Lecture 5

Project

The project requires you to build a system with a GUI for solving matrix puzzles. Including magic square and Sudoku. The user interface has to be flexible in what the puzzles can be and should allow new types of puzzles to be added easily in the future. The puzzles all use a matrix to represent the problem, but the matrices can have different types of elements and the games can have different constraints, rules, and objectives. In magic square for example have to make all rows/cols/diags add up to the same number, in Sudoku have to make certain areas contain the same integers.

In addition to the user interface that allows users to play the games, you also have to build solvers that automatically solve instances of Sudoku and Magic Square at the click of a button.

Project

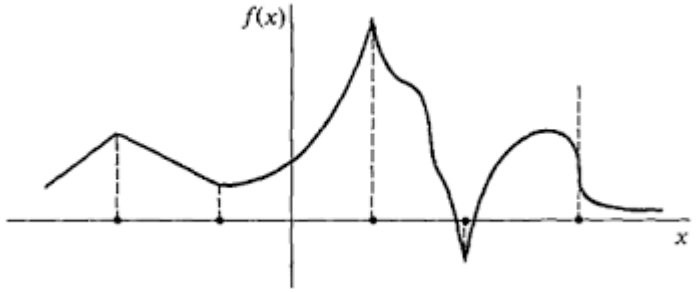
Object oriented design (already discussed).

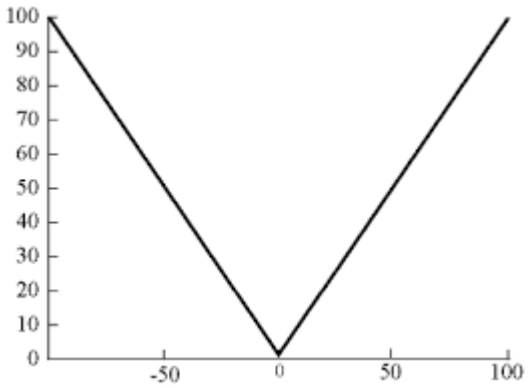
Graphical User Interface (GUI), later this lecture…

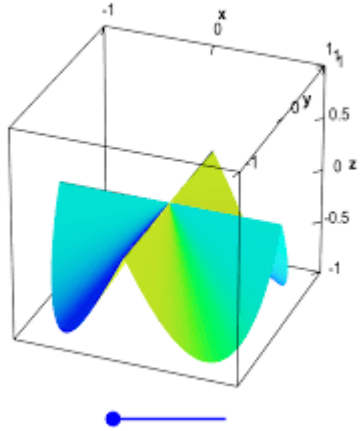
First, we will discuss some ideas for the solver(s).

