```

It translates the sentence from the user and displays it on the screen.

```
20 |     String translation =  
    translator.translate(sentence);  
21 |     if (translation.length() > 0) {  
22 |         System.out.println(translation);
```

In case the translation failed, the program notifies the user it doesn’t know these words. It then asks the user to provide the translation of these words so that it can learn and improve. Next time the program is restarted it will remember what it had been taught.

```
23 |         } else {  
24 |             System.out.println("I don't understand");  
25 |             translator.ask(sentence);  
26 |         }
```

The translator program first knows nothing. Through user interaction it builds up a dictionary – for every expression or word, it stores the corresponding translation. Next time the program restarts, it brings back the whole dictionary from the previous execution. So it gets better and better.


```
31 | Hashtable dictionary = new Hashtable();
```

The translate function is given a sentence to translate. It returns the translation. It uses the dictionary. First it checks the dictionary knows how to translate that exact sentence. Otherwise it breaks up the sentence into smaller groups of words and look up the dictionary for every one of them. If unlucky it attempts to translate word by word. And if a word is unknown to the dictionary then the translator asks the user to teach the program about that word.

```
41 | public String translate(String sentence) {
```

The function asks the user how the whole sentence should be translated. It also asks the user how expressions and words in the sentence get translated. It stores all these translations into the dictionary and save the dictionary so that it can be reused next time the program is restarted.

```
91 | public void ask(String sentence) {
```

The read function builds up the dictionary from all the translations learnt in past executions.

```
109 | public void read() {
```

The save function stores the dictionary somewhere safe so that even if the computer is shutdown the read function can rebuild it the next time the computer and the program are restarted.

```
118 | Public void save() {
```

Alex added a feature Sarah demanded insistently. If Sarah taught the wrong translation to the program, then the program would never have a chance to learn the correct word - it though it knew the word already. Alex modified the program so that if Sarah enters a word or a sentence starting with a question mark, then the program will ask Sarah for the meaning of the word following that question mark. Sarah found it very useful to teach new idioms to the program even after the program knew every word in the idiom.

```
17 |     if (sentence.startsWith("?")) {  
18 |         translator.ask(sentence.substring(1));
```

To go further

- 1954 - Georgetown University and IBM demonstrate machine translation of Russian sentences into English. Program runs on an IBM 701 computer. It knows 250 words and six grammar rules.
- 1982 – Makoto Nagao proposes the example-based machine translation method. The machine is taught a list of examples paired with their translation. It combines and substitutes parts of these examples to translate complicated sentences.
- 1991 – Statistical machine translation is reintroduced by IBM Research after it was mentioned in 1949 by Warren Weaver. It uses very large collections of documents in both languages to determine which sequence of words is most likely to represent the translation of a particular text.
- 2006 – Franz Och announces Google Translation web site is better than past programs because it uses statistical machine translation based on billions of words of text and learning techniques. Machines can now learn from the millions of documents very well translated by human translators.
- 2006 – Mandarin Chinese is the mother tongue of more than a billion people, followed by English with half a billion people and then Hindi and Spanish (more than four hundred million people each). There are an estimated 7,330 main languages spoken in the world, and over 40,000 dialects.

Exercise 1: Teach Computer how to translate French into English

Daunting task, isn't it? Let's start with something you can use the next time you visit a good french restaurant or salon de thé.

Start the program. It asks you to enter a sentence.



Type in '**je voudrais une table pour deux personnes**'. Program doesn't know yet how to translate this sentence so it asks you 'teach me'. You start the training and enter the expected translation '**a table for two please**'. The program asks you the meaning of individual words as well. At some point it is satisfied it has learnt enough from this sentence. Program is ready to ask you for another sentence. Follow up the basic training indicated below. Enter the expressions from the column 'Human teaches the machine'.

Table 1 - French to English

Computer asks questions	Human teaches the machine
Please enter sentence:	je voudrais une table pour deux

Teach me: 'je voudrais une table pour deux personnes'	personnes a table for two please
Teach me: 'je'	i
Teach me: 'voudrais'	would like
Teach me: 'une'	a
Teach me: 'table'	table
Teach me: 'pour'	for
Teach me: 'deux'	two
Teach me: 'personnes'	people
Please enter sentence:	c'est par ici
Teach me: 'c'est par ici'	this way please
Teach me: 'c'est'	it is
Teach me: 'par'	by
Teach me: 'ici'	here
Please enter sentence:	je prendrais des coquilles saint jacques
Teach me: 'prendrais des coquilles saint jacques'	will have scallops
Teach me: 'prendrais'	will have
Teach me: 'des'	many
Teach me: 'coquilles'	shells
Teach me: 'saint'	saint
Teach me: 'jacques'	jack
Please enter sentence:	crème brûlée
Teach me: 'creme brulee'	burnt cream
Teach me: 'creme'	cream
Teach me: 'brulee'	burnt
Please enter sentence:	un
Teach me: 'un'	a
Please enter sentence:	et
Teach me: 'et'	and
Please enter sentence:	gateau
Teach me: 'gateau'	cake
Please enter sentence:	mousse au chocolat
Teach me: 'mousse au chocolat'	chocolate mousse
Teach me: 'mousse'	mousse
Teach me: 'au'	at
Teach me: 'chocolat'	chocolate
Please enter sentence:	pain perdu
Teach me: 'pain perdu'	french toast
Teach me: 'pain'	bread
Teach me: 'perdu'	lost
Please enter sentence:	flan
Teach me: 'flan'	caramel custard

Good job. It was hard work but isn't it comforting that you are training a machine to be useful? It is time to try out and order some delicious dessert.

Table 2 – Machine translation results

Human enters French sentences	Machine translates into English
-------------------------------	---------------------------------

je voudrais une crème brûlée pour deux personnes je prendrais un gateau au chocolat et un flan	i would like a burnt cream for two people i will have a cake at chocolate and a caramel custard
---	--

Not bad at all for a first lesson. You will have noticed the ‘cake at chocolate’ should be improved. Our computer is a good student and tries its best with what it knows so far. Word by word translation isn’t good enough in that case. Let’s refine the learning process. We want the program to ask us for the translation of ‘gateau au chocolat’. We do it by using the question mark followed by the expression.

Table 3 - French to English

Computer asks questions	Human teaches the machine
Please enter sentence: Teach me: 'gateau au chocolat' Teach me: 'gateau' Teach me: 'au' Teach me: 'chocolat'	?gateau au chocolat chocolate cake cake at chocolate

Now the next time we want to translate ‘gateau au chocolat’, computer will translate using the exact match ‘chocolate cake’. Not convinced? Let’s try again.

Table 4 - Machine translation results

Human enters French sentences	Machine translates into English
je prendrais un gateau au chocolat et un flan	i will have a chocolate cake and a caramel custard

There are no tricks. Computers are outstanding learning machines. The question mark is also useful to correct a mistake a human may have taught to the program. Now keep in mind all your efforts are not lost when you stop the program or shutdown the computer. Everything is stored in a file named ‘dictionary.data’. The program will read this file on startup and remember everything learnt so far.

Exercise 2: Stephen King’s novels in Spanish and English

Do you speak Spanish? A little bit? If you answered no then the game is even funnier. Can you guess the title of a novel written by Stephen King from its Spanish translation?

Let's play. Mr. King is a famous writer and his novels have been translated into many languages. We give a list of actual Spanish titles for ten of his novels below. Can you match them with their corresponding original English title?


Table 5 - Quizz

Spanish titles (guess English match)	Original English titles (different order)
1. los ojos del dragón	A. the shining
2. el cazador de sueños	B. the dead zone
3. desesperación	C. storm of the century
4. la zona muerta	D. desperation
5. el juego de Gerald	E. the eyes of the dragon
6. ojos de fuego	F. dreamcatcher
7. la mitad oscura	G. riding the bullet
8. montando la bala	H. gerald's game
9. el resplandor	I. the dark half
10. la tormenta del siglo	J. firestarter

Game - match Spanish and English titles

Let's teach the right answers to the computer. Give the Spanish title and provide the corresponding English title. What happens after you type in 'los ojos del dragón'? The program wants to learn every single word as well. It is too smart for this exercise.

Frequently engineers and programmers must reuse existing tools to resolve different problems. That's what we are going to do here. Let's remove lines of the program so that it no longer asks for the meaning of each word.

Lines 47 to 66 teach the computer how to split a sentence into words and search for sub sentences it previously learnt. Lines 96 to 103 ask the user for every single word in a sentence. 

Remove lines 96 to 103 and lines 47 to 66.



Run the simplified program.



We can now teach the titles to the program. Follow the instructions below.

Table 6 - Spanish to English

Computer asks questions	Human teaches the machine
Please enter sentence: 'Teach me: 'la zona muerta'	la zona muerta the dead zone
Please enter sentence: 'Teach me: 'ojos de fuego'	ojos de fuego firestarter
Please enter sentence: 'Teach me: 'los ojos del dragón'	los ojos del dragón the eyes of the dragón
Please enter sentence: 'Teach me: 'mitad oscura'	la mitad oscura dark half
Please enter sentence: 'Teach me: 'el juego de gerald '	el juego de gerald gerald's game
Please enter sentence: 'Teach me: 'desesperación'	desesperación desperation
Please enter sentence: 'Teach me: 'la tormenta del siglo'	la tormenta del siglo storm of the century
Please enter sentence: 'Teach me: 'montando la bala'	montando la bala riding the bullet
Please enter sentence: 'Teach me: 'el cazador de sueños'	el cazador de sueños dreamcatcher

Let's try out. Ask a friend to enter 'la tormenta del siglo. See below.

Table 7 - Machine translation results

Human enters Spanish title	Machine displays the English title
la tormenta del siglo	storm of the century

The machine displays the English title 'storm of the century'. It knows how to translate the titles from Spanish into English. You did a good work.

Remember to insert back the lines we removed in this exercise. It is a lot more fun to use this program when it is eager to learn every word. If you know multiple languages, try teaching the computer how to translate from one to the other. Sometimes the program surprises us by reusing portions of knowledge in a different context.

Activity 5 - Launcher

By now, Mike and Sarah had written a few programs. They wanted to be able to show them to their friends but it was not really easy to select and start a program. Alex told them it was a good time to write a launcher. A program whose job is to present to the user a list of applications available on the machine and execute the one selected by the user. How does a launcher work?

Computer executes the program Launcher.java

```
--- Launcher ---
0. PathFinder
1. WebBrowser
2. VectorGraphics
3. Translator
4. SlowFast
5. RandomWalk
6. PseudoRandom
7. MonteCarloGraphics
8. MonteCarlo
9. Launcher
10. Chess
11. SpaceSimulation

Please enter program number to execute:
11
```

Alex said it wasn't really nice to expect every person to learn how to program in order to start programs. Only programmers, teenagers and people curious about how things are done should worry about that. So how about writing a program that tells the user what programs are available? The user can then select the program to execute. Mike and Sarah agreed, it all sounded very simple when listening to Alex. They were in front of the computer, thinking on how they would do it.

“We need to create a program called Launcher. It will include all the other programs we have done.” was thinking Sarah.

“Yes, that's exactly my idea too. And we display the list of all these programs and ask the user to enter their choice.” responded Mike.

“And then, how do we start this program?”

Mike and Sarah were not sure. They glimpsed at their Alex, but he was comfortably seated in his armchair, reading his daily newspaper. Anyone who knew Alex knew better than disturbing him during such a ceremonial time.

“Well let's get started. Alex will tell us how to start a program. For now, we will just create our own method called execute. It won't do anything but Alex will tell us how to make it work.” Sarah pursued.

“You type or I type?”

“You type. I like to think.”

Mike did not answer to that. He could now type really fast and was very proud of it. Of course, that was a french keyboard, the letters were not all in the same location as the ones at his college. But once you knew how to type on one, it was fairly easy to adapt to another one, he thought.

About fifteen minutes later, our two hard-working friends had finished their orange juices, croissant and launcher program. The program looked pretty good. It displayed the list of their programs. User typed a number to select one entry. Then nothing happened because, well, they had to talk to Alex about that part.

Alex had finished reading the international news in his newspaper and was about to open the section on national policy. Good timing. He noticed two pairs of eyes aiming at him as he was folding the pages. “How is it going?” he asked.

They showed him their program. Alex read through it in silence, and then asked to see it run. When it was done, Sarah explained “we're not sure how to start the program.” Alex nodded. “You're very close. If the user selects program zero, then you can start the program MonteCarlo. To execute the program MonteCarlo from your program launcher, you call its method main. Every one of our programs has a method main, which is where the program starts. In your case, you are starting it yourself, so you call the main method yourself.”

Mike and Sarah wrote the execute method in their launcher program. If the user selected the MonteCarlo program, then they would call the main method of the MonteCarlo program. They just had to check and do the same for every other possible choice. The program was a little bit long - they wanted to be able to execute every one of their programs after all! But it worked just fine. Mike and Sarah thought they were done. Well, Alex wasn't done yet.

“So you are going to give your programs to your friends and show them how to run the launcher, right?” asked Alex.

“Sure. They will be quite amazed.”

“What if in a few more days you want to send them a couple of extra programs? Would the launcher work with these two new programs too?”

“Well, the launcher program doesn't know about them. Only about the programs we wrote till today.”

“So we would simply send a new launcher program that knows about these two new ones as well! Simple and easy.”

“Will you remember which programs you sent to each of your friends? So you can keep the list of programs that they can execute in your launcher?”

Mike could smell trouble. Alex was onto something, but what? What else could they do?

“When you buy a computer, it often comes with a few programs. But is that all that you will ever use?”

“No. Mike keeps on adding more programs that makes Dad very upset sometimes.” “Thanks, sister...” Mike was thinking.

“Precisely. Your computer shows you all the programs you had when you bought it. It also shows you all the programs you added later - which it didn't know about when the computer was constructed.”

“So there is a trick?”

“You may see it that way. A program can find out which other programs are currently available on a machine. There are different ways to ask for this information on different machines, but they pretty much all support this kind of service. So now we can write something very useful. The launcher program needs first to search for all programs that you wrote. Then, as you already did, it asks the user which program to execute. Then it executes that program.”

“I want you to think this through for a few moments. You write a program which does not know about specific programs - yet it can ask these programs to execute. It's a little bit like telling a person to do something for you. You don't know that person directly, you have never met. But you find that person's address in the yellow pages; you send that person a letter and some money. And that person will mail you back what you asked for. It does not matter who that person is in details - how she looks, her name. All that matters is that you know her address and she does the service that you need from her.”

“Here is how we are going to write the final version of the launcher. First we ask where we live. That is, where the launcher program is executed from. Then we ask for the list of all the programs near the launcher program. That's where your programs are. Now we have a list of files, we only want programs. So we only care about files that we can execute, that is, they have a main method. That's where it is really neat. You can ask a program if it has a main method! You can actually ask a lot of things about it. It's a little bit like Socrates who said 'Know who you are'. That's also called reflection. You know you are Mike; you have two legs, two arms and one head. You know how to run, swim and many other things. The same for programs. We can search for them, ask them a few things like 'are you a main program or are you there just to support other main programs?’”

“That sounds really cool.”

“Now we have found the list of main programs you wrote. We can show their name. Then we ask that program to execute its main method. And we are done.”

“And we are done.”

“You can remove programs. You can add programs. When you start your launcher, it gets the current list of programs that are available near it. It's always up to date! You don't need to worry about modifying your launcher to run future programs.”

Liliane's notes

The program starts in the main part.

```
9 | public static void main(String[] args) {  
10 |     Launcher launcher = new Launcher();
```

First we search for available programs – at the moment this program executes on the user's computer.

```
11 |     launcher.findPrograms();
```

Then we show the list of programs we have found on the screen.

```
12 |     launcher.display();
```

And we wait for the user to decide which one to execute. User simply types a number representing the program in the list shown on the screen.

```
13 |     int program = launcher.input();
```

Finally we execute the selected program

```
14 |     launcher.execute(program);
```

See also the following related activities

- Web Browser

To go further

- 1958 – John McCarthy invents the computer programming language Lisp. Lisp programs can easily introspect and modify themselves while they are executing.
- 1965 – Operating system named Multics is released. An operating system is a program that defines how other programs access various parts of a computer such as the screen, keyboard, memory and storage and how user can run other programs. Multics runs initially on the GE-

- 645 computer. User enters commands to the machine through a program called Shell.
- 1968 – Douglas Engelbart demonstrates revolutionary ways to collaborate with humans located at different places using computers connected to a network, with a graphic interface, simultaneously sharing and modifying documents with a new device: the mouse.
 - 1970 – Operating system Unics (later renamed Unix) is first used on PDP-11/20 computers. Derived from Multics, Unix favors breaking features in very small programs that can be combined together to realize more complex tasks. User also enters commands through a program called Shell to run other programs.
 - 1984 – First Macintosh computer is released. User interacts with the machine exclusively through a graphic interface and a mouse unlike Unix machines which rely on command lines through a Shell program.
 - 1991 – Tim Berners-Lee writes the first web browser. Web browsers have later become a popular way to access information and many services on the web such as news, weather, banking, etc.
 - 1993 – Microsoft releases the Component Object Model (COM). COM allows any program on the Microsoft Windows operating system to reuse other programs ('embed') which follow the COM format. A program can ask which COM programs are available ('interfaces') on a machine and dynamically call them (even if they were added to the machine after that program was created or installed). E.g. it is possible to write a program that shows documents reusing Microsoft Word or Internet Explorer components. Programs can become more modular, still well integrated and in theory able to work even when newer versions of Word and Internet Explorer are installed on the machine ('backward compatibility').
 - 1995 – James Gosling at Sun Microsystems releases the first version of the Java language and virtual machine. Java programs are portable. They can run on different operating systems because they are executed on a virtual machine that runs on top of each particular operating system. Other programs using languages such as Perl, Ruby, Smalltalk or Lisp can also work on various platforms.
 - 2005 – There are about 900 million personal computers used in the world and as many people using the Internet. Windows, Mac OS and Linux (derived from Unix) are the most widely used operating systems. Programs written for one operating system usually don't work on the other unless a special software adapter is used (called a virtual machine or emulator).

Exercise 1: Extend the Launcher program so that it can run multiple programs simultaneously

Run the Launcher program. Run one of the programs listed by the Launcher. The Launcher program runs that program. And that's the end of the story. If you want to start another program you need to restart the Launcher.



Let's modify the Launcher program so we can run multiple programs simultaneously. Because we all know computers know how to multitask even if it isn't always easy for us to do multiple things simultaneously...

In the main method we replace the following three lines.