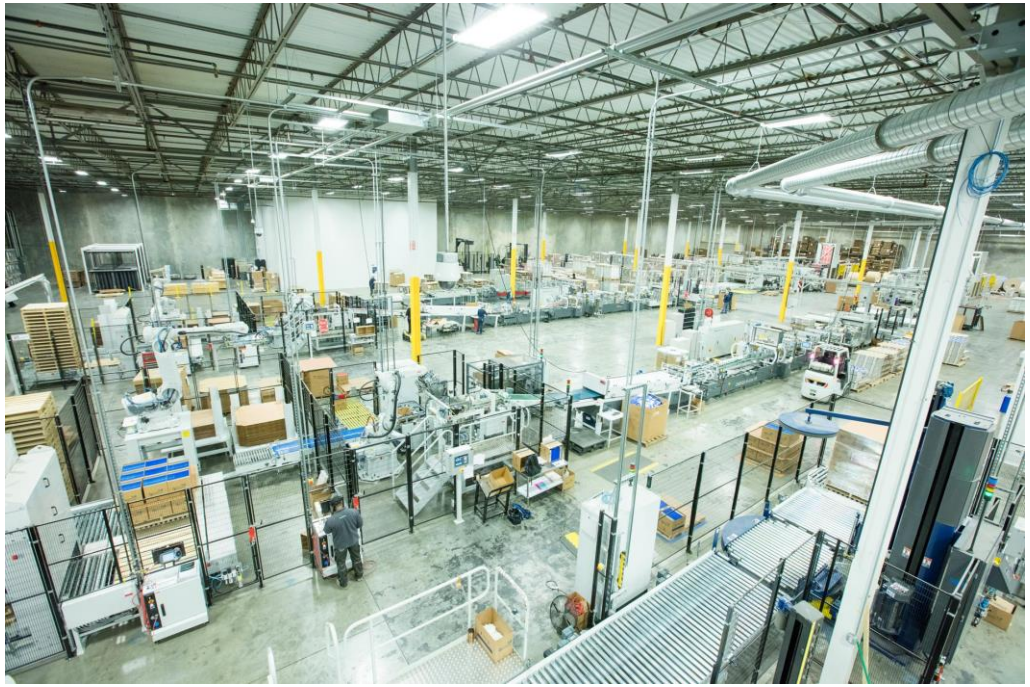
Problem Types for Solvers

- Non linear
- Non differentiable
- Non convex
- Large and complex









Combinatorial Optimisation Problems

- Set of variables
- Variable domains
- Constraints on variables
- Objective function

Definition 1.1. A *Combinatorial Optimization* problem $P = (\mathcal{S}, f)$ can be defined by:

—a set of variables $X = \{x_1, \ldots, x_n\}$;
—variable domains $D_1, \ldots, D_n$;
—constraints among variables;
—an *objective function* f to be minimized,[1] where $f : D_1 \times \cdots \times D_n \to \mathbb{R}^+$;

The set of all possible feasible assignments is

$$\mathcal{S} = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i, s \text{ satisfies all the constraints}\}.$$

$\mathcal{S}$ is usually called a *search (or solution) space*, as each element of the set can be seen as a candidate solution. To solve a combinatorial optimization problem one has to find a solution $s^* \in \mathcal{S}$ with minimum objective function value, that is, $f(s^*) \leq f(s)\ \forall s \in \mathcal{S}$. s^* is called a globally optimal solution of $(\mathcal{S}, f)$ and the set $\mathcal{S}^* \subseteq \mathcal{S}$ is called the set of globally optimal solutions.

Searching for a solution

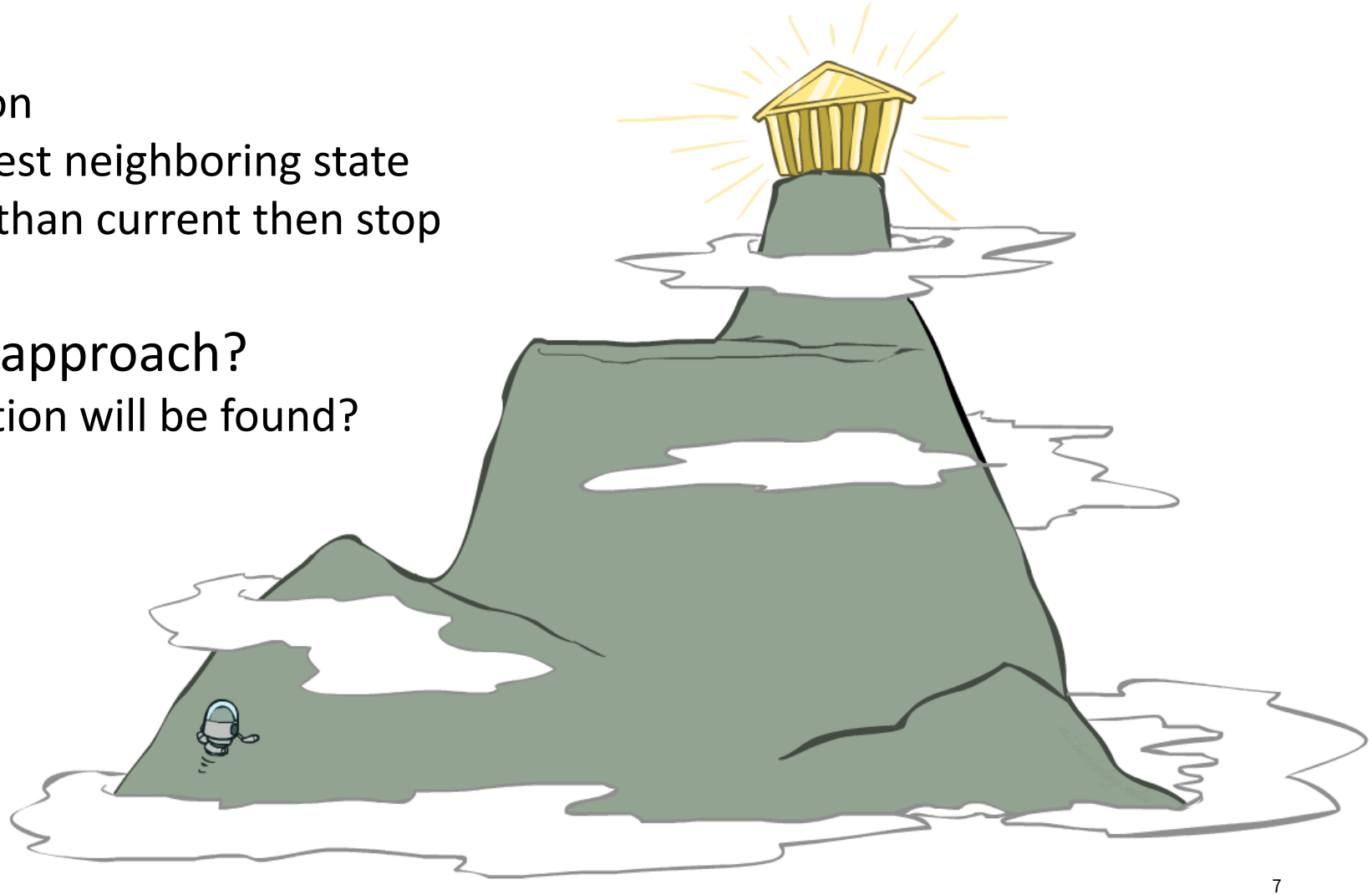
- We can consider a heuristic algorithm that guides the search process (we can’t evaluate every possible solution because there are too many, so which ones should we look at to try to find one we can use?)

- The goal is to efficiently explore the search space to find `near` optimal solutions