```
12 | launcher.display();  
13 | int program = launcher.input();  
14 | launcher.execute(program);
```

By the five lines below.



```
12 | while (true) {  
13 |     launcher.display();  
14 |     int program = launcher.input();  
15 |     launcher.execute(program);  
16 | }
```

You notice the Launcher program never stops. It shows the list of programs, waits for you to choose one, executes it and repeats this sequence for ever.

The execute function must order the computer to start the program while rapidly returning to our Launcher application. We want the computer to run both the selected program and the Launcher.

In the execute function we replace the line below.



```
74 | main.invoke(null, args);
```

With this one.



```
74 | executeAndContinue(main, args);
```

Let's write the new function executeAndContinue. Insert the following lines after the execute function.



```
79 |  
80 | public void executeAndContinue(final Method main,  
   | final Object[] args) {
```

```

81 | Thread thread = new Thread() {
82 |     public void run() {
83 |         try {
84 |             System.out.println("Executing new program");
85 |             main.invoke(null, args);
86 |         } catch (Exception e) {
87 |             System.out.println(e);
88 |         }
89 |     }
90 | };
91 | thread.start();
92 | System.out.println("Continuing");
93 | }

```

Have you observed how our older code is still here at line 85? We are telling the computer to execute the program separately from the current Launcher application. So we can now use the same Launcher application to start as many programs as we want.

Our Launcher application is getting very interesting but it still lacks a little something. It doesn't look pretty. Let's work on this next.

Exercise 2: Display the launcher inside a window on the screen

It would be more convenient to have the launcher execute into its own window on the screen. User can then easily resize and move it like any other application. Let's add the lines below at the very top of the program. They are required to instruct the computer to create a window (also called a frame) and to detect when the user clicks their mouse in our window.

```

1 | import java.awt.Frame;
2 | import java.awt.event.MouseAdapter;
3 | import java.awt.event.MouseEvent;
4 | import java.awt.event.WindowAdapter;
5 | import java.awt.event.WindowEvent;

```



We edit the main method. It should now be like the following.

```

14 | public static void main(String[] args) {
15 |     Launcher launcher = new Launcher();
16 |     launcher.findPrograms();
17 |
18 |     Screen screen = new Screen(launcher);
19 |     screen.setVisible(true);
20 | }

```



The main method will open a window on the screen for our launcher application.

We can delete the function `display()`. We no longer print characters into the window of another program. We have our own. Delete the lines below.



```
54 | public void display() {  
55 |     System.out.println("--- Launcher ---");  
56 |     for (int i = 0; i < programs.length; i++) {  
57 |         System.out.println(i + ". " +  
    | programs[i].getName());  
58 |     }  
59 | }
```

The Launcher nicely draws the list of programs into its own window with the code below. Insert this code at the end of the program for example after the `executeAndContinue` function and just before the final line that contains a `}` character.



```
92 | static class Screen extends Frame {  
93 |  
94 |     Launcher launcher;  
95 |  
96 |     java.awt.List list;  
97 |  
98 |     public Screen(Launcher launch) {  
99 |         super("Launcher");  
100 |         launcher = launch;  
101 |         setSize(100, 300);  
102 |         list = new java.awt.List();  
103 |         add(list);  
104 |         list.addMouseListener(new MouseAdapter() {  
105 |             public void mouseClicked(MouseEvent e) {  
106 |                 int program = list.getSelectedIndex();  
107 |                 launcher.execute(program);  
108 |             }  
109 |         });  
110 |         addWindowListener(new WindowAdapter() {  
111 |             public void windowClosing(WindowEvent e) {  
112 |                 System.exit(0);  
113 |             }  
114 |         });  
115 |         for (int i = 0; i < launcher.programs.length;  
    | i++) {  
116 |             list.add(launcher.programs[i].getName());
```

```
117 |      }  
118 |      }  
119 |      }
```

This creates a window on the computer's screen. Window is filled with a list of programs' names. User clicks the mouse button over the program's name in the list. The Launcher gets the index of the program selected in the list and executes it. Prior to this exercise that number was entered by the user with the keyboard.

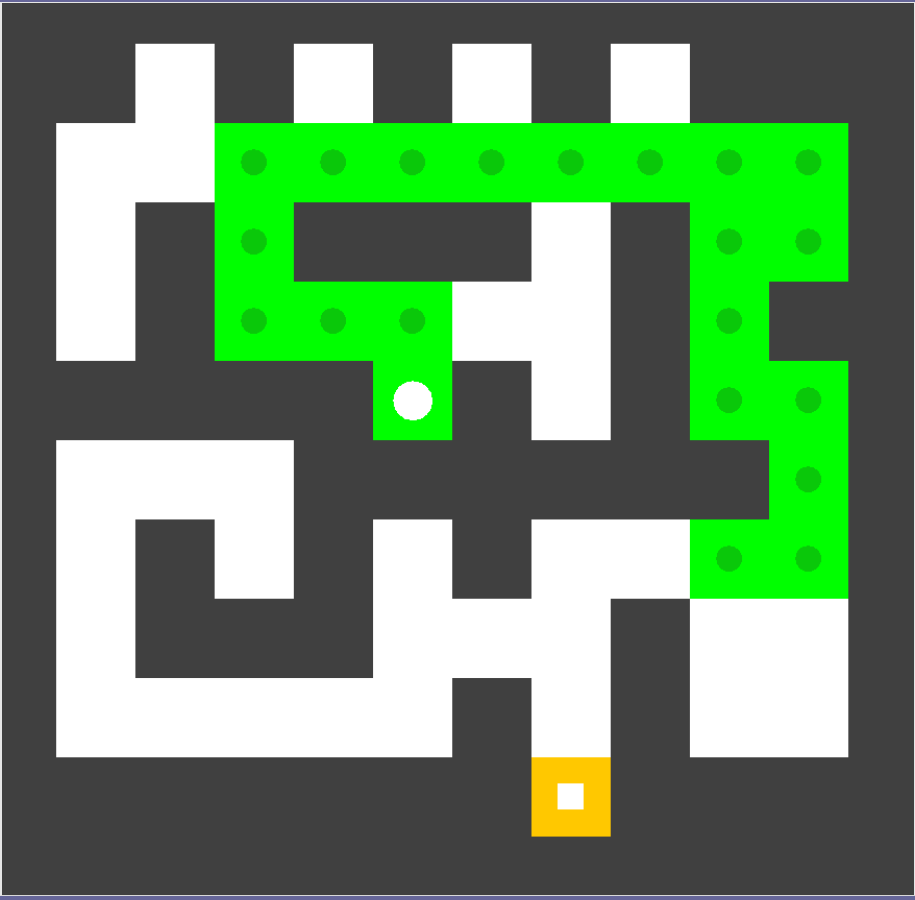
Run the program and click the mouse's button anywhere in the list. Nice work! It is now very easy for you and your friends to execute your programs.



Activity 6 – Path Finder

What will happen if we get lost in this forest?” asked Sarah to Alex. They were walking through a deep forest of pine trees picking up mushrooms. Sarah really had no idea how to return to the chalet.

Computer executes the program PathFinder.java



Alex reassured her they would be home very soon.

“Sarah, if you are lost, stay on your path and wait for people to search and look for you. You have a cell phone so the rescue team will be able to locate

you. Now in the imaginary world – the mathematical world – what would someone abandoned in a maze do to find the exit?”

Sarah jumped over a large rock before answering.

“You could do like the Little Poucet. Drop stones so you know where you come from. And explore the maze from there? With a little bit of luck or patience, one day you might get to the exit.”

“Yes, marking your path with stones is the right idea, Sarah. Walk along the path for some time till the path splits up into two or more other paths. There, leave a mark of your passage with a stone or a small branch. If that first path proves to be a dead end, turn back and try a new path. And so forth till you find the exit. That is also how a machine can be taught to find the path between two locations in a maze. I will show you when we arrive to the chalet in a few more minutes.”

It had been a good but long day hiking with Alex. Sarah felt asleep during Alex’s explanations. Liliane and Mike had stayed home and were very attentive. Alex told Mike video games relied on similar techniques to code complicated maps or mazes, catching the teen’s interest.

Sarah woke up on time for dinner. Liliane served a delicious omelet filled with the freshly picked mushrooms. Alex had invited his friend over. That friend was a pharmacist who had inspected the mushrooms with a professional eye. Everything was set for a good evening.

Liliane’s notes

The program behaves very much the same way a person left alone in a forest would find a way out. From the current position, look to the left, right, in front and behind you. Begin walking from the starting position (figure 1 and figure 2).

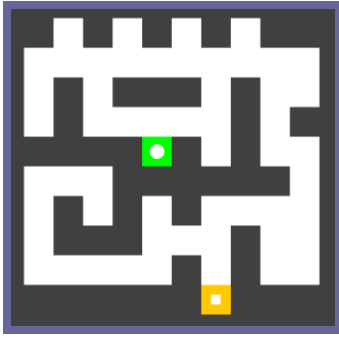


Figure 1 – Initial maze

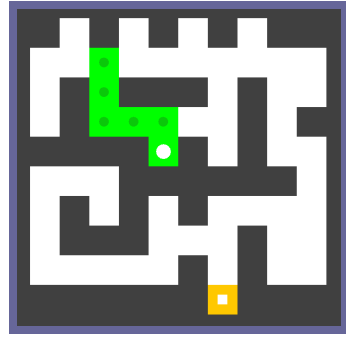


Figure 2 – Searching path, can go left or right

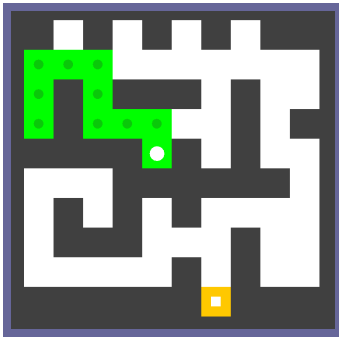


Figure 3 – Exploring left side and hitting a dead end

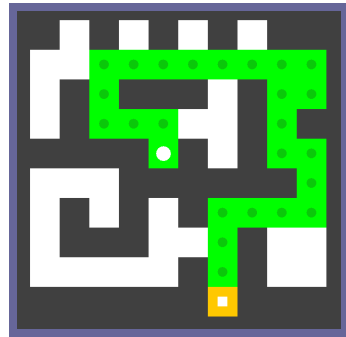


Figure 4 – Exploring right side and finding the exit

Leave a stone at every place you visit. You can also mark or color where you are. When there are more than one possible place to visit (figure 2), pick one (figure 3). At the new place, reexamine what paths are possible next. And so on. If you get lucky, you walk straight into an area that brings you to the exit. Most likely, you get blocked (figure 3). Return to where you come from and try from a new direction (figure 4). If the map is of finite size and if there is a path, you will find it (figure 4). It may take a lot of trials.

The program starts in the main part. It requests a portion of the computer's screen. That will be used to show the computer exploring the maze.

```

10 | public static void main(String[] args) {
11 |     Screen screen = new Screen();
12 |     screen.setVisible(true);

```

The maze is initialized. We will ask the program to pause for half a second – five hundred milliseconds - between each move. That will give us some time to see the progress made on the screen.

```
13 |   Pathfinder maze = new Pathfinder();  
14 |   int milliseconds = 500;
```

Program starts searching for the exit. Every half second it displays its current move to the screen. Program ends when all possible paths have been explored or the first time the exit is found.

```
15 |   maze.searchExit(screen, milliseconds);
```

Trails in a forest are stored in a map. The map is composed of squares representing the trails in the maze, borders of the maze, starting and exit in the maze. For the computer the map is a simple string named maze. Each character in the string represents a particular square in the map. First character in that string represents the top left corner of the map. Last character is for the bottom right corner. Character 'W' represents a wall. Blank character ' ' is a trail. 'S' is the starting point and 'E' the exit of the maze.

See also the following related activities

- Random Walk
- Graphics Vector

To go further

- 1736 – Leonhard Euler publishes a paper on the Seven Bridges of Königsberg. Mr. Euler proves an inhabitant of the city of Königsberg in Russia cannot walk and cross each of the seven bridges exactly once and return to the starting point. This paper is the foundation of graph theory.
- 19th century – William Rowan Hamilton invents a game in which the player must find a round trip which crosses every city positioned on a dodecahedron figure. This problem is later generalized and named the traveler salesman problem.
- 2004 – David Applegate and his team prove the shortest path to visit all 24,978 cities in Sweden is 72,500 kilometers long. They calculate the solution with a network of Linux workstations. At this time this is the largest ever traveler salesman problem that has been resolved.

Exercise 1: Flip flop start and exit entries of the maze

Edit the program PathFinder.java so that the starting point of the maze and the exit are replaced by one another. You need to modify the string maze in the program. Replace the character S (representing the starting point) in line 19 by the character E (representing the exit). Replace the other character E in line 20 by the character S.

Run the program again. Is the path found exactly identical to the path before start and exit entries are flip flopped?

Exercise 2: Create a very large maze

Create a very large maze by modifying the maze string in the PathFinder.java program at line 18. For example, create a maze of sixteen by sixteen blocks like the one below.



```
String maze =
    "S      W      W W      " +
    "WWW      W W      W W      " +
    "EW W W      W W W W      " +
    "  W W W      W W W      " +
    "    W      W W W      " +
    "W WW      W WW W      " +
    "  W WW W      W W      " +
    "W      W W W W W      " +
    "  W W WW      W W W      " +
    "  W W W      W W W W      " +
    "  W      W      W      " +
    "  WWW WWWWW WWWWWW      " +
    "    W      W W      " +
    "  W  WWWWWW W W      " +
    "      " +
    "  W      W      W      ";
```

Run the program.

This maze was designed to be challenging to the PathFinder program. A person would quickly find out a way out of the maze. On the other hand the program seems to be taking a long time going over many paths that look very similar to each other. The computer will try over one hundred thousand different moves before successfully reaching the exit.

Stop the program. We definitely need to edit the program so that it runs full speed. The easiest thing to do is to edit the following line in the main part.



```
14 | int milliseconds = 500;
```

So the program no longer waits half a second after every move.



```
14 | int milliseconds = 0;
```

To be certain to run as fast as possible you should entirely remove the following four lines from the function showMaze.



```
195 | try {  
196 |     Thread.sleep(milliseconds);  
197 | } catch (Exception e) {  
198 | }
```

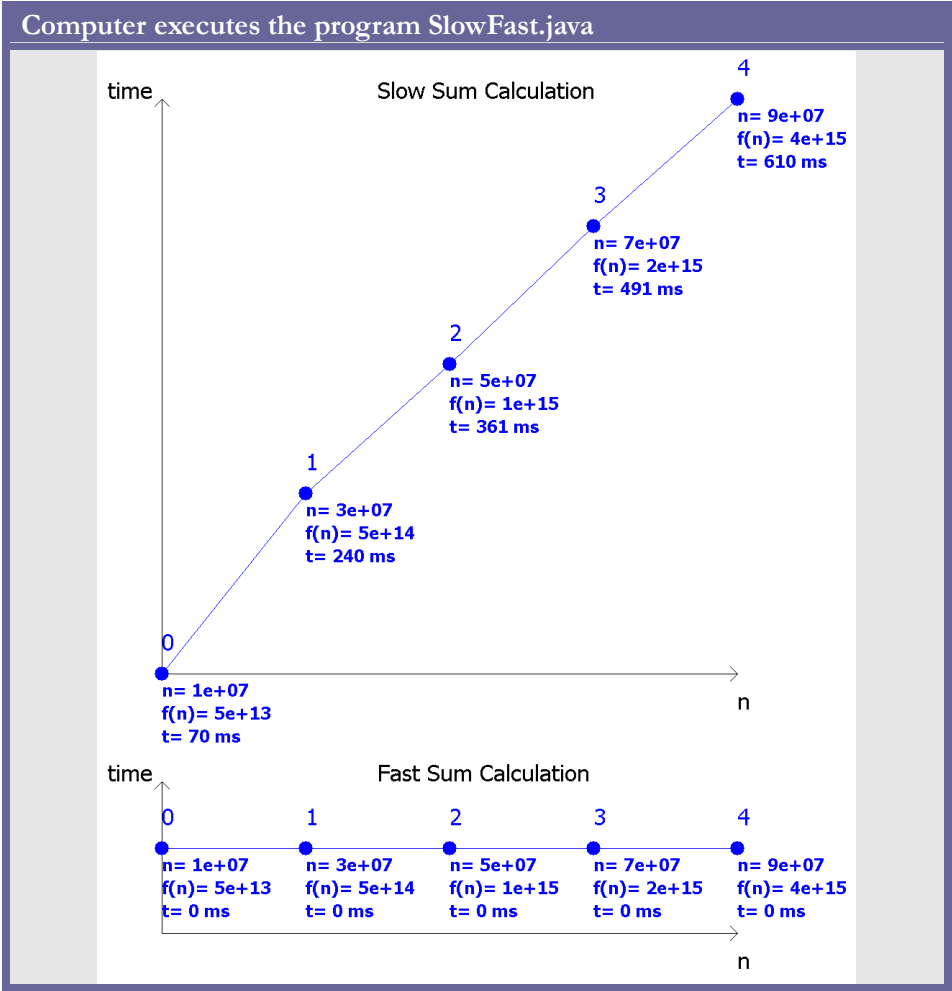
On certain machines sleeping zero milliseconds still slows down things a lot. The computer decides to do something else and returns to your program 'as soon as possible'. Drawing the maze on the screen takes some time so for very intense computations you would avoid displaying graphics on the screen until you have the final solution. That way the computer has only one thing to do and so it will be done quickly.

Run the program again. Notice the path is found extremely quickly now. One hundred thousand moves aren't very scary for most machines.



Activity 7 - Slow or Fast

Can you make a lot of money if you write programs?” asked Mike. He had to decide what to do at university next year. “If you write good programs that are useful and well done, companies will need you a lot” answered Alex.



“What is a good program?”

“It's a program that instructs the machine how to efficiently and quickly resolve an important problem. But that's not always easy.”

“Machines are so fast today. It ought to be easier today than when you were my age.”

“They are much faster, that's true. A very bad program on the fastest machine could still run slower than a very good program on a very old computer. I'll give you an example...”

Alex took a sip of coffee Liliane had put on the table.

“Sarah, what is the sum of the numbers between 1 and 5?”

“One plus two, three. Three plus three, six. Six plus four, ten. Ten plus five, fifteen. Fifteen, Alex.”

“Exactly. And the sum of the numbers between 1 and 10?”

“That's easy too. Fifteen plus six, twenty one. Twenty one plus seven, twenty eight. Twenty eight plus eight, thirty six. Thirty six plus nine, forty five. And plus ten, fifty five.”

“Very well. And the sum of the numbers between 1 and 100?”

Sarah frowned at him. She did not look amused at all. “Well, that would take a lot of time, why do you ask me to do that?”

Mike intervened to help his sister. “We could program it and have the machine do the calculations instead of Sarah.”

Alex approved. Sarah started to type in the program, helped by Mike. It was very short. They told the computer to do one hundred loops and for each loop to increment one counter and add the value of that counter to a variable they called 'result'. So the computer added one to the counter one hundred times, and every time it added the value of the counter to the result variable. The result variable remembered the previous result from the past loop. When the one hundred loops were completed, the computer could be told to display the value of the result variable. It contained the sum of the one hundred first numbers.

“5050. That was simple.” “We're getting pretty good at it!”

But Sarah soon noticed Alex had that little look on his face, when he is about to tease them. “Not so simple, maybe” She thought. She was right.

“Now give me the sum of the numbers between 1 and 1 billion.”

Mike replaced the value 100 with 1 billion in the program. He started the program and waited about five seconds.

“And between 1 and 10 billions.” This time, it took almost a minute.

Alex took a deep breath. “So now suppose this program is used to control a medical device that controls the heart of a patient. The program needs to give the answer in less than a second or the patient may die. What do you do?”

“We need to build a faster computer.”

“Maybe you can. Maybe a faster computer will cost too much money for the hospital to afford it. Maybe another developer in another company will find a way to get the answer fast enough with a cheap computer.”

“That developer can demand an awesome bonus if his company can avoid using super computers and save millions of dollars.”

“That's right. That's why good developers need to know mathematics very well, Mike. In particular, a good developer will remember the sum of the numbers between 1 and 100 can directly be calculated using this formula: 100 multiplied by the number 100 plus 1, then divide by two. Try it on the computer.”

Mike and Sarah tried the formula. It gave the same answer as their old program. It took less than a second when the old program was taking up to a minute.

Mike was thinking this through. “The new program is doing one multiplication, an addition and a division. It does not matter if we are asking for the sum of the numbers up to one hundred or one billion - it still only does one multiplication, one addition and one division. Our own program instead is doing two additions 100 times or 1 billion times. It's doing two hundred additions for a sum of up to 100, and it's doing 2 billion additions for a sum of up to 1 billion. So it gets slower and slower. No wonder your program is way better!”

Liliane's notes

Sarah typed in the program named SlowFast. I noticed three interesting parts in the program.

The code defines a function called `sumSlow`. It takes an argument that represents the number up to which we want to sum all the numbers to.