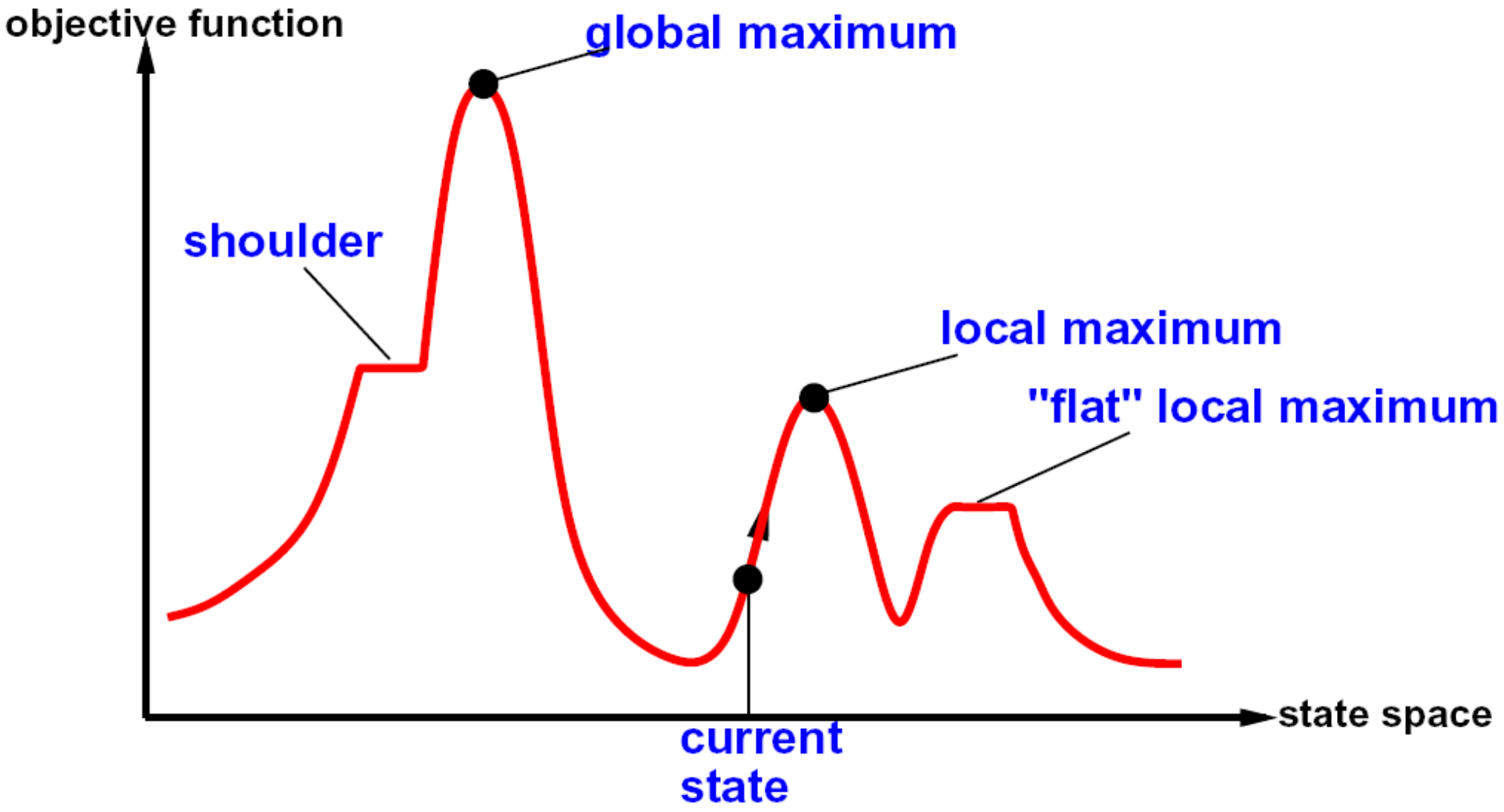
- Search algorithms can be simple or complex (e.g. hill climber, simulated annealing, genetic algorithm)

Hill Climbing

- Basic idea:
  - Start at random location
  - Repeat: move to the best neighboring state
  - If no neighbors better than current then stop
- What's bad about this approach?
  - Does it promise a solution will be found?
- What's good about it?
  - Non linear
  - Non differentiable
  - Non convex
  - Large and complex