```
25 | public static long sumSlow(long n) {
```

The function seems to be doing a lot of calculation – it calculates the sum of numbers starting from one till the desired number `n`. I understand why it can be slow when the number `n` is very big. It does a lot of additions.

```
27 |     for (long i = 1; i <= n; i++) {  
28 |         sum = sum + i;  
29 |     }
```

The function returns the result. So for example, `sumSlow(2)` should return 3 (i.e. $1 + 2$), and `sumSlow(100)` should return 5050.

```
30 | return sum;
```

The second part is the function `sumFast`.


```
33 | public static long sumFast(long n) {
```

This function uses the mathematical formula $n * (n + 1) / 2$ to directly compute the sum of all the numbers up to n . It does not have to do many additions. So yes it looks like it should be fast even when n is very large.

```
34 |     return n * (n + 1) / 2;
```

Last, the program also defines a timer. A timer is very useful to verify how much faster or slower two functions resolving the same problem in two different ways are. A computer has a clock. It is easy to ask what time it is inside a program at the moment that part of the program is executed.

```
62 |     startTime = System.currentTimeMillis();
```

To find out how long `sumSlow(n)` takes, the program starts the timer before calling `sumSlow`. Then after `sumSlow` has finished the calculation, the program stops the timer which displays the time spent since it was started. It is exactly like using a stop watch to measure the performance of an athlete running a race.

```
16 |     timer1.start();  
17 |     long result = sumSlow(n);  
18 |     timer1.stop(n, result);
```

We use all this in the main part.

The program always starts from the main part. It calls `sumSlow` and `sumFast` to compute the sum of numbers up to a fairly large number. It displays the result returned by each function, and the time each took. When the number is large enough, it becomes clear that `sumFast` is greatly faster than `sumSlow`.

```
10 | public static void main(String[] args) {
```

I asked Alex if there was always a way to resolve any problem in a fast way. Alex explained it depended on the type of the problem. “For some problems, we know there is or isn’t a direct way to compute the solution. For some problems we have no proof yet and we search new techniques to resolve them faster. Mathematicians and computer scientists can spend their entire life improving the way one particular class of problems is resolved within a certain range of values. This can get very complicated. Frequently these advanced techniques must be written by expert developers. They need to document clearly how efficient is the technique for a certain range of arguments so that other developers know which techniques to use to fit their own particular needs.

A great technique for a particular problem with large numbers is often very much slower than a simpler technique for small numbers – so it depends on the arguments to compute the solution for”.

“And who found the formula to directly compute the sum of numbers up to a particular number?”

“We believe it comes from the mathematician Gauss, when he was only five years old at school. His teacher gave the kids what he assumed to be a lengthy task – calculate the sum of numbers up to one hundred. Every kid was busy doing painful additions of every number – one hundred of them, right! The teacher noticed Gauss appeared to do nothing. Upset, he came close to the young genius. Gauss had already written down the answer.”

“But why does the formula works? I can see it works, but it’s not clear as to why it does.”

“Here is one way to look at it. Adding all the number up to 100 is the same as adding 50, then 49 and 51, then 48 and 52, then 47 and 53 etc. down to 0 and 100. Adding 49 and 51 is the same as adding 50 twice. Adding 48 and 52 is also the same as adding 50 twice. So when you carefully examine this, it is like adding 50 one hundred and one times. Thus the formula $n / 2 * (n + 1)$.”

“Oh? That looks so simple now...”

See also the following related activities

- Path Finder
- Chess

To go further

- 1st century – Chinese book titled ‘The Nine Chapters on the Mathematical Art’ defines algorithms to calculate the volume of a sphere, solve linear equations or manage the emperor’s taxes.
- 9th century – Muhammad al-Khwarizmi publishes the Algebra and Arithmetic books in Baghdad. Three hundred years later the Western world adopts the decimal notation publicized by this mathematician. The modern word algorithm is derived from his name. It designates a technique to resolve a particular problem.
- 1960 – Tony Hoare invents Quicksort now the most widely used sorting algorithm. Quicksort is frequently used by programmers to sort names alphabetically, order a list of dates or numbers.
- 1965 – Gordon E. Moore predicts the number of transistors on cheap integrated circuits doubles every 24 months. This empirical rule has been true for over forty years. Computers are twice as fast as their ancestors two years older.

- 2000 – Clay Mathematics Institute sets a one million dollar prize to whoever proves that NP problems can be assimilated to P problems. No one has received the prize yet. Will you?
- 2006 – Cooper, Boone and GIMPS discover the largest prime number ever. The number is 2 multiplied by itself 32582657 times minus 1. It has almost ten million digits. Compare that to this entire book filled with only two hundred thousands characters.

Exercise 1: Slow? How slow is it?

The program SlowFast.java uses a timer to measure how fast or slow a portion of a computer program is. It's like a clock watch. Let's verify if this clock watch is actually accurate.

Remove lines 10 to 35 from the program SlowFast.java.



Replace them with the lines below.



```

10 | public static void main(String[] args) throws
    | Exception {
11 |     TimerGraph timer = new TimerGraph("Constant Timing
    | Test");
12 |     timer.setVisible(true);
13 |     for (int n = 1; n <= 5; n++) {
14 |         int calculationTime = 1234;
15 |         timer.start();
16 |         Thread.sleep(calculationTime);
17 |         timer.stop(n, n);
18 |     }
19 | }
```

On line 15 timer is started. On line 16 computer is asked to wait for a certain amount of time before executing the next line. This amount of time is here set to a little bit more than one second – one thousand two hundred and thirty four milliseconds to be exact. After that delay the machine executes line 17, it stops the timer. So we would expect the timer to have measured 1234 milliseconds.

We repeat this measurement 5 times to see if it changes.

Run the program. Verify the time displayed by the timer for each measurement is close to 1.2 second. Our timer does work...



An operation that takes an argument n often takes more time when the argument n gets bigger. Parents with two or three children have usually less free

time than those with a single child... Let's simulate programs that get slower differently when the argument n increases.

1. First let's say the calculation time is directly related to the argument n .

Replace the line 14 with the line below.



```
14 |      int calculationTime = 1234 * n;
```

Run the program. Time grows linearly with n . If n is multiplied by two, the calculation time is also multiplied by two.



2. Replace line 14 with the line below.



```
14 |      int calculationTime = 1234 * n * n;
```

Run the program. Time grows with the square of n . If n is multiplied by two then the calculation time is multiplied by four.



3. Again let's replace line 14 with the following line.



```
14 |      int calculationTime = (int) (1234 * Math.log(n))  
    |      + 1234;
```

Run the program. Time grows with the logarithm of n . The logarithm is a function that always grows but grows increasingly slower when n increases. Imagine someone who walks and gets tired. The person never stops but keeps walking slower and slower. Logarithm of 2 is close to 0.7, logarithm of 4 is about 1.4, just about twice as much.



4. Finally let's run the program with the following line 14.



```
14 |      int calculationTime = 1234 * (int) Math.exp(n);
```

Time grows exponentially with n . Exponential function grows increasingly rapidly when n is larger. Exponential of 2 is close to 7 and exponential of 4 is already as large as 54. You will have to wait a few minutes to see the fifth point when you run the program in this case.



The timer should display a graph that looks as follow in the four different cases we have just tried out.

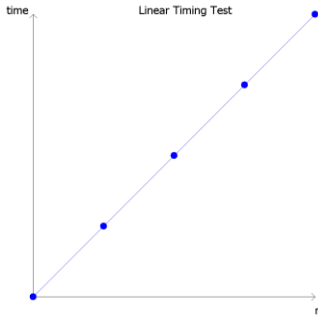


Figure 1 – linear function

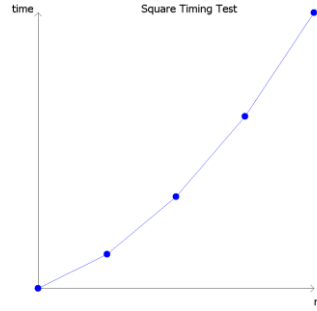


Figure 2 – square function

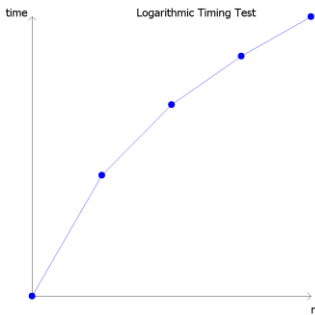


Figure 3 – logarithmic function

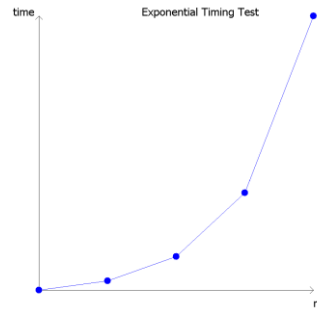


Figure 4 – exponential function

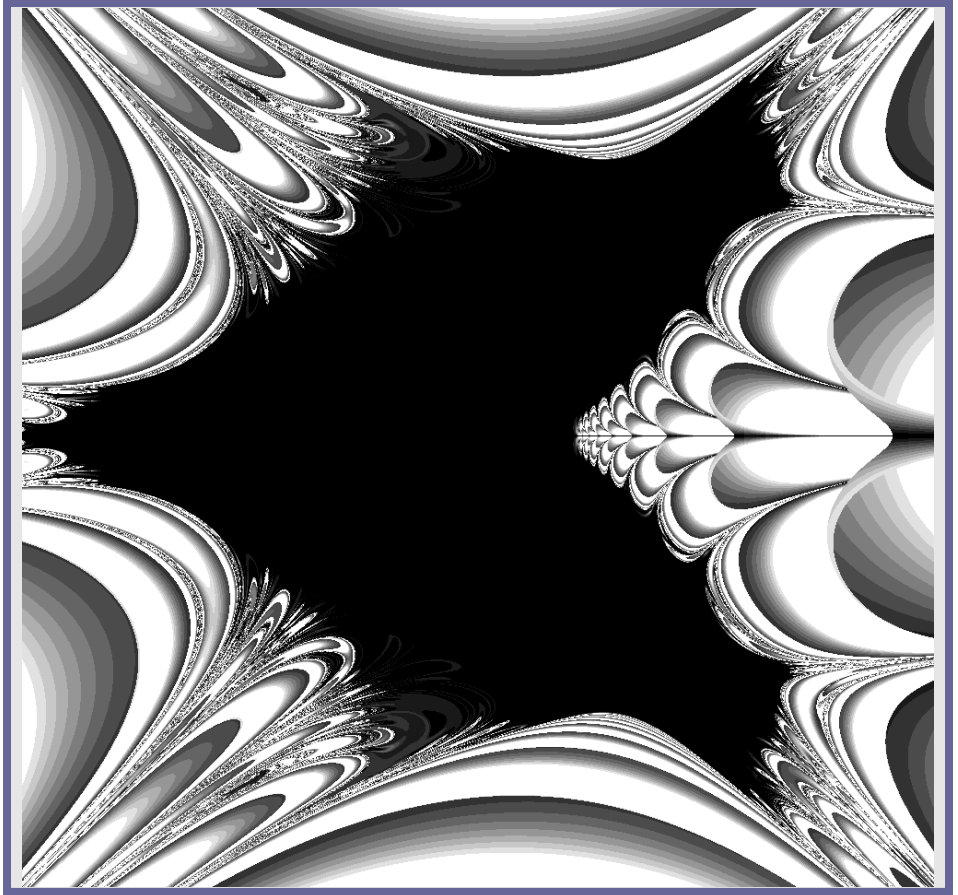
Computer programs rarely run in constant time when the data to process gets larger. Scientists are usually very happy when the program runs in linear time because it is fair to be twice as slow when you deal with data twice as large. Many problems are four times slower when the data is twice as complex. That is the square case, also called quadratic by mathematicians. A bad case is the exponential growth because the program rapidly takes too much time when the data increases. When you write your own programs ask yourself this question. How slow does it get when the number of spacecrafts drawn to the screen is doubled? Or when using realistic images with many more points? Constant and linear slowness are good news, logarithmic slowness isn't usually bad. Quadratic and exponential are very common and rapidly problematic no matter how fast your machine is... Let's try with a concrete case in the next exercise.

Exercise 2: Slow or fast video games?

Great video games calculate and draw realistic images as many times per second as possible onto the computer's screen. For example a flight simulator will draw an airplane flying over a city thirty times per second for a good animation. If the computer is slow it becomes necessary to draw an image with a smaller number of points – say about 500 points per line and 500 lines. A faster computer could run the same program and displays images with double that resolution – 1000 points per line and 1000 lines for example. However the computer needs to be four times faster... In the first case there are two hundred and fifty thousand points to calculate. In the second case we now have one million points to calculate – 1 megapixel resolution as digital camera vendors tell us. That's four times more points.

Let's calculate and draw a complicated image at different sizes. First with only ten points per line and ten lines. Then double that. And so on. To make things appealing to our eyes we will calculate a fractal with different shades of grey. The color of every point in the image must be calculated separately. The more points, the longer it takes to draw the whole image.

Computer executes the program `SlowFast2.java`



Remove lines 10 to 19 in the program SlowFast.java i.e. remove the main method.



Replace them with the lines below. It is a little bit long because we draw a fractal in increasingly larger areas of the screen – each area being double the width and the height of the previous image. We measure how long it takes to draw that image then hide the fractal image. Here is the actual code.