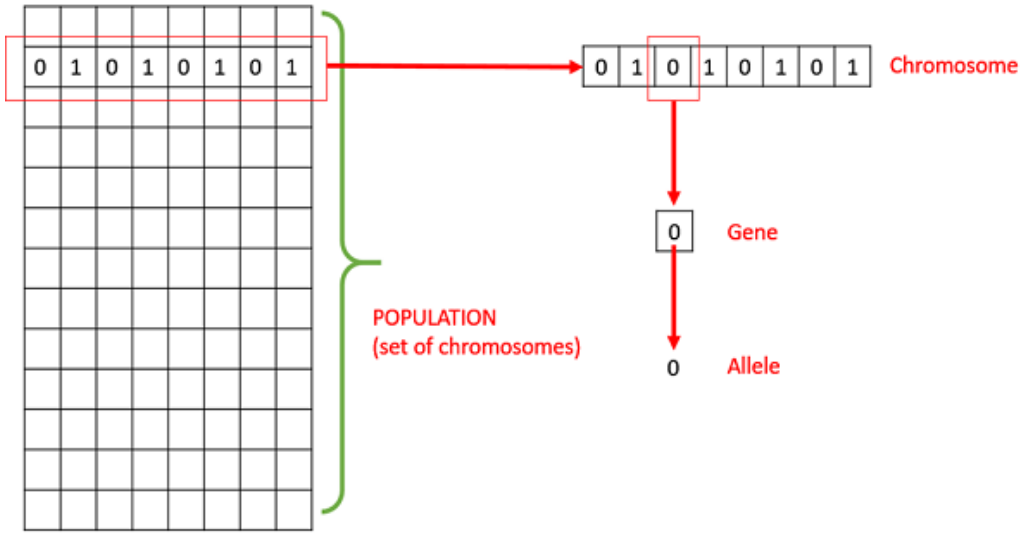
Searching


objective function
global maximum
shoulder
local maximum
"flat" local maximum
state space
current
state

Metaheuristics

- Approximate (don’t promise to find a solution or the absolute best one)
- Have mechanisms to avoid being trapped in local optima
- Use basic concepts to encourage abstract description
- Not problem specific
- Can utilise domain knowledge
  - EG heuristics that are controlled by a higher level strategy or handle constraints and certain cases with special “business rules”
- Some advanced heuristics can use past search experience to guide further searches in subsequent problems

Evolutionary Computation

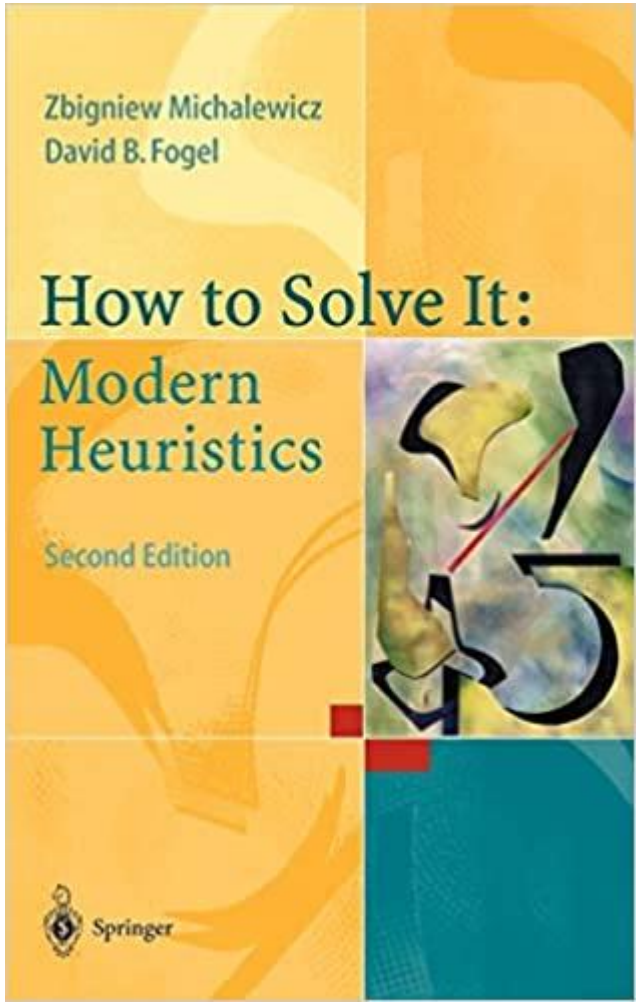


$P \leftarrow$ GenerateInitialPopulation()
Evaluate(P)
while termination conditions not met **do**
  $P' \leftarrow$ Recombine(P)
  $P'' \leftarrow$ Mutate(P')
  Evaluate(P'')
  $P \leftarrow$ Select($P'' \cup P$)
endwhile

Fig. 13. Algorithm: Evolutionary Computation (EC).

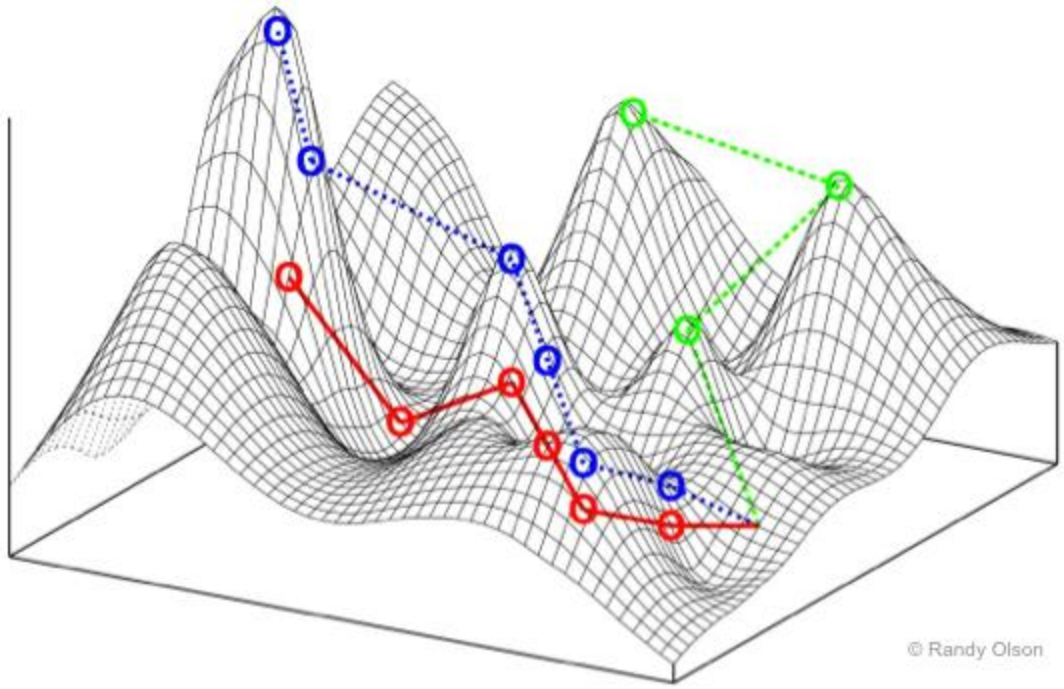
Some background materials (Further Reading)