```
10 | public static void main(String[] args) {  
11 |     TimerGraph timer = new TimerGraph("Fractals");  
12 |     timer.setVisible(true);  
13 |     for (int width = 10; width < 2000; width = width *  
2) {  
14 |         FractalScreen screen = new FractalScreen(width);  
15 |         screen.setVisible(true);
```

```

16     timer.start();
17     screen.drawFractal();
18     timer.stop(width, width * width);
19     screen.setVisible(false);
20 }
21 }
22
23 static class FractalScreen extends Frame {
24
25     int WIDTH, HEIGHT;
26
27     public FractalScreen(int width) {
28         super("Fractal");
29         WIDTH = width;
30         HEIGHT = WIDTH;
31         setSize(WIDTH, HEIGHT);
32         addWindowListener(new WindowAdapter() {
33             public void windowClosing(WindowEvent e) {
34                 System.exit(0);
35             }
36         });
37     }
38
39     public void drawFractal() {
40         float cxMin = -1.5f, cxMax = -cxMin, cyMin =
41         cxMin, cyMax = cxMax;
42         float cx, cy;
43         int N = 10;
44         float[] z = new float[2];
45         Graphics gc = getGraphics();
46
47         for (int y = 0; y < HEIGHT; y++) {
48             cy = cyMin + y * (cyMax - cyMin) / HEIGHT;
49             for (int x = 0; x < WIDTH; x++) {
50                 cx = cxMin + x * (cxMax - cxMin) / WIDTH;
51                 z[0] = 0;
52                 z[1] = 0;
53                 for (int i = 0; i < N; i++) {
54                     z[0] = z[0] + cx;
55                     z[1] = z[1] + cy;
56                     multiply(z, z);
57                     if (Math.abs((int) z[0]) > N ||
58                         Math.abs((int) z[1]) > N
59                         || ((int) (z[0] * z[0] + z[1] * z[1]) > N *
60                             N)
61                     )
62                         break;
63                 }
64             }
65         }
66     }
67 }

```



```

60         int r0 = Math.max(0, Math.abs((int) z[0]));
61         int r1 = Math.max(0, Math.abs((int) z[1]));
62         if (r0 < N) {
63             gc.setColor(new Color(r0 * 25, r0 * 25, r0 *
25));
64             gc.drawLine(x, y, x, y);
65         }
66         if (r1 < N) {
67             gc.setColor(new Color(r1 * 25, r1 * 25, r1 *
25));
68             gc.drawLine(x, y, x, y);
69         }
70     }
71 }
72 gc.dispose();
73 }
74
75 public void multiply(float[] z1, float[] z2) {
76     z1[0] = z1[0] * z2[0] - z1[1] * z2[1];
77     z1[1] = z1[1] * z2[0] + z1[0] * z2[1];
78 }
79 }

```

Run the program. It is very noticeable how small images draw much quicker than very large images. Time is largely related to the total number of points in the image. It should be linear to the number of points in the image which is equal to the square of the width of the screen. Is that confirmed in your case?

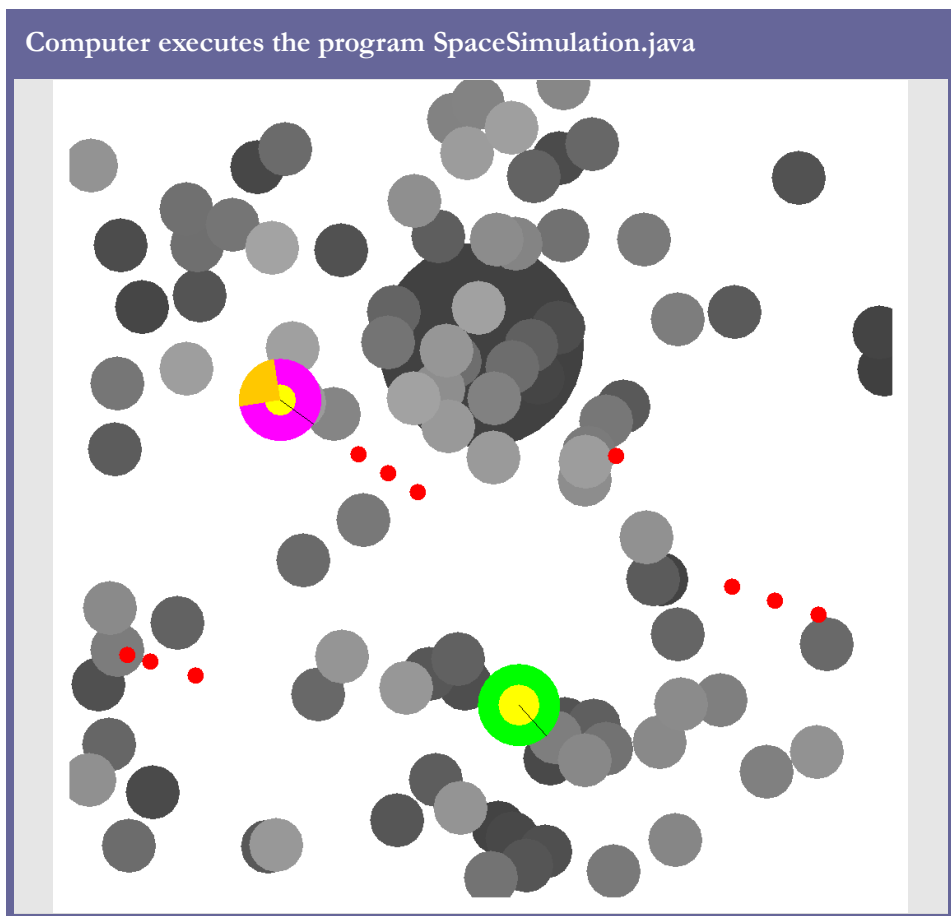


Programmers have to decide to calculate fewer points per image if calculating a realistic color for each point is slow. They often try to simplify the calculation while keeping enough realism to please the player and look better than the other games... On the other hand airplane pilots are trained on advanced simulators using the most powerful computers because realism must be as good as possible.

Activity 8 – Space Simulation

Sarah and Mike were observing the dark sky with a telescope Alex had built himself. Alex was showing them the different constellations and two planets that looked like two very luminous dots. Jupiter was bright yellow and followed by four little dots well aligned - its four biggest moons. Saturn looked orange with a very distinguishable ring. Sarah could hardly believe these distant objects all rotated around the Sun – that they were part of our solar system. How fascinating that the same force that described an apple falling from a tree also described the movements of these planets around the Sun. Alex was getting cold and decided to return to the chalet. They would get a hot chocolate and try out a space simulation. Yes, they would invent their own solar system and play with gravity...

Computer executes the program `SpaceSimulation.java`



Alex turned on the computer and ran the space simulation for Mike and Sarah who were now drinking a very warm chocolate milk in the chalet. Their eyes popped out. That was very different from the programs Alex had previously showed them. Alex could see he had received their full attention. He started the discussion.

- This program simulates the movement of bodies of different mass. It applies the laws of gravity - discovered by Newton - to calculate where each body is going to be, many times per second. This gives a realistic animation. You see why it is useful to study mathematics at college. Without math, we can't teach the computer how to simulate the movements of planets.

- what do the laws of gravity tell us?

- A body of a given mass is attracted to any other body in the universe that has a mass, and vice versa. So the body starts to accelerate toward other bodies, especially if these bodies are very heavy and very close to one another. For example, the moon keeps on rotating around the earth because it is attracted toward our planet which is very heavy compared to the moon.

- Why doesn't it fall on the earth then?

- It would if it were initially immobile with respect to the earth. Instead, it rotates in 28 days around our planet, and that counters pretty closely the effect of the earth's gravity. At the moment, the moon gets 4 centimeters further away from the earth every year so at some point it might just leave the earth all together, but that gives us some time.

- How does this rotation counter the gravity of the earth?

- When you rotate a ball attached to your hand with a rope, the faster you rotate, the stronger you need to hold on to the rope, right? Without the rope, the ball would simply go straight and not turn around you. So the gravity of the earth acts like a rope with the moon.

Alex pointed to the screen.

- With the simulation, we can decide how many planets there are, what size, position and initial velocity they have. There are also spacecrafts that can throw missiles. Spacecrafts and missiles have mass too so they are also subject to gravity. In this imaginary solar system, there is one extremely heavy star, moving very slowly. And one hundred planets and two spacecrafts. Having one hundred planets is certainly unusual but it is quite interesting to see how they move toward the heavy star, gain speed and then start moving all around. Of course, any setup can be defined, our solar system for example, or any imaginary one.

- How do you play the game?

- Each player controls a spacecraft. You try to hit each other. The spacecraft moves under the gravitational influence of the other bodies. It has one engine with which you can accelerate in any direction. Player can fire missiles. Now, nothing comes for free in the universe. The energy of the spacecraft decreases when powering the engine or when firing missiles. The energy level is

represented by a yellow circle. When it goes down to zero, the spacecraft vanishes from the universe. Simple, isn't it?

- Oh, it's like Pac Man, when a planet gets to one side of the screen it reappears on the other side.

- Mathematicians call that a two dimensional torus. It isn't very realistic to represent the actual universe, I admit. But it is particularly convenient for our simulation game on a small screen.

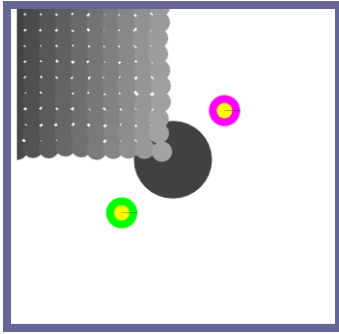


Figure 1 – initial position

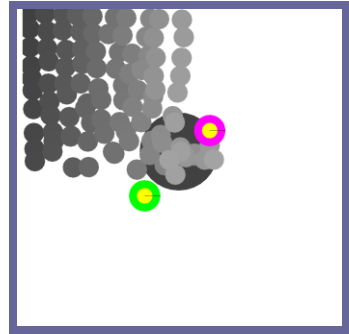


Figure 2 – after two seconds

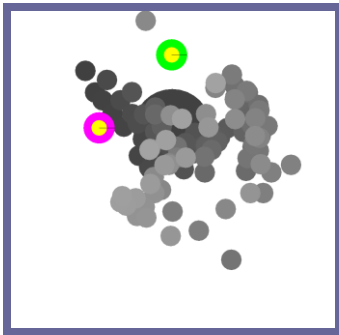


Figure 3 – after ten seconds

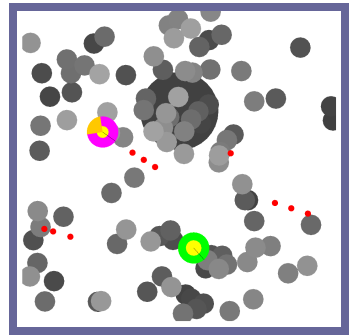


Figure 4 – after one minute

Figures above show the space simulation at different time intervals. In the last figure, the pink spacecraft is firing missiles which look like red circles. It is using thrust to control its trajectory – notice the orange portion at its rear end. Using thrust decreases the energy level represented in yellow inside the spacecraft.

And so Mike and Sarah started to play.

Mike controlled the green spacecraft using keys on the left side of the keyboard. Sarah controlled the pink spacecraft with keys on the right side of the keyboard.

Table 1 - Spacecraft control keys

Command	Key for green spacecraft	Key for pink spacecraft
Accelerate	d	l
Fire missile	f	;
Rotate toward left	a	j
Rotate toward right	s	k

Mike maneuvered very aggressively. He quickly ran out of energy. Sarah took her time to position her spacecraft near Mike’s one. Mike could only watch Sarah closing in on his spacecraft while he orbited helplessly in space around the big star. With no energy left, he just could not power up his engine. He also had no shield left to guard against missile hits. So Sarah won the first game.

Liliane’s notes

The program starts in the main part, space simulation is initialized. The program determines how many planets have to be shown, at which location and with what initial speed. A different initialization would give an entirely different simulation.

```
15 | public static void main(String[] args) throws
    | Exception {
16 |     SpaceSimulation simulation = new
    | SpaceSimulation();
17 |     simulation.create();
```

Here we ask the computer to give us a portion of the screen to draw the planets.

```
18 |     Screen screen = new Screen(simulation);
19 |     screen.setVisible(true);
```

We will calculate a new position of the planets every fifty milliseconds. This means up to twenty images per second will be shown on the screen. It’s a pretty smooth animation. We can increase the delay if the computer is not fast enough to calculate so many updates per second. Keep in mind that a movie requires about 30 images per second. That’s one update every 33 ms.

```
20 |     long displayDelayMilliseconds = 50;
```

Our simulation program can increase or decrease the simulation speed. If the planets move fast we will use a small delay. On the other hand, for very slow moving planets – like the Earth around the Sun, that’s a full year – we would use a large interval of time so that we see things accelerated on the screen.

```
21 |      long simulationDelayMilliseconds =  
    |      displayDelayMilliseconds / 10;
```

The computer will keep on repeating the following lines.

```
22 |      while (true) {
```

Sleep for 50 milliseconds before we calculate and show the next position of the planets.

```
23 |          Thread.sleep(displayDelayMilliseconds);
```

Calculate the new position and speed of the planets after the given amount of milliseconds set in `simulationDelayMilliseconds` has passed by.

```
24 |          simulation.update(simulationDelayMilliseconds);
```

We ask the computer to redraw our portion of the screen with the new positions of the planets.

```
25 |          screen.repaint();
```

After that we go back to sleep for fifty milliseconds, update the positions, repaint etc. That’s a lot of work but the result looks very nice on the screen and that’s why computers were created – to do a lot of calculations quickly.

See also the following related activities

- Chess

To go further

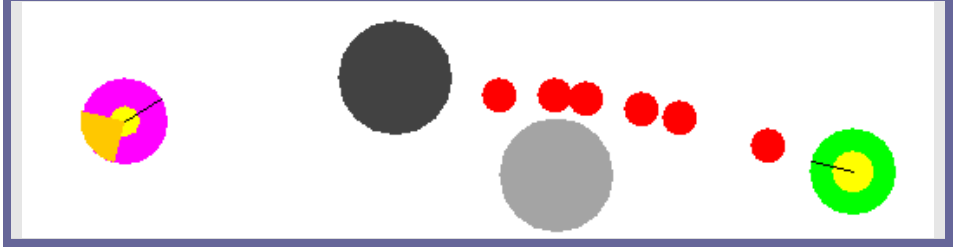
- 384BC – Aristotle claims heavy objects fall faster than light objects. Aristotle believes the earth is at the center of the universe.
- 1543 – Nicolaus Copernicus describes the universe using eight spheres, placing the sun at the center and with the earth rotating around it in his treatise “On the Revolutions of the Heavenly Spheres”. This

- representation is far more concise than Aristotle's representation (in which the earth is fixed at the center): the trajectory of planets from the perspective of the earth is very complicated; it is simpler to describe them from the perspective of the Sun.
- 1589 – Galileo Galilei drops iron balls of different mass from the top of the Leaning Tower of Pisa in Italy. They hit the ground at the same time, proving that Aristotle and his followers had been wrong for two thousand years.
 - 1687 – Isaac Newton shows that an apple falling on the earth and planets moving around the sun are governed by the same laws. He describes these laws in his treatise *Philosophiae Naturalis Principia Mathematica*.
 - 1915 – Albert Einstein presents a theory of gravity named 'general relativity'. That theory is more precise than the one described by Isaac Newton. It accurately predicts deviation of light by heavy objects like the Sun and gravitational time dilatation (a clock under very strong acceleration because it rotates very rapidly around the earth, such as a GPS satellite, indicates a time that's behind the time shown by an identical clock that has remained at a fixed point on the earth). For many situations, Newton's theory is accurate enough and yields much simpler calculations, so it is still very widely used.
 - 1962 – Steve Russell programs the first space simulator game named 'Space War' at the Massachusetts Institute of Technology on a PDP-1 computer.
 - 1965 – Arno Penzias and Robert Woodrow Wilson build a horn-shaped antenna about ten meters long. They accidentally discover a radiation signal that comes through any direction in the universe and is thought to have been generated by the Big Bang itself.
 - 2006 – Voyager 1 spacecraft is 15 billion kilometers away from the sun. It was launched in 1977. It explored Jupiter, Saturn, Uranus and Neptune through gravity assist technique – use the gravity field of these planets to travel through the solar system. Voyager 1 is now moving so fast it will never return to our solar system.

Exercise 1: Play battleship around two super massive black holes

A black hole is a star so dense that even light cannot escape from it. Astrophysicists have discovered binary black holes: two black holes rotating around each other under their gravitational field's influence. Binary black holes are indirectly observable because they disturb everything around them – galaxies, stellar gas, and light. It is believed they ultimately merge into one another.

Computer executes the program SpaceSimulation1.java



Let's replace the create method in the program with one that creates the elements for our simulation. Remove lines 55 to 122.



We need to provide enough room for a fun game play. Next we add the first black hole. It is very heavy and initially located in the lower right corner of the simulation space. It is animated with a horizontal movement toward the right.



```
55 public void create() {
56     width = 20;
57     height = width;
58
59     Element planet1 = new Element();
60     planet1.type = "planet";
61     planet1.id = 100;
62     planet1.x = 2 * width / 3;
63     planet1.y = 2 * width / 3;
64     planet1.radius = width / 30;
65     planet1.mass = 10E12;
66     planet1.vx = 10;
67     planet1.vy = 0;
68     planet1.thrust = 0;
69     planet1.thrustAngle = 0;
70     planet1.maxEnergy = Double.MAX_VALUE;
71     planet1.energy = planet1.maxEnergy;
72     elements.add(planet1);
```

The second black hole is the companion of the first one. It is placed in the upper left corner. It moves horizontally toward the left.



```
73
74     Element planet2 = new Element();
75     planet2.type = "planet";
```



```

76 | planet2.id = 1000000;
77 | planet2.x = width / 3;
78 | planet2.y = width / 3;
79 | planet2.radius = width / 30;
80 | planet2.mass = 10E12;
81 | planet2.vx = -10;
82 | planet2.vy = 0;
83 | planet2.thrust = 0;
84 | planet2.thrustAngle = 0;
85 | planet2.maxEnergy = Double.MAX_VALUE;
86 | planet2.energy = planet2.maxEnergy;
87 | elements.add(planet2);

```

We now place the first spacecraft. Notice its mass is incredibly much smaller than that of either of the black holes. Clearly the spacecraft will be under the influence of the black holes powerful gravitational attraction. The spacecraft is initially immobile.



```

88 |
89 | Element spacecraft = new Element();
90 | spacecraft.type = "spacecraft";
91 | spacecraft.id = 1;
92 | spacecraft.x = width * 1 / 3;
93 | spacecraft.y = width * 2 / 3;
94 | spacecraft.radius = width / 40;
95 | spacecraft.mass = 1;
96 | spacecraft.vx = 0;
97 | spacecraft.vy = 0;
98 | spacecraft.thrust = 0;
99 | spacecraft.thrustAngle = 0;
100 | spacecraft.maxEnergy = 100;
101 | spacecraft.energy = spacecraft.maxEnergy;
102 | elements.add(spacecraft);

```

A second spacecraft is added. The game is all set for an intense game play in one of the most exotic parts of the universe.



```

103 |
104 | spacecraft = new Element();
105 | spacecraft.id = 2;
106 | spacecraft.type = "spacecraft";
107 | spacecraft.x = width * 2 / 3;
108 | spacecraft.y = width * 1 / 3;
109 | spacecraft.radius = width / 40;
110 | spacecraft.mass = 1;

```

```

111 | spacecraft.vx = 0;
112 | spacecraft.vy = 0;
113 | spacecraft.thrust = 0;
114 | spacecraft.thrustAngle = 0;
115 | spacecraft.maxEnergy = 100;
116 | spacecraft.energy = spacecraft.maxEnergy;
117 | elements.add(spacecraft);
118 | }

```

Run the program. Play with another player.



Exercise 2: Simulate the solar system HD 160691

Astrophysicists are searching for solar systems similar to ours. We might one day find a planet similar enough to Earth. Such planet could even possibly host extra terrestrial life.

Star HD 160691 has a size and weight similar to our Sun. It is only forty nine light years away from us – meaning its light takes forty nine years to travel to reach our telescopes located on Earth. Between 2000 and 2006, astronomers discovered four planets rotating around HD 160691.

Computer executes the program SpaceSimulation2.java



Modify the program SpaceSimulation so that it simulates the HD 160691 solar system. The function create is rewritten so that the simulation represents an area of space about the size of the HD 160691 solar system with the star and three of its biggest planets. We will ignore the fourth one that is very small and very close to the star itself.

The simulation will show an area of space about two times the distance between the star and its most distant planet (ten times the distance from our Earth to the Sun).



```

55 | public void create() {
56 |     width = 10 * 149.598E9;
57 |     height = width;

```

Then we add Star HD 160691. Its mass is almost identical to that of our own Sun (1.08 times to be exact). Its radius is 1.245 times that of the Sun. We will show it ten times bigger in the simulation so that we actually see something on the computer screen. We initially place the Star in the middle of the simulation area, immobile (speed vx and vy set to zero).



```
58 |
59 |     Element hd160691 = new Element();
60 |     hd160691.type = "planet";
61 |     hd160691.id = 1;
62 |     hd160691.x = width / 2;
63 |     hd160691.y = width / 2;
64 |     hd160691.radius = 1.245 * 695E6 * 40;
65 |     hd160691.mass = 1.08 * 1.988435E30;
66 |     hd160691.vx = 0;
67 |     hd160691.vy = 0;
68 |     hd160691.maxEnergy = Double.MAX_VALUE;
69 |     hd160691.energy = hd160691.maxEnergy;
70 |     elements.add(hd160691);
```

We add the first planet. It is named HD 160691b and was discovered in 2000. It is one and a half astronomical units away from the star. One astronomical unit represents the distance between our planet Earth and the Sun. So that means this planet is not that much further away from its Star than our planet is from the Sun. Its mass is about one and a half times one of the planet Jupiter. That's a very heavy planet.



```
71 |
72 |     Element hd160691b = new Element();
73 |     hd160691b.type = "planet";
74 |     hd160691b.id = 100;
75 |     hd160691b.x = width / 2 + 1.5 * 149.598E9;
76 |     hd160691b.y = width / 2;
77 |     hd160691b.radius = hd160691.radius / 2;
78 |     hd160691b.mass = 1.67 * 1.899E27;
79 |     hd160691b.vx = 0;
80 |     hd160691b.vy = -1e10;
81 |     hd160691b.maxEnergy = Double.MAX_VALUE;
82 |     hd160691b.energy = hd160691b.maxEnergy;
83 |     elements.add(hd160691b);
```

The second planet is HD 160691C. Discovered in 2004, it is three times heavier than Jupiter. It is about four astronomical units away from its Star. Quite far.



```
84 |
85 |     Element hd160691c = new Element();
86 |     hd160691c.type = "planet";
87 |     hd160691c.id = 10000;
88 |     hd160691c.x = width / 2 + 4.17 * 149.598E9;
89 |     hd160691c.y = width / 2;
90 |     hd160691c.radius = hd160691.radius / 3;
91 |     hd160691c.mass = 3.1 * 1.899E27;
92 |     hd160691c.vx = 0;
93 |     hd160691c.vy = -1e10;
94 |     hd160691c.maxEnergy = Double.MAX_VALUE;
95 |     hd160691c.energy = hd160691c.maxEnergy;
96 |     elements.add(hd160691c);
```

Planet HD 160691e was discovered in 2006. It is half the weight of Jupiter and at about the same distance to its Star than the Earth to the Sun.



```
97 |
98 |     Element hd160691e = new Element();
99 |     hd160691e.type = "planet";
100 |     hd160691e.id = 100000000;
101 |     hd160691e.x = width / 2 + 0.921 * 149.598E9;
102 |     hd160691e.y = width / 2;
103 |     hd160691e.radius = hd160691.radius / 5;
104 |     hd160691e.mass = 0.5219 * 1.899E27;
105 |     hd160691e.vx = 0;
106 |     hd160691e.vy = -1e10;
107 |     hd160691e.maxEnergy = Double.MAX_VALUE;
108 |     hd160691e.energy = hd160691e.maxEnergy;
109 |     elements.add(hd160691e);
110 | }
```

Run the program and observe the planets in rotation around their star.



The planets move really slowly on the screen. To observe their orbital movements we can increase the amount of simulated time spent between each step of the simulation. Remove line 21.



```
21 | long simulationDelayMilliseconds =  
    displayDelayMilliseconds / 10;
```

And replace it with this line. The new value is 100 times bigger than earlier.



```
21 | long simulationDelayMilliseconds =  
    displayDelayMilliseconds * 10;
```

Run the program again. Now we can really see the planets rotating around their star. Will a human being ever visit them?



Activity 9 - Web Browser

Mike and Sarah were browsing the web, checking the latest sports news, and visiting their friends' blogs. Sarah turned to Alex and asked if he knew how to use the web browser. Alex smiled and replied that yes, he knew how to use a web browser a little. Then he asked her if she knew how to build a web browser. "One that can go onto the real web?" the young girl enquired. "Certainly," answered Alex. "Give me some room, and let's build our own web browser".

Computer executes the program WebBrowser.java

```
Please enter url:
http://www.funsciencewithyourcomputer.org/cern/cern.html
.....
You are at the CERN entrance.
Mr. Higgs has lost his boson. Can you find it?
[0] Go to P2 site [1] Go to P8 site

Please enter url:
0
```

Rain had been pouring down all day, and most of the nearby mountains were hidden in grey clouds. Yet the atmosphere was joyful in the 'salle des fetes' of the village. This was the main indoor room, used for many activities such as wedding parties, movies and official ceremonies. The 'salle des fetes' also hosted a very modern computer room, connected through a fast internet connection to the nearby bigger city Lyon. Alex lived in a small mountain village. Thanks to funding from the European Union the local inhabitants of this remote village could be trained in useful computer techniques by students from the University of Lyon via video conferencing. Via the web a few of them were able to sell their delicious dairy products to customers located anywhere in the world. Alex had taken part in this project and he was proud to show the two modern teenagers what locals from a small village could do. Today the weather was preventing everyone from enjoying rides in the mountain, so it was a perfect opportunity to visit the 'salle des fetes'.

"So Alex, how do we write a web browser? This sounds so complicated, I really don't know where to start."

“You're right Sarah. A web browser nowadays is a very sophisticated application, made of many different pieces. It ought to be, when you see all the wonderful things it allows you to see and do. So we will simplify a little bit. But we will still build a web browser that can actually visit real websites. Tell me, how do you browse the web?”

“Well, I enter the address of the website I want to visit in here. Then I click here, and the page shows up.”

“And next?”

“I read, I see a part I like and I click on it to get the full story.”

“How do you know you can click on it to access another document, Sarah?”

“The words look different from the rest of the text, so I know I can click on it with the mouse cursor to visit some other place.”

“You click on a hyperlink. You see the description of the hyperlink, it shows in a different color than the rest of the text. When you click over it, you are asking the web browser to visit the document the hyperlink refers to. What happens?”

“The web browser gets the content of that new document, and then displays it. This new document often has words I can click on, I mean hyperlinks. And so on. So it's really easy and fun to use.”

“It is. Did you know the person who created the first web browser was not too far from here? His name was Tim Berners-Lee, and he wrote a program called WorldWideWeb in 1990 at CERN - the European Particle Physics Laboratory. Of course at that time there were no websites. Tim wanted to make it easy to visit documents related to other documents' contents, regardless of where these documents were actually stored - in one of the many computers of the CERN, in North America or elsewhere. Now you know why so many addresses start with the three letters www. World Wide Web. So every document can have a unique address, and the document is written in a format that contains its information and also the addresses of other documents the reader may want to visit when reading certain parts of that document. Do you know the name of the format used for most documents in the web, Sarah?”

“I remember Mike talked about it once. Something like HTML? But I don't know what it means?”

“Exactly. So the documents on the web are written in a certain way, the HTML way - Hyper Text Markup Language. It's very popular and most people know what it looks like. When you look at the web browser, it shows you the document in a way that you can read it - showing nice images, font colors, and text you can read or click on to visit other documents. That's the part most people see. What the web browser program sees is something else - it's the HTML instructions that say how the document should be shown to readers, and what documents should be visited when the reader clicks on a particular hyperlink.”

Sarah was making efforts, but Alex could see she was getting lost. “Enough theory, Sarah. Let's try something. You are looking at a particular document, now. What is it?”

“The weather forecast, Alex. It's going to be sunny tomorrow afternoon.”

“Marvelous. Now right click on the web browser and choose 'Show Source'.”

A window opened, showing a weird and apparently complicated document. Some parts seemed readable, but it contained also many unusual words like `<p>` and `<a>`.

“What are those words, Alex?”

“That's what we call the markups, Sarah. This tells the web browser what to do with the content, what it represents. Here is a simple example.”

```
1 | <html>
2 | <body>
3 |   <p>It is raining today</p>
4 |   <p>It will be sunny tomorrow</p>
5 | </body>
6 | </html>
```

“The web browser knows what to do when it sees this kind of document. The first word indicates the document uses the HTML format. That is, it follows the rules of HTML to organize the document. HTML defines little words called markup to describe different parts of a document. You see, toward the end there is a word that is `</html>`, right? Well, everything in between `<html>` and `</html>` is the content of the HTML document. And then the main part of the html document is within the two words `<body>` and `</body>`. In our case, it contains two paragraphs.”

“That's what `<p>` means? Paragraph?”

“Yes. Between `<p>` and `</p>` you actually have the text for that paragraph. Then we have a second paragraph.”

“But why does it matter? When I write on a notebook, I don't type in `<p>` words?”

“By typing the `<p>` words, you are saying to the web browser 'start a new paragraph here'. So the web browser can show paragraphs separately from each other. But wait, the more interesting part is to come. Let's add one line.”

```
1 | <html>
2 | <body>
3 |   <p>It is raining today</p>
4 |   <a href="http://www.weather.ca">Click here to see
   | the weather in Canada</a>
6 |   <p>It will be sunny tomorrow</p>
7 | </body>
8 | </html>
```


“What do you think this means, Sarah?”

“I guess it's showing the sentence 'Click here to see the weather in Canada', but I do not really understand the first part. Oh let's see, I know this website, we use it all the time at home. It's the website to see the weather in our country, in Canada.”

“So do you see where this is going?”

“Is that a hyperlink? The user can click on the text 'Click here to see the weather in Canada'. Then the web browser visits the website <http://www.weather.ca>?”

“Exactly. The `<a>` word in HTML defines hyperlinks. It's the key to hyperlink navigation on the web.”

“Is that all there is to HTML?”

“I'm afraid not, Sarah. There are dozens, hundreds of different special words in the most recent versions of HTML format. People and companies agree to add new words to do something that could not be done previously. And after a few years, things get really complicated, especially when the people and companies don't necessarily agree with each other and want their web browsers to do things other web browsers cannot do. In our case, we will simplify things a lot. We will write a web browser that knows how to show paragraphs and hyperlinks.”

“So we just care about the words above?”

“That's right. We will just care about the words `<p>` and `<a>`. Now let's discuss the different parts our program needs to do.”

“First, I need to be able to say which document I want to see. So I need to be able to enter an address.”

“Then we need to download that document from the machine that stores it into our own computer - using the address.”

“We need to search through that document for all the paragraph words `<p>` and hyperlinks `<a>`.”

“We can then display the contents of all the paragraphs we have found and the descriptions of the hyperlinks.”

“The user can read it, and we can ask the user to enter a new address or to select one of the hyperlinks we have discovered so far.”

“You got it, Sarah. I'll show you how all that is done in the WebBrowser program. Downloading a document from the web is made very easy nowadays.”

Liliane's notes

The program starts in the main part.

```
10 | public static void main(String[] args) {  
11 |     WebBrowser browser = new WebBrowser();
```

And it will keep repeating the lines that follow.

```
11 | while (true) {
```

Ask the user which website he or she wants to visit.

```
12 |     String url = browser.askUrl();
```

The program connects to the Internet and requests the content of the document at the address given by the user.

```
13 |     String html = browser.download(url);
```

The program searches for html paragraphs and hyperlinks inside the content received from the Internet. That's what we want to show to the user.

```
14 |     String document = browser.parse(html);
```

The program asks the computer to show the document on the screen so the user can read the paragraphs and decide if he or she wants to visit one of the hyperlinks.

```
15 |     browser.display(document);
```

At this point the program comes back and waits again for the user to enter the address he or she intends to visit.

See also the following related activities

- Vector Graphics

To go further

- 1989 – Tim Berners-Lee invents the World Wide Web while working at CERN. Mr. Berners-Lee ingeniously combines the networking capabilities of personal computers with hypertext (text with links to further information), making it possible to browse documents referencing other documents without having to care about their actual location.
- 1993 – Lou Montulli creates a text-based web browser named Lynx. It is very widely used for the many computers that don't support displaying graphics.

- 1994 – Netscape Navigator 1.0 becomes the most popular and most advanced web browser. It supports web pages that include both text and graphics. It is the first to support on the fly display. It can show portions of the web page before the whole content has been loaded through a slow internet connection.
- 1997 – Internet Explorer 4.0 from Microsoft is about to end the war of the browsers with Netscape. It is the first web browser supporting dynamic web pages. A dynamic page can animate itself by running a script and changing its own HTML markup content inside the web browser. By 2002, 96% of users browse the web with Internet Explorer.
- 2004 – Google allows users to quickly search over 4 billions web pages and almost a billion images and newsgroup messages. It announces revenues over 800 million dollars. Three years later this web company is valued at around 140 billion dollars.
- 2006 – Google purchases the web site YouTube for over 1.6 billion dollars. YouTube allows web users to share their video for free.
- 2007 – Web browsers have become among the most sophisticated and modular computer programs. The current winners are Internet Explorer, Firefox and Safari. They all require teams with dozens or hundreds of excellent developers. These programs are modular because users can install separate plugins to support even more advanced web pages. Popular plugins include Flash for web games and animations, Adobe Reader for visualizing PDF documents. HTML is now in its fourth version and talented graphic artists and web designers create interactive and beautiful websites every day. People blog, share videos with their friends, bank and book travels through the web...

Exercise 1: Search the Higgs Boson at the CERN

Tim Berners-Lee was working at CERN. CERN accelerates particles and analyses how they collide with each other. The physicist Higgs has speculated the existence of a particle named the Higgs Boson. A new and giant detector is being built at CERN to help detect it. The detector weighs seven thousand tons.

Run the WebBrowser program. Go to the site <http://www.funsciencewithyourcomputer.org/cern/cern.html> . The site contains information about different parts of CERN where we may find the Higgs Boson one day. It is a good way to test our WebBrowser program.



Exercise 2: Support additional HTML tags

Our WebBrowser program understands paragraph and hyperlink tags. Let's modify the program so it also supports the <hr/> tag. The <hr/> tag is used to introduce a horizontal line.

Find the parse function defined between lines 55 to 84.



Insert the following lines to the parse function after line 80.



```
81 |         if (isHrTag(tag)) {  
82 |             text += "\r\n_____ \r\n";  
83 |         }
```

We insert the isHrTag function after the isHyperlinkTag function.



```
109 | boolean isHrTag(String tag) {  
110 |     return tag.startsWith("<hr");  
111 | }
```

And that's it. It is hard to imagine a simpler tag, no?

Run the program. Visit the site <http://www.funsciencewithyourcomputer.org/cern/cern.html> presented in the previous exercise. You can now notice the horizontal dividers. They were previously ignored.

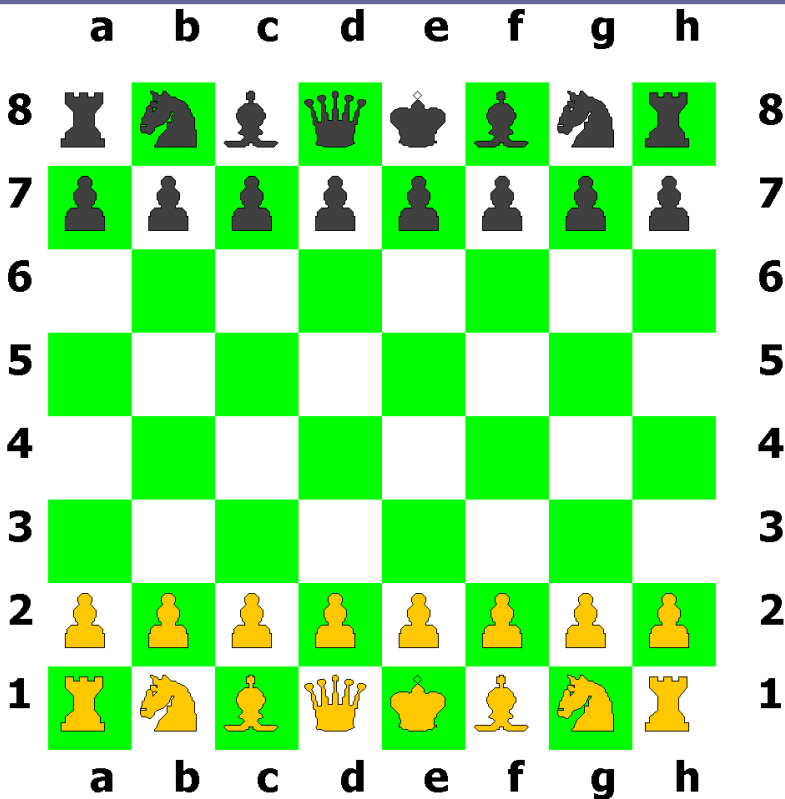


You can implement other tags as an exercise, but you will probably keep using a real web browser application because it takes so much more work to correctly parse HTML in order to support modern websites...

Activity 10 – Vector Graphics

Sarah was drawing flowers. Alex was observing how she used a pencil to draw the contour, and then switched to various colored pencils to fill her artwork with vivid shades. Alex waited till she was done and praised her skills. "I really can't draw, Sarah. The best I can do is use geometry to teach a computer how to draw and color various shapes. It certainly does not have that artistic and human touch I see in your drawing but if the two of us pair our talents, we might achieve something very worthy, what do you think?"

Computer executes the program VectorGraphics.java



- Sarah, see this chess board on the table. It was hand-made by my grandfather a few years after he had built this chalet. I was thinking of teaching you

Chess one of these days, and Mike could learn how to program a chess game. For today, I suggest we instruct the computer how to draw a chess board with its pieces. Then Mike will another day use this program to display the moves played between a human player and his chess program.

- Alex, what is it that you want me to do? Paint this board?
 - Here are a few sheets of paper. As you can see, this paper has a grid. What I'd ask you to try is to draw the shapes of the chess pieces - each piece within an area of let's say thirty six little squares.
- Sarah did look a little bit surprise. Really, there was always something new to do with Alex... She started to draw the rook.

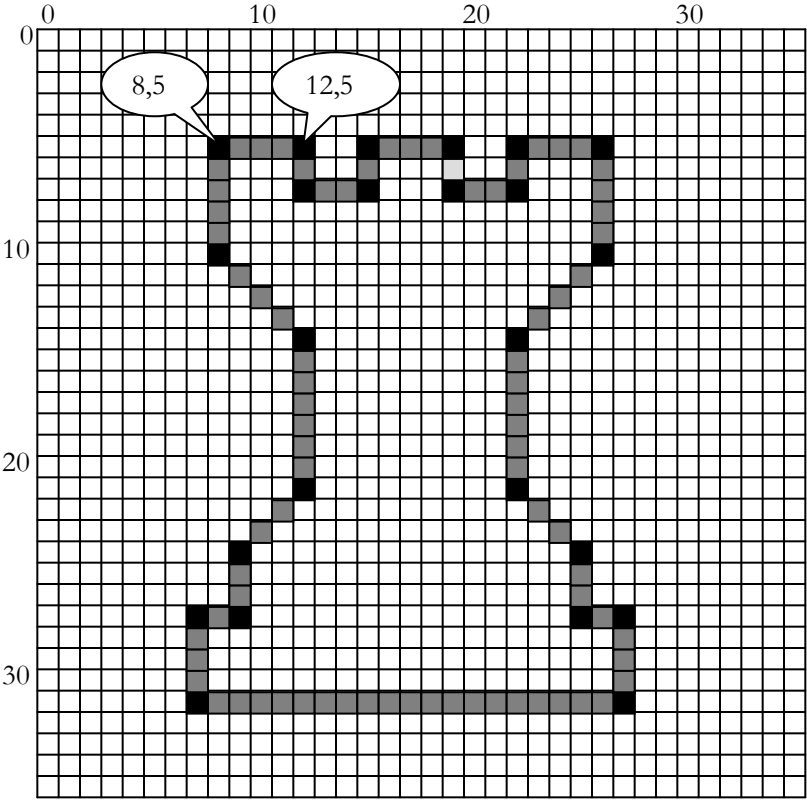


Figure 1 - Rook Vector Drawing

- Very well Sarah, that's exactly what I need.
- Mike was intrigued and Alex explained to him what they were doing. Mike offered his help and soon brother and sister were reproducing each piece on the grid of the paper. Alex noticed that Sarah had done a very nice job - pawn,

horse and rook were quite nice. Mike had completed the bishop, queen and the king. It did look a little bit less polished but Alex did not say anything.

- Thanks to the two of you. You were very patient and now we can move on to the next step. We are going to vectorize these figures and teach the computer how to display them on the screen.

- Vectorize?

- We are going to store the list of all the segments that constitutes a piece. For example, let's start with the rook. The rook is inside an area that contains thirty six little squares horizontally by thirty six little squares vertically. Let's begin from the top left corner of the rook. That point here is on the ninth column and on the sixth row. We will start our coordinates from zero, so that point is at coordinate 8 (horizontally) and 5 (vertically). The first segment goes from that point to that point at 12 and 5.

- So we go all around the shape and mark the coordinates of the points?

- Yes, coordinates of the points that are linked through straight lines - horizontal lines, vertical lines or in diagonal.

The three of them divided the work and started vectorizing the figures. I.e. they were collecting the points that were forming the figures. After about fifteen minutes, they were done. "Things go fast when you work in a team" thought to herself Sarah.

Alex collected the numbers for every figure. Mike had forgotten to mark the name of the figure under his list of numbers, so it took a few more minutes to try to read these numbers and decide the figure they represented.

- Now what we are going to do is enter these numbers into a file in the computer. We are going to use a format similar to what many applications use today - it's called scalar vector graphics format - or svg. It's a little bit like the HTML language we talked about in the web browser activity - it uses tags to define a graphic (tag g). The graphic represents here one of our chess pieces. It has a name, given inside the symbol tag - for example, king or knight. The graphic itself will be described with our list of points, i.e. a list of segments which as you know is called a polygon. We will name this file chess.svg.