- How to Solve IT: Modern Heuristics

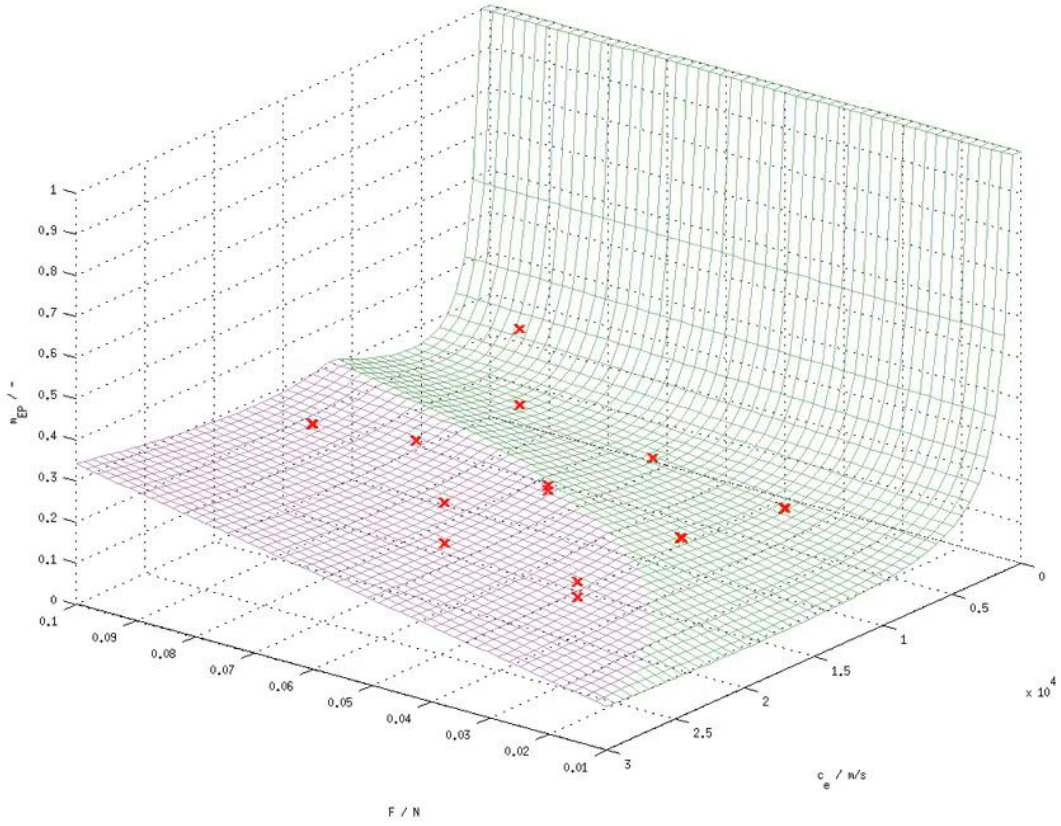


- Quick introduction (for example): https://blog.otoro.net/2017/10/29/visual-evolution-strategies/

Fitness Landscape concept



Demonstration (animation shows a minimization problem)



Individual Representation

- Binary

0	0	1	0	1	1	1	0	0	1

- Real

0.5	0.2	0.6	0.8	0.7	0.4	0.3	0.2	0.1	0.9

- Integer

1	2	3	4	3	2	4	1	2	1

- Permutation (order)

1	5	9	8	7	4	2	3	6	0

Basic Idea

- Apply variation **operators** (e.g. swap 2 elements) to find better solutions

1	5	9	8	7	4	2	3	6	0

- What is better? Depends on the objective function
- EG in magic square we can measure the total difference between the sum of each row, col, and diagonal and the mag

magic sum $S = 0.5 \times n \times (n^2 - 1)$

Evolutionary Strategies

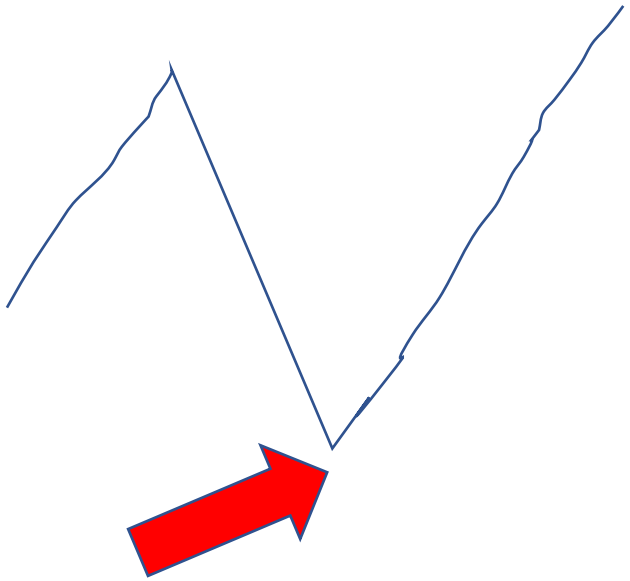
- Developed: Germany in the 1970's
- Early names: I. Rechenberg, H.-P. Schwefel
- Often applied to numerical optimisation
- Attributed features:
  - fast
  - good optimizer for real-valued optimisation
  - relatively much theory
- Special:
  - self-adaptation of (mutation) operator parameters