```
<svg>
  <g>
    <symbol>rook</symbol>
    <polygon>8,5,12,5</polygon>
  </g>
</svg>
```

chess.svg (partial version that defines a couple of points for the rook)

- And now, all we have to do is write the program that will read this file to learn how to draw each chess piece. The program will also be taught how to draw a board game of eight squares by eight squares. It will draw the pieces representing a position of the chess game.

- Will it also play chess, Alex?
 - No Sarah. Today we will just focus on the drawing part. That will be for some other time.
 - How will we know if the piece is for white or black? We are only giving the contour of the piece, not its color?
 - Very well observed Mike. We are going to teach the computer how to fill up a polygon with a particular color. There are many ways to do it, we will choose one method that is relatively easy to program. First, we will search for the center of the piece to fill. Then we will fill all the little squares around that are neighbors of the center with the color. Then we will repeat on the neighbors of the neighbors. And so on, till we have reached all the little squares within our polygon.
 - How do you calculate the center of the piece, Alex?
 - You go through each point in our polygon. You remember how far to the left and to the right you have been, how far up and down. The middle point is then given by the sum of left and right coordinates divided by two, and sum of up and down divided by two.
 - And how do you know a square is inside the polygon or not?
 - We will assume the center point is inside the polygon. This is true in our case, but it won't be true in all cases so in such case a different method would need to be applied. First we draw the polygon, say in the color black. We said the center point is inside the polygon. The square that is its neighbor immediately on the left on the same line is tested - if its color is black, then it is on the polygon and we have reached the limit of the chess piece for that line on the left side. Otherwise, we can fill its color as desired. We check the point above the center point, the point to its right and the one below. Then when that's done, we have to check the neighbors of these points as well. This ends when all the points we are testing are black or of the color to be filled.
- Mike was thinking this through. He never had really asked himself how computers could fill up polygons. Sarah did not get every detail, but it looked close enough to the way she was filling up a flower - starting from the center and avoiding to touch the border of the flower.
- This can be a lot of work for the computer to fill up large figures? Mike asked.
 - Yes. There are more complicated and more efficient algorithms but it can still be a significant amount of work. That's why video games with very powerful graphics forced engineers to design computers with always faster graphics processors.
 - Would it be possible to simply take a photo of the pieces and reproduce it on the screen? Why do we need to calculate polygons and fill them up one by one?
 - You're correct. This is called a bitmap image, as opposed to our vector graphics discussed here. A bitmap image can be very realistic. On the other

hand it is difficult to adjust its size - it will look not as good if you try to show it much larger than it was originally, because you don't have these details. With a vector image, you can set the color you want, and the size you want. It practically doesn't matter. That's very useful when your program has to draw on screens of very different resolution - small screen like your cell phone, Sarah, or a large screen like the one of a very good computer.



Figure 2 - 2000 pixels wide



Figure 3 – 1000 pixels wide



Figure 4 – 500 pixels wide



Figure 5 – 250 pixels wide

Vector graphics can be drawn at different sizes

Liliane's notes

The program starts in the main part.

```
16 | public static void main(String[] args) {  
17 |     VectorScreen screen = new VectorScreen();
```

First we learn how to draw chess pawns by reading the content of the file that defines polygons representing chess pawns. The program itself knows how to draw polygons but it is the file “chess.svg” that teaches the computer how to draw useful polygons. We could draw any other shape (animals, birds, etc.) by writing the correct file.

```
18 |     screen.read("chess.svg");
```

Our graphics program is used to draw pieces on a chess board. We want to draw the initial configuration of a chess game. First we put a black rook at the top left corner (indicated with a single letter r for simplicity). It is followed by a black knight (letter n). And so on. Notice the rows covered with pawns (letter p) or empty (space character) and the white pieces (capital letter P, R, etc.). It is easy to draw any possible chess configuration by putting these characters at the right place.

```
19 | String board = "rnbqkbnr" + "pppppppp" + "  
20 | " + "      " + "      " + "      " + "PPPPPPPP" + "  
   | "RNBQKBNR";
```

The program is then ready to draw the chess board with all its pieces as demanded. We have thirty two pieces to draw and fill up.

```
21 | screen.drawBoard(board);
```

And finally we ask the computer to show our artwork on a portion of the screen so we can admire our work.

```
22 | screen.setVisible(true);
```

See also the following related activities

- Chess
- Web Browser

To go further

- Early 1950s – machines use X-Y displays. Electron beam of the display monitor is controlled to directly draw segments (vectors) while the rest of the screen remains black. Such displays were used till 1999 for US air controls.
- 1963 – Ivan Sutherland creates an innovative graphics program called Sketchpad. Sketchpad runs on a Lincoln TX-2 computer at the Massachusetts Institute of Technology. The geometric data is structured into objects and instances that can be modified using a light pen.
- 1984 – John Warnock and Chuck Geschke create the PostScript language. PostScript describes everything in term of lines (vectors) and Bezier curves. PostScript fonts could be printed at very different sizes

- with very good quality. Till today almost all printers understand the PostScript language.
- 1996 Macromedia releases a vector-based animation software Flash 1.0. It becomes a widely popular format to add animation and graphics to web pages.
 - 2001 – World Wide Web Consortium publishes the first specification of Scalable Vector Graphics (SVG). SVG is a language for describing two-dimensional graphics with a tag format (XML), unlike the Flash format which is binary (not human readable). Recent Web browsers now support this format (e.g. Mozilla 1.5).

Exercise 1: Drawing the Chess board at different sizes and in different configurations

The main reason for using vector graphics is to be able to draw the same shape in a small or large screen. Vector graphics are expected to look great on a state of the art ultra wide screen and on your tiny cell phone. Let's try it out.

Look up line 27 of the VectorGraphics program.



```
27 |    int WIDTH = 1000, OFFSET = 25;
```

The width represents the size of the area on the screen the program draws to. Chess board is scaled in proportion to that width. The eight columns and rows have to fit within that area. The width can be set to any value you wish.

Edit line 27 so Chess board looks much smaller.



```
27 |    int WIDTH = 400, OFFSET = 25;
```

Run the program. Everything on the board has been properly resized. Try other values such as 2000 or 100. You'll probably decide to use a value between 400 and 1000.



Traditionally white occupy rows 1 and 2 and black rows 7 and 8. Let's forget about Chess rules for a moment and let's reverse black with white. Examine lines 19 and 20.



```
19 |    String board = "rnbqkbnr" + "pppppppp" + "  
20 |    " + "      " + "      " + "      " + "  
    "PPPPPPPP" + "RNBQKBNR";
```

This sequence of characters defines the content of any of the sixty four positions on the Chess board. “rnbqkbnr” represents row 8. To flip flop black and white replace lines 19 and 20 with the two following ones.



```
19 | String board = "RNBQKBNR" + "PPPPPPPP" + "  
20 | " + "      " + "      " + "      " + "  
   | "pppppppp" + "rnbqkbnr";
```

Run the program and verify black now shows up in rows 1 and 2. You see how you can configure any position. A Chess board showing up on a screen may not be as satisfying as a beautifully handcrafted wood board. On the other hand it is possible to invent configurations you cannot encounter with a traditional chess set.



Replace lines 19 and 20 with the two lines below. That is a fun game to play, especially if you are a beginner or a little tired of the traditional chess rules...



```
19 | String board = "      k      " + "qqqqqqqq" + "  
20 | " + "      " + "      " + "      " + "  
   | "qqqqqqqq" + "      K      ";
```

Follow your imagination. After you complete the last activity you will be able to play your own ‘Chess’ with your own favorite initial configuration. This one with eight queens on each side is simple and quite original to play...



Exercise 2: a failed experiment with Pac Man

Let’s draw each Chess pawn with a Pac Man figure. That will give a fun and original look to our Chess board. How do we vectorize Pac Man? How do we draw Pac Man with small lines?

Pac Man figure is easy to draw on a piece of paper. Draw a circle with a protractor or using a circular glass. Add two lines representing an open mouth. See the figure below.

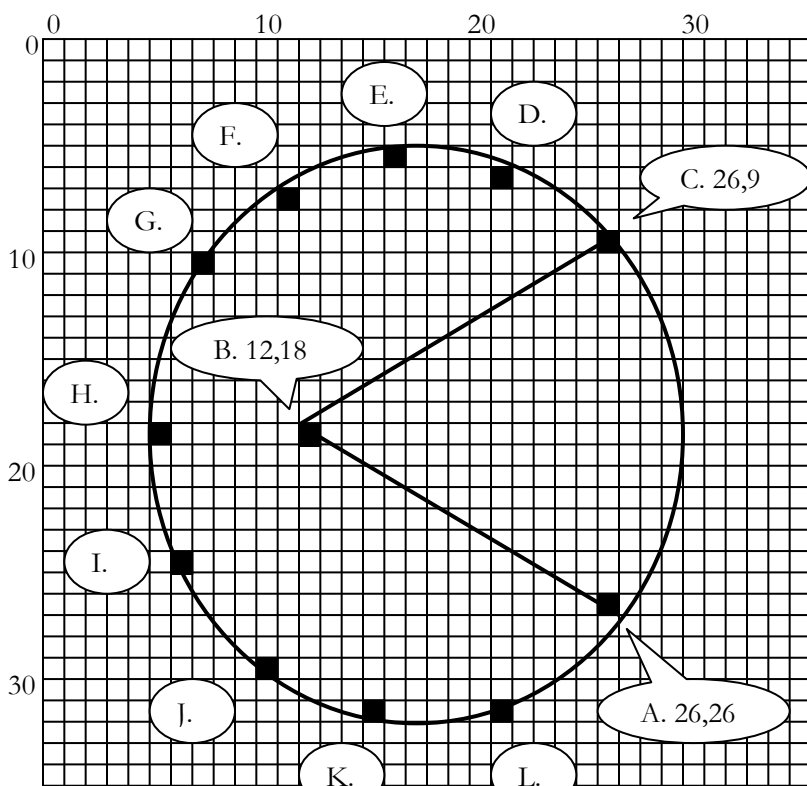



Figure 6 – Pac Man Vector drawing

We approximate the circle by picking up a few points from that circle and connecting them with straight lines. In the figure above Pac Man is represented with twelve points noted A to L. You can decide to use the same points or take different ones. The more points you take, the better the circle will look like on the screen. You may have noticed the points are not exactly on the circle. This is normal and part of our vectorization approximation. A circle is a perfect geometric figure which just does not exactly fit into a screen composed of tiny squares. But an approximation is useful enough...

The file 'chess.svg' defines all the points used to draw the Chess pieces.

Line 4 defines the pawn. Let's replace it with the coordinates of the points representing our Pac Man. Notice the first two numbers are the coordinates of point A and the next two numbers the coordinates of point B and so on. The last two numbers put us back where we started on point A. Note A and C are not directly linked because Pac Man must have an open mouth. 



```
4 | <polygon>26,26,12,18,26,9,21,6,16,5,11,7,7,10,5,
   | 18,6,24,10,29,15,31,21,31,26,26</polygon>
```

Work is done. Run the program VectorGraphics. It reads our modified file ‘chess.svg’ and we expect to see pawns looking like Pac Man.



And what actually happens? No Pac Man shows up on the screen. The computer does not like at all our new figure. After you give it a try for a few seconds, stop the program. It just won’t work. Experiment failed...

Can you guess what is happening? First check you have correctly modified the file ‘chess.svg’ as described previously. You must have all the points for the pawn on a single line. But that is not the problem here.

Our program fills the Pac Man shape with a color for a white pawn and with a different color for a black pawn. It starts from the center of the Pac Man. The center of the Pac Man shape is somewhere to the right of Point B, i.e. outside of the Pac Man shape itself represented by the segments connecting A, B, C, .. L, A. So the program tries to fill up the entire screen and ends up going beyond the screen which makes the computer unhappy. And nothing is drawn.

The easiest way to fix this problem is to modify the mouth of our Pac Man so that the center of the circle is inside the Pac Man. Let’s move point B further to the right from 12,18 to 18,18. Edit the line 4 of file ‘chess.svg’ one more time. Only the third number needs to be modified from 12 to 18.



```
4 | <polygon>26,26,18,18,26,9,21,6,16,5,11,7,7,10,5,
   | 18,6,24,10,29,15,31,21,31,26,26</polygon>
```

Run the program VectorGraphics one more time. It now draws every pawn with a Pac Man shape. Our experiment is a success.

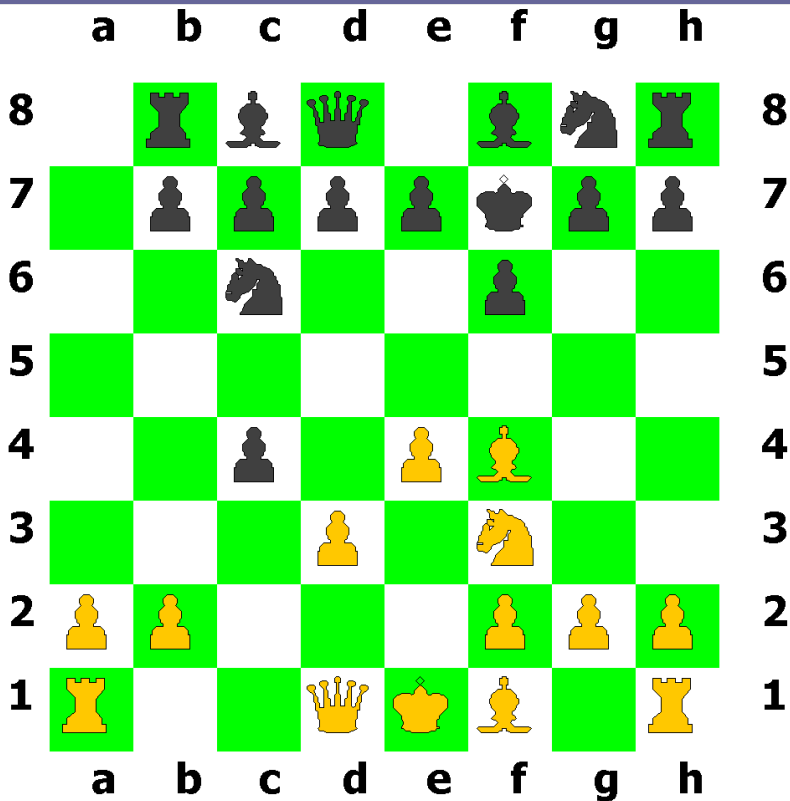


If you wish to draw Pac Man with a nicer circular look, use more points. More complicated svg programs also know how to draw circles and arcs in addition to lines. With such advanced vector program you’d describe the Pac Man shape with an arc whose origin is point B, radius is BC and two angles describing where to start the arc from at point C and where to terminate it in point A.

Activity 11 - Chess

Alex was returning from the basement. He was bringing back a beautiful chess board made from solid wood. “My father handcrafted the pieces using pines from the mountain.” Alex explained. “Notice how the king piece has a distinct nose. It's not Lagardere’s nose, but it is certainly similar to my father’s prominent one. Mike, how about teaching your sister how to play Chess? Then we will explain it to our computer.”

Computer executes the program Chess.java



Please enter white move:
d3d4

Mike and Sarah were seated on both sides of a very sturdy table made from oak. It had also been made by Alex's father who was a very good carpenter. Alex was observing Mike giving instructions to Sarah, sometimes interrupting to clarify a point and give a piece of advice.

"Sarah, take the white pawns. This tall piece here is your king. Here in black, that's my king. You win if you capture my king. You lose if I capture yours." Mike explained while placing the pieces on the board. "Each side symbolizes an army, you see. Each piece moves differently. The king can move in any direction, by one square. The queen is very mobile. She moves in any direction, any number of squares. So take good care of your queen, you don't want to lose her. You have two rooks, do you see them?"

"They look like two chalets to me? Alex?" asked Sarah.

"Yes, my father did it after our home. They are still rooks for the sake of the game, Sarah." answered Alex. Mike shook his head and went on.

"Rooks move vertically or horizontally, any number of squares. Bishops are here. They move diagonally. And knights..."

"Knights look like squirrels. They are very cute."

"So I was saying, the squirrel, err, the knight is a little bit complicated. It jumps. One step horizontally followed by one step in diagonal."

"This makes sense if that's a squirrel. It doesn't feel right for a knight."

"And you have eight pawns. They move one step forward. They take along the diagonal. If there's something in front of them, they are stuck."

"Weird. I wonder who came up with such rules."

"There are a few more moves, but that's enough for our first game. You have the white pawns, please begin."

Alex was now looking at Sarah attentively. His young student was frowning with her eyes focused on the chess board. She was considering which pieces she could move. Finally she tried something. Mike told her that move was not allowed. A couple of mistakes later, Sarah and Mike were engaged in a very intense play.

"Check mate." Mike announced, falsely modest. Sarah had accidentally let Mike capture her queen. Her chances to win had turned very thin as a result. Alex gave her a few tips.

"First examine the moves you can make. Try to take your opponent's pieces so the opponent has fewer pieces left to defend their king or attack your king. When you consider a particular a move, be cautious. Review the possible moves your opponent can do if you do that move. Are you going to lose a piece that's more valuable than the one you are taking? Mobile pieces like the queen are very precious. A pawn doesn't matter too much compared to other pieces. Of course, you must always protect your king."

"But Mike also tries to guess what I am doing. So how can one win?"

“The person who can examine the possibilities one move deeper than the other player is likely to win. It’s also about strategy and the human factor. Amusingly, Mike blinks his eyes every time he has setup a trap to take one of your pieces.”

“That cannot be true! I don’t blink. I will wear sunglasses then.”

“How do you like that game, Sarah?”

“It’s complicated. There are so many rules.”

“True. It is a very ancient game, played probably nearly two thousand years ago in Persia. The game is very interesting because each play is unique. There are so many ways to position the pieces. We often have to choose from more than twenty moves. That leaves a lot to think through.”

“Can’t the player who makes the first move always win?”

“It’s a very good question, Sarah. Unfortunately, till today, we don’t know. We believe white has an advantage - in particular it gets to decide how to open the game. But there is no proof white can always win or black can always force a stalemate. Unlike very primitive games like Tic Tac Toe.”

“When the first computers were created, we believed it would take one hundred years before a machine can truly play chess. It turns out both machines and Chess programming techniques evolved at a much greater pace. In 1997 a machine called Deep Blue defeated the world chess player, Garry Kasparov. For sure, it wasn’t fair to the human. The machine had been taught all the moves played by Kasparov throughout his life. On the other hand, Kasparov knew very little about the machine’s playing style. I’d like to see what would happen if you leave the machine and the human player together for a week, and then run the tournament.”

“You’re saying humans can learn quickly how to exploit a program’s weakness?”

“Yes. You are going to be able to judge by yourself. Let’s write our own chess game.”

Sarah smiled. Everything looked so simple with Alex!

Alex first suggested a way to memorize the positions of the pieces on the chess board in the computer. The program needs to know how to remember which pieces are available, black or white, and where they are located. A simple way is to use a string. Each character in the string represents a particular place in the board. There are sixty four positions on the board, so our sentence will be sixty four characters long. An empty location has no character - or rather it uses the space character ‘ ’. Every other type of piece uses a special letter.

Table 1 - Characters used to represent Chess pieces

Piece	White player	Black player
-------	--------------	--------------

King	K	k
Queen	Q	q
Rook	R	r
Bishop	B	b
Knight	N	n
Pawn	P	p
Empty	“	“

Alex decided the first character of the sentence would represent the bottom left part of the board. Chess players call this position ‘a1’. The second character would represent the position ‘a2’ and so on.

“A computer can store a large number of sentences. So we will be able to store many Chess boards. This will be very useful when searching for the best move.” explained Alex.

Alex asked Mike to take care of showing the Chess board on the screen. Mike remembered they had previously learnt how to draw vector graphics representing a chess board. Alex warmly approved Mike’s choice to reuse the work from [Activity 10 – Vector Graphics](#).

“Reusing other programs is very important, Mike. It’s too complicated to do everything from scratch every time. It’s sometimes useful to learn how to do things. It is often impractical. Our chess program will only take care of finding the best moves. The drawing on the screen has already been done in our VectorGraphics program.”

“Now, we are going to undertake the first important part of the program. Sarah, what was the first thing Mike taught you?”

“He told me how each piece moves.”

“Indeed. An important part of the program is to teach the machine which moves are allowed for a given configuration. For a given sentence representing pieces on a chess game, what sentences would represent legal moves.”

“How do we do that?”

“The program needs to examine every character in the sentence. If it isn’t empty, it examines which kind of letter it is. Then check how the piece represented by that letter is allowed to move. If it can do a move, it creates a new sentence which is almost a copy of the previous sentence - except that the piece has been moved. We repeat for all the positions in the sentence. After that, we have a list of sentences that represent all the possible moves.”

explained Alex. “Of course it is fundamental to ask if you need the moves to be computed for black or white.”

It took a little bit of time to get this part to work. There were many rules dictating how to move pieces on the Chess board. “Notice how computing the queen’s moves is the same as computing the moves for the knight and for the bishop combined” noticed Alex. “We can reuse those rules for the queen.”

“Are we done, now?” asked Sarah. “We’ve made good progress, Sarah. And it is time we test what we have achieved so far. Let’s see if we have made some mistakes. Our program now knows how to display a chess board. It knows how to compute all the possible moves that can be played. However it still does not know how to pick up a good move among all the possible moves. For now, let’s just ask the program to randomly pick one of the moves.”

Mike followed Alex’s suggestion. The program displayed the initial board on the screen. Then it waited for the human’s first move. Human entered four characters like e2e4. The piece located at e2 was requested to be moved to e4. The program computed every possible white move and verified move e2e4 was one of these legal moves. The program then updated the board with that move. The board was described by a new sentence that included the white move e2e4.

The program now needed to respond to the white move. It calculated all possible black moves from that new sentence. It randomly picked one of these moves and created yet another sentence with both white and black moves. That new sentence was displayed to the screen. Thanks to the work in [Activity 10 – Vector Graphics](#), it looked like a nice chess board, not just a series of sixty four characters. The computer and the human use different representations of the Chess game, yet they still play the same game regardless of whether they use strings or nice graphics. White and black had completed one move. Program then waited again for the human to enter their move, and so on.

“Very well, Mike. Just examine if there is still a king after every move. In that case the game is over and you will display the winner.”

“Oh right, of course. Thanks, Alex.”

The program was started. “Sarah, May I ask you to try it out?”

Sarah entered her move using the keyboard. The program refused it. Sarah got upset.

“Sorry, Sarah, the program is right. You can’t move the bishop over your own pawn.”

“Oh all rights Just need to say so.”

Sarah and the program exchanged a few moves. Then the machine moved its rook to a new position. Actually no, it had not exactly moved it. It had kept it to its initial position, and it also had added a new rook to the new position.

“Well, now that’s not fair.” Sarah bitterly complained. Mike was a little bit confused. Alex and he checked through the program. Sure enough, the part in charge of creating the new sentence representing the move of a rook was incorrect. It had a bug, like programmers say. It correctly put the character at

the final position in the new sentence, but did not remove it from its initial location. That was easy to fix.

This time, Sarah won the game.

"I take it that at this point Sarah would win every game against our program."

"I'm sure getting good at it."

"So it's now time to teach the program how to do better than randomly pick its moves."

"How do you do that, Alex?"

"The same way you play yourself, Mike. You evaluate if a position is better than another. If so, then you do the move that leads you to it."

"But how do you explain that to a machine?"

"The second big part of a chess program is called the evaluation function. That part examines a particular chess board and decides how good it is for the white or the black."

"So if the chess board has a white queen but the black queen was taken earlier, it would say things are going well for the white?"

"That's the idea. Follow my instructions, Mike. We are going to write that evaluation function."

Alex's evaluation function was given a sentence representing pieces on a chess board. It computed a number from it. A large positive number meant white were winning or had a strong advantage. A large negative number indicated black had the advantage. If the number was near zero it was undecided whose side had the advantage.

"We will use the standard values for each piece. Examine every character in the sentence representing the chess board. If you find a 'Q', that's a white queen. Add nine points. That's good for the white. If you find a black queen, subtract nine points. If the game has both white and black queens, they cancel each other - nine minus nine. We do that for all the characters. Bishop and knight count for three, rook for five. A pawn counts for one."

"What about the king?"

"Well kings are special. If the chess board is missing the white king, black has won. So we will give a very large negative number to signal it is the best possible move for the black. On the other hand, if there is no black king, we score a very large positive number - the best possible move for the white. If both kings are present, we don't add or remove anything."

"At the beginning of the game, every player has the same amount of pieces of each kind."

"So that configuration is noted zero. Each white piece is countered by a black piece with the same value."

"For the first move, nobody can take any piece from the other side. So every player still has the same equal amount of pieces, isn't it? Therefore, each possible move is still represented by a sentence that is worth the same value - zero."

“Absolutely true, Mike. For now let's try it out, and we will discuss improvements later.”

Sarah was again asked to play against the computer. “Why is it always me?” her eyes seemed to ask. Sarah won. She had not given much chance to the computer. A few times, the computer had aggressively taken her pieces, but she was able to immediately take the computer's piece back - the computer even sacrificed its queen to take Sarah's pawn!

Mike wasn't much impressed at all. Yet Alex was showing no sign of surprise.

“Now comes the third important part of a chess program. It is known to programmers as the mini max technique. Forget its name. It's again very similar to how you Mike play chess yourself. First you examine your possible moves right?”

“Yes.”

“And what do you do after you decide you like one of the possible moves, say you want to take a piece of your opponent?”

“I check if the opponent can take one of my pieces after I move my own piece. When I move my piece, I may leave some of my pieces unprotected, for example my queen or my king. Not a good thing.”

“If that's the case, you don't do the move - you don't take the piece?”

“I do it if the piece I'm taking is more interesting than the one I might lose. Sometimes the opponent does not realize they can take a piece after my move. So I can take some risk too.”

Alex smiled. “Well, we don't like our program to take any risk. So we will always assume the opponent would play the best possible move - meaning the best possible move the program would itself play if it was just switching side. The program plays black. Examine all the moves the black can do. There are three moves. One of these moves is the black queen taking a white pawn. The two other moves don't take any white pawn. Our evaluation function scores the first move higher than the two other ones - the white has one pawn gone. That's how our program decided to play the first move as a result.”

“And then I took the program's queen!” laughed Sarah.

“Exactly. Now what we are going to teach the program is how to examine all the possible moves the white can do in response to its first move - the possible answers after the pawn is taken by the black queen.”

“One of the white moves is to take the black queen.”

“And that is very bad for the black. So if the black do their first move, take a white pawn with their black queen, they know the white can then take their black queen. This leads to a chess board that has a bad score for the black - the evaluation function will give at least an eight point advantage to the white - nine points because the black has no queen, less one point because the white suffered the loss of a pawn.”

“I'm starting to understand.” said Mike.

Mike wrote that third part of the chess program under the supervision of Alex. That was the most interesting part of the program. Black calculated all their legal moves - they had a list of sentences representing how the chess board looked like for each of their moves. Then black examined each possible move individually. For each possible black move, it now calculated each possible move white could answer with. Evaluation function was used to decide the move that was the best answer from the white's perspective. This gave the final and more accurate evaluation of the black move.

Sarah was getting a little bit tired about all this. Alex and Mike insisted she played against the new program. For the first time, she lost to the computer.

Mike and Alex were pleased. Sarah wasn't. She found an excuse to leave the room and went to bed. It was getting late, as a matter of fact.

Mike did a game. And won. He did a second game. He won again.

Alex modified just a couple of lines in the program. "Try again, Mike."

Mike obeyed without much expectation.

The computer won the next play. And the one following.

"What did you do, Alex? It was so easy to beat, and now it's playing so well?"

Alex padded Mike's shoulder. "I did almost nothing, Mike. You taught the machine how to find the best move by examining all the answers of the opponent to any of the moves the machine can make - and picking up the move that leads to a situation where the best opponent's answer is still the most favorable to the machine. What I did is push the same logic further. Examine all your moves. Examine all the possible answers for every move you can make. And now examine all the possible answers you can do to reply to any of the answers your opponent can do to your first move. Examine three moves ahead. Four moves ahead."

Mike was now really excited. "Of course! I did notice the program was taking a couple of seconds or even half a minute to answer sometimes. Previously it was always instantaneous. It must have been busy calculating all the answers to the answers to its moves. And picking up the move that yields to the best black answer to all the white answers to all its possible moves."

"Something like that, yes, Mike."

Mike changed the line Alex had modified earlier. He started another play.

"Well it's not working anymore or what?" asked Mike. The screen did not show any move. They waited a few minutes. "Is it broken?"

"No Mike. You asked the program to examine all the possible moves at a greater depth than what I had set previously. Instead of taking a few seconds, it is now literally exploding the amount of moves to examine. Imagine every time there are about twenty possible moves. There are about twenty by twenty possible answers to calculate at depth one - the white answer to your move. That's four hundred possible chess boards - or sentences. Not a big deal for our computer. At depth two we are checking each black answer to each possible white answer. Four hundred multiplied by twenty again. That's eight thousand

chess boards to examine before picking up the best solution at this depth. For depth three, eight thousand multiplied by another twenty. We are now asking the machine to evaluate one hundred and sixty thousand configurations. Depth four requires over three millions, depth five, sixty four millions moves.”

"Every depth increase is likely to take about twenty times more time to be computed. I get it"

The machine did its move six minutes later.

“Even computers can't resolve the chess game and compute all the possible moves in their memory in a reasonable amount of time for example that white moves and wins in two hundred and six moves. That's a dream beyond our technology and current algorithms. In the short term, that makes playing chess still pleasurable.”

“Alex, could our program beat Kasparov?”

“No, Mike. Even on the best machine available today, it would not. The machine that beat Kasparov used similar technique to ours, with many more refinements. It knew how to avoid calculating many times the same moves - if you move your pawn first, then your queen, it can lead to the same configuration than if you had move your queen first then your pawn. Our program simply evaluates these different possibilities even though they regroup to the same chess board. Also these machines have a much better evaluation method. They literally know millions of previous games by important players - and Kasparov in particular. So to some extend these programs also behave like the Translator program we did earlier - they were taught how humans played their best move so they know how to find them back and evaluate their answers to it.”

Mike and Alex started to play against each other. And Alex won - even though he was also a bit tired and was not calculating sixty four millions moves in six minutes. He knew certain things Mike did not know - how occupying the center of the board gives you the advantage toward the middle of the game - how to deploy your pieces so they are free of their movements and can paralyze the opponent. It was fun programming a computer to play chess. Playing against a computer could be good training. But playing against Alex and not a machine, thought Mike, was way more pleasurable. Oops! He just reminded himself – don't blink your eyes!

Liliane's notes

The program starts in the main part.

```
9 | public static void main(String[] args) {  
10 |     Chess chess = new Chess();
```

And it will keep repeating the lines that follow.

```
11 | while (true) {
```

Display the chess board onto the screen. This uses the nice vector graphics we learnt about in the previous activity so Mike and Sarah did not do all the chess drawing for nothing. We can reuse their artwork.

```
12 |     chess.displayBoard();
```

First white player must move a white piece. In our program the white player is always the human so the program waits for the user to type in the position of the piece to move and the position to move it to.

```
13 |     chess.whiteMove();
```

The program examines if black or white has lost (their king is missing). This could have happened if the human player just succeeded a check mate. If either side has lost, the game is stopped; nobody else can move pieces any further.

```
14 |         if (chess.gameOver())
15 |             break;
```

Otherwise it is now the turn of the black player to play. Black player is represented by our program which examines many possible moves and decides which one it finds to be the best.

```
16 |     chess.blackMove();
```

After the program moved a black piece, we check again if one king is missing (that would happened if black takes the white king and wins). If the game isn't over yet, then it repeats itself from the above. White plays, then black etc.

```
17 |         if (chess.gameOver())
18 |             break;
19 |     }
```

We arrive here when one king has been taken. If the black king has been taken, then the human player representing the white is the winner. Otherwise our program is the winner.