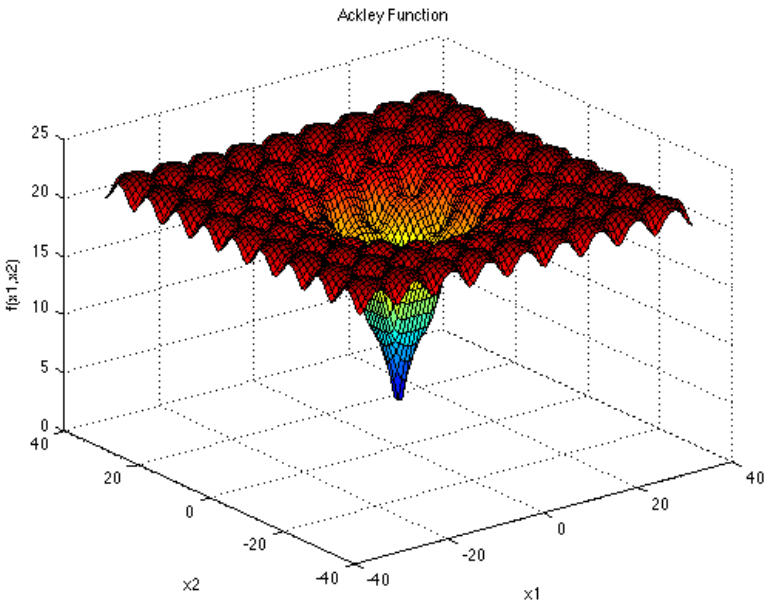
Evolutionary Strategies

Representation	**Repair**
Recombination	Discrete or intermediary
Mutation	Gaussian perturbation
Parent selection	Uniform random
Survivor selection	(μ,λ) or $(\mu+\lambda)$
Specialty	Self-adaptation of mutation step sizes

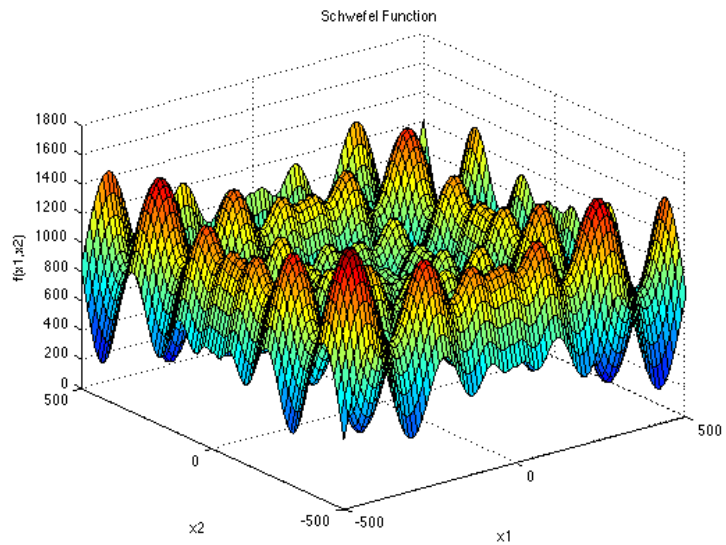
Example

- Task: minimise $f : R^n \rightarrow R$
- Original algorithm: “two-membered ES” using
  - Vectors from R^n directly as chromosomes
  - Population size 1
  - Only mutation creating one child
  - Greedy selection

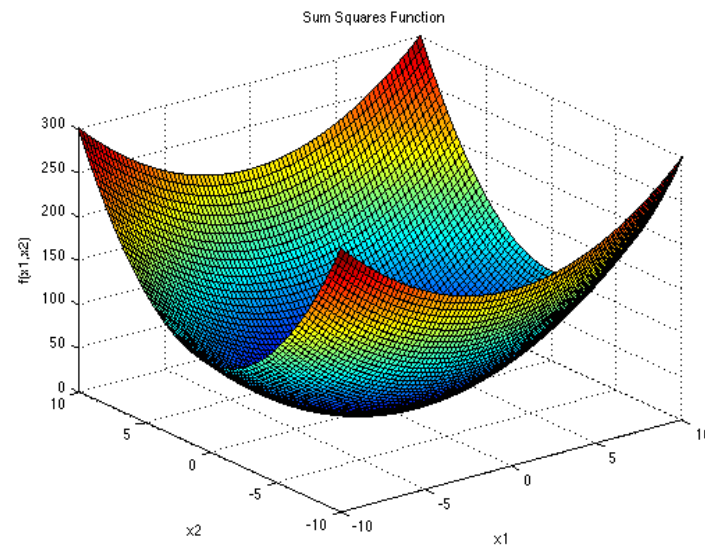