```
21 |     chess.displayWinner();
```


See also the following related activities

- Vector Graphics
- Slow or Fast

To go further

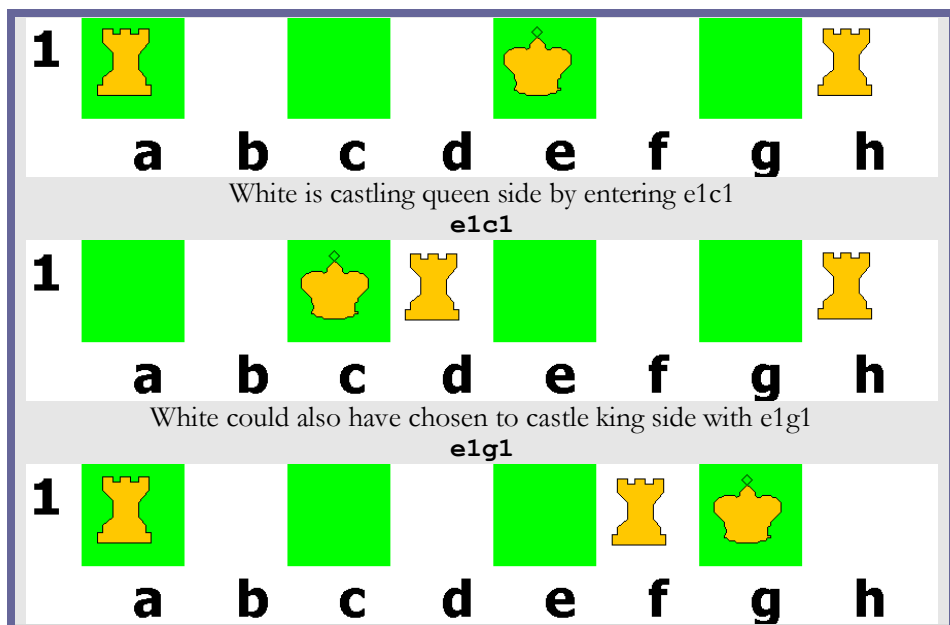
- 4th century – First traces of Chinese chess named Xiangqi. This game is still widely popular today in Asia.
- 6th century – Modern chess is believed to be derived from a Persian game called Shatranj itself derived from a game named Chaturanga and originating from India. Others allege Chess derives from earlier Xiangqi game.
- 1928 – John Von Neumann introduces the minimax theorem which is an important foundation of game theory.
- 1947 – Alan Turing designs the first chess program (but no machine can run it yet).
- 1950 – Claude Shannon publishes an article explaining chess can be programmed through ‘brute force search’ (like this chess program) using the minimax algorithm, or through strategic artificial intelligence (examine only very few moves with deep understanding of what is a good or a bad move).
- 1957 – Bernstein writes the first complete chess program for an IBM 704 computer.
- 1997 – Chess computer Deep Blue defeats world chess champion Garry Kasparov.

Exercise 1: implement modern chess castling moves

During the 15th century, European players added special moves to make it easier to defend the king. Modify the chess program so that the moves named castling are recognized.

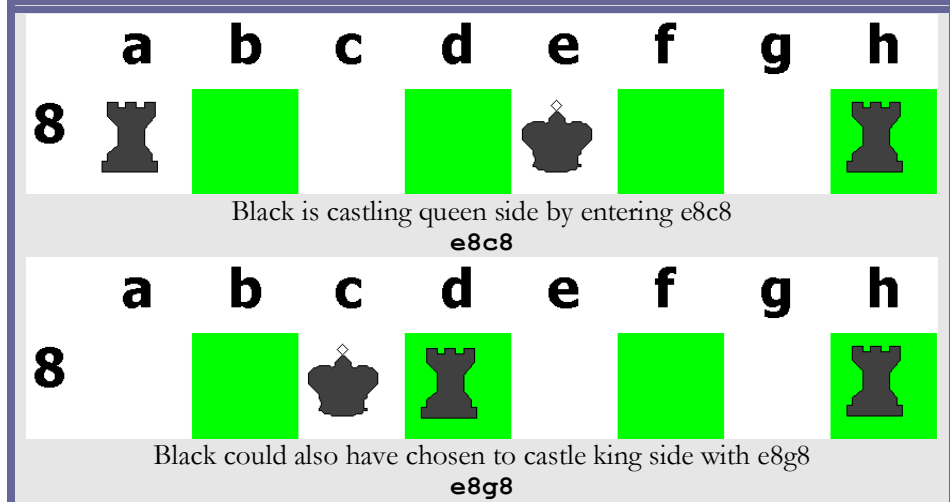
A king and a rook can do a spectacular move together if that is their first move and if there are no pieces between them. King moves two squares to the left or to the right, and one of the rooks crosses its path. The details of the moves are shown for the white king below.

Computer executes the program Chess1.java – White castling moves



Castling moves from the black side are presented here. They are similar to the white side. Of course moves now occur on row number eight.

Computer executes the program Chess1.java – Black castling moves





Program Chess.java needs to be taught these very important moves. When user enters e1c1, program will recognize a queen side castling and will also move the left rook accordingly.

Edit the function `getNewBoard`. This part creates a new board based on the move requested by white player. It now needs to recognize the new castling moves. Look for the following line.



```
75 |         int y2 = column.indexOf(move.charAt(3));
```

And insert these new lines next to it.



```
76 |         if (isQueenSideCastlingMove(board, x1, y1, x2,
77 |         y2))
77 |             return getQueenSideCastling(board, x1, y1);
78 |         if (isKingSideCastlingMove(board, x1, y1, x2, y2))
79 |             return getKingSideCastling(board, x1, y1);
```

Edit the function `getPossibleMovesKing`. It also clearly needs to learn the newly authorized castling moves. Search for the lines below.



```
297 |     String[] getPossibleMovesKing(String board, int x,
298 |     int y) {
298 |         List result = new ArrayList();
```

Once you have found the previous lines, insert the next lines right after them.



```
299 |         String castlingBoard =
300 |         getQueenSideCastling(board, x, y);
301 |         if (castlingBoard != null)
302 |             result.add(castlingBoard);
303 |         castlingBoard = getKingSideCastling(board, x, y);
304 |         if (castlingBoard != null)
304 |             result.add(castlingBoard);
```

If queen side castling or king side castling is allowed for the king on the given board, then we are adding these moves to the list of the king's possible moves. With that done, the program is now ready to accept castling moves from the white player or to do castling moves itself for the black side.

Well, we still need to actually define the details of the castling moves. We need to add a few more rules. You can insert the following code after the function `getPossibleMovesKing`.



```
315 | boolean isQueenSideCastlingMove(String board, int  
    | x1, int y1, int x2, int y2) {  
316 |     String piece = get(board, x1, y1);  
317 |     if (!isKing(piece))  
318 |         return false;  
319 |     if (y1 != y2 || (y1 != 0 && y1 != 8))  
320 |         return false;  
321 |     if (x1 != 4 || x2 != 2)  
322 |         return false;  
323 |     String piece1 = get(board, 0, y1);  
324 |     if (!isRook(piece1) || isDifferentColor(piece1,  
    | piece))  
325 |         return false;  
326 |     String piece2 = get(board, 1, y1);  
327 |     String piece3 = get(board, 2, y1);  
328 |     String piece4 = get(board, 3, y1);  
329 |     if (!isEmpty(piece2) || !isEmpty(piece3) ||  
    | !isEmpty(piece4))  
330 |         return false;  
331 |     return true;  
332 | }
```

We have taught the program how to recognize if a move on a board represents a legal queen side castling move. It checks if the piece is a king and a few other things like there are no pieces between the king and the rook.

Program next needs to learn how to move the pieces on the board after a queen side castling. First check if the move is legal. Second, move the king to the second column. Third move the rook to the third column. Note this works for both white and black. In the first case the argument `y` will be zero. In the latter case argument `y` will be equal to seven.



```
333 |  
334 |     String getQueenSideCastling(String board, int x,  
    | int y) {
```

```

335 |     if (!isQueenSideCastlingMove(board, x, y, 2, y))
336 |         return null;
337 |     String newBoard = move(board, x, y, 2, y);
338 |     String newBoard2 = move(newBoard, 0, y, 3, y);
339 |     return newBoard2;
340 | }

```

Details are subtly different but once you know how to program the queen side castling moves it is not too difficult to program the king side castling rules.



```

341 |
342 |     boolean isKingSideCastlingMove(String board, int
343 | x1, int y1, int x2, int y2) {
344 |         String piece = get(board, x1, y1);
345 |         if (!isKing(piece))
346 |             return false;
347 |         if (y1 != y2 || (y1 != 0 && y1 != 8))
348 |             return false;
349 |         if (x1 != 4 || x2 != 6)
350 |             return false;
351 |         String piece1 = get(board, 7, y1);
352 |         if (!isRook(piece1) || isDifferentColor(piece1,
353 | piece))
354 |             return false;
355 |         String piece2 = get(board, 5, y1);
356 |         String piece3 = get(board, 6, y1);
357 |         if (!isEmpty(piece2) || !isEmpty(piece3))
358 |             return false;
359 |         return true;
360 |     }
361 |
362 |     String getKingSideCastling(String board, int x,
363 | int y) {
364 |         if (!isKingSideCastlingMove(board, x, y, 6, y))
365 |             return null;
366 |         String newBoard = move(board, x, y, 6, y);
367 |         String newBoard2 = move(newBoard, 7, y, 5, y);
368 |         return newBoard2;
369 |     }

```

So we are done and can now play modern chess game. Run the program and try these new moves.



We have written a lot of code. What happens when a lot of code gets added? It has errors – familiarly also called bugs from the old days when real bugs ate electrical cables and blocked computers. So we need to test the new moves very

well. Try in particular the queen side castling move when b1 and c1 squares are empty but d1 is still occupied by the queen. After you enter e1c1 the program should not allow the castling because the queen is blocking it. Bug... The queen is eaten and the castling is accepted. Oops. We need to fix this bug in the function `isQueenSideCastlingMove`. Here is the corrected version you should use.



```

315 |   boolean isQueenSideCastlingMove(String board, int
      |   x1, int y1, int x2, int y2) {
316 |       String piece = get(board, x1, y1);
317 |       if (!isKing(piece))
318 |           return false;
319 |       if (y1 != y2 || (y1 != 0 && y1 != 8))
320 |           return false;
321 |       if (x1 != 4 || x2 != 2)
322 |           return false;
323 |       String piece1 = get(board, 0, y1);
324 |       if (!isRook(piece1) || isDifferentColor(piece1,
      |   piece))
325 |           return false;
326 |       String piece2 = get(board, 1, y1);
327 |       String piece3 = get(board, 2, y1);
328 |       String piece4 = get(board, 3, y1);
329 |       if (!isEmpty(piece2) || !isEmpty(piece3) ||
      |   !isEmpty(piece4))
330 |           return false;
331 |       return true;
332 |   }

```

It is so easy to make mistakes. That's the reason programs get tried many times before they can be trusted.

If you are a great fan of the Chess game, you will have noticed there are still many little rules missing here and there. You should by now be able to implement them yourself. You have reached that point when a student outpasses their teacher. Congratulations!

Exercise 2: Change the level and personality of the chess program

It is easy to control how smart the program is. Tell it how many moves ahead it should evaluate. Zero or a single move ahead makes for a very fast opponent who is great for beginners.



```

385 |   void blackMove() {

```

```

386 |     String[] boards = getPossibleMoves(board, false);
387 |     boards = evalBest(boards, 1);

```

You can ask the program to check two, three or four moves ahead if you are a good player.



```

387 |     boards = evalBest(boards, 2);

```

But the program starts to take a very long time to examine all the moves. So you can't just increase that number to any value even if your computer is a very fast machine. It gets exponentially more complicated every time you increase that value by one...



```

387 |     boards = evalBest(boards, 4);

```

A human player sometimes loses their concentration or gets tired. At times we really focus and want to defeat our opponent. We can make our program behave more like a human. We simply set randomly the level for each black move. Edit the blackMove function so it is as shown below.



```

385 |     void blackMove() {
386 |         String[] boards = getPossibleMoves(board, false);
387 |         int depth = new Random().nextInt(3) + 1;
388 |         boards = evalBest(boards, depth);
389 |     }

```

A random number between one and three is picked every time the program must do a move.

Play against the chess program. Do you notice the new behavior of the chess program? Sometimes it plays really well and takes its time. Sometimes it makes really silly moves. It's a fun opponent if you are a beginner in Chess.



Let's give some true personality to our program. Nothing is more attaching than some well identifiable playing behavior. Computers can be so annoying with their efficient but non imaginative, non daring gaming style. We humans are more impatient. Risk does not always pay off and can have serious consequences. But we are talking about a game here. We want some fun. Let's get inspired by the style of General Patton, Alexander the Great or Napoleon. 'L'audace, toujours l'audace' – 'audacity, always audacity!' said General Patton.

We will modify the evaluation function of the program so that it encourages very aggressive moves into the opponent's territory. This will make for furious game plays. Computer will take any chance to charge into the white's ranks.

Edit the function eval below.



```
437 | int eval(String board) {
```

Replace the following line in the eval function.



```
446 | int value = value(piece);
```

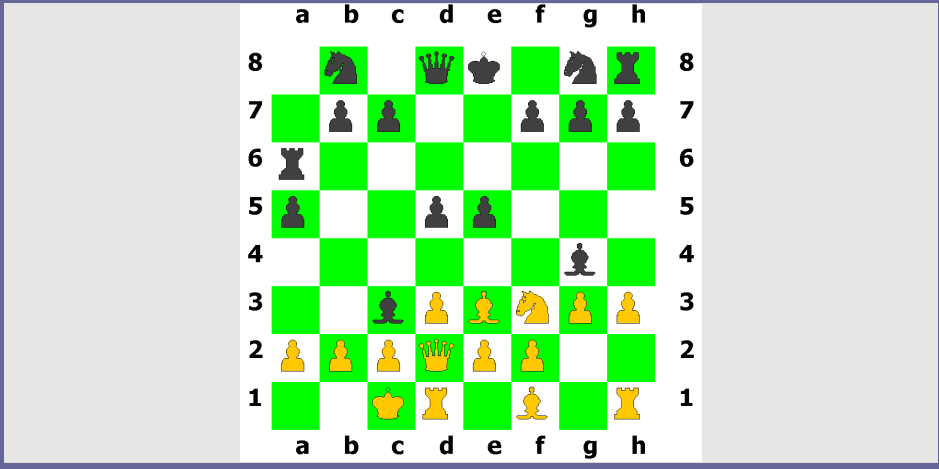
With the line below. Small change, isn't it?



```
446 | int value = value(piece) * 10 - y;
```

The evaluation function of the chess program determines its personality. That tiny change completely alters the computer's playing style. Let's see how it works for a black pawn. Standard value of a pawn is one. We multiply that standard value by ten, so that's ten points so far. Now we remove points unless the pawn is at the very bottom of the chess board – i.e. into the white territory. If the black pawn is at its initial location on raw six, pawn is only valued ten minus six points. Four points only. If it moves down two squares, it is scored six points. As a result, the computer is encouraged to aggressively move its pieces into the bottom part of the board. Here is a real example after only a few moves.

Computer executes the program Chess2.java – Black plays aggressively



Black moves its pawns two squares at a time – as fast as it can. It moved its bishops very close to the white pieces. Black is very daring. Sadly for them, white can take the bishop on c3 with their white pawn on b2 or their queen on d2. Black made that mistake because it did not calculate enough moves ahead the consequences of its aggressive tactic – the level varies randomly between one and three and black simply thought its second bishop on g4 would take the white horse on f3 and make up for the loss. But white pawn e2 will then take that bishop...

We can build different personalities into the Chess program. It becomes a matter of discovering that personality and taking advantage of its weaknesses – like when playing against a good old friend. Of course you can also think about writing a program that learns from your own strengths and weaknesses.

Epilogue

Summer went by very quickly. After giving a big hug to their animator Alex and all their new friends, Mike and Sarah took the bus that would take them away from the Alps. Liliane took the train with them from Grenoble to Paris. She also had to prepare to return to university. In the airplane flying back to Canada, the two teens were chatting about the many souvenirs they had collected.

Some eight hours later, the teens had crossed the Atlantic Ocean and were thrilled to meet their parents. Their mother asked them: “Are you hungry? Let’s all have dinner together.”

But Daddy looked uncertain: “I have a couple of projects to finish up. I don’t think I can.”

The two teens gave no choice to their father: “We will help you... Programming isn’t scary... It’s fun.”

Their dad started laughing. “Yes you are right, programming and science are fun, even though I sometimes forget they are. Let’s go to that restaurant and you will tell us everything about your summer...”

Index

A

Adobe Reader · 99
Al-Khwarizmi: Muhammad · 74
Applegate: David · 67
Aristotle · 86

B

Berners-Lee: Tim · 59, 98
Bernstein · 121
Bezier · 106
Big Bang · 87
Black hole · 87
Bohr: Niels · 20
Boone · 75
Brown: Robert · 35, 38
Brownian · 35

C

Castling · 121
CERN · 95
Charles III: Prince · 29
Clay: Mathematics Institute · 75
COM · 59
Cooper · 75
Copernicus: Nicolaus · 86

D

Deep Blue · 121

E

Einstein: Albert · 20, 38, 87

Emulator · 59
Engelbart: Douglas · 59
ENIAC · 20
Euler: Leonhard · 67

F

Fermat: Pierre de · 20
Fermi: Enrico · 30
Fermiac · 30
Firefox · 99
Flash · 99, 107

G

Galilei: Galileo · 87
Game theory · 121
GE-645 · 59
Gennes: Pierre-Gilles de · 38
Georgetown: University · 50
Geschke: Chuck · 106
GIMPS · 75
Google · 99; Translation · 50
Gosling: James · 59
Graph theory · 67

H

Hamilton: William Rowan · 67
HD 160691: Star · 90
Heisenberg · 20
Higgs · 99
Hoare: Tony · 74
HTML · 95
Hypertext · 98

I

IBM · 50

IBM 701 · 50
Internet Explorer · 99

J

Java · 59

K

Kasparov: Garry · 121
King · 52

L

Lazzarini: Mario · 29
Lincoln TX-2 · 106
Lisp · 58, 59

M

Mac OS · 59
Macintosh · 59
Manhattan Project · 30
Massachusetts Institute of Technology ·
106
McCarthy: John · 58
Microscope · 35
Microsoft · 59
Minimax · 121
Montulli: Lou · 98
Moore: Gordon E. · 74
Multics · 58

N

Nagao: Makoto · 50
Netscape: Navigator · 99
Neumann: John Von · 20, 30, 121
Newton: Isaac · 20, 87
Nine Chapters on the Mathematical Art ·
74
Nobel · 38

O

Och: Franz · 50

P

Pac Man · 84, 108
Pascal: Blaise · 20
PDP-11/20 · 59
Penzias: Arno · 87
Perl · 59
Philosophiae Naturalis Principia
Mathematica · 87
Pi · 24
Pollen · 35
Polymer · 38
PostScript · 106

Q

Quicksort · 74

R

relativity: general · 87
Revolutions of the Heavenly Spheres · 86
Ruby · 59
Russell: Steve · 87

S

Safari · 99
Schrödinger · 20
Seven Bridges of Königsberg · 67
Shannon: Claude · 121
Shell · 59
Sketchpad · 106
Smalltalk · 59
Space War · 87
Sun: Microsystems · 59
Sutherland: Ivan · 106
SVG · 107
Sweden · 67

T

Traveler salesman problem · 67
Turing: Alan · 121

U

Ulam: Stanislaw Marcin · 23
Unix · 59

V

Vectorize · 103
Virtual machine · 59
Voyager 1 · 87

W

Warnock: John · 106
Weaver: Warren · 50
Wilson: Robert Woodrow · 87
Windows · 59
World Wide Web · 98, 107

X

Xiangqi: Chinese chess · 121
X-Y display · 106

Y

YouTube · 99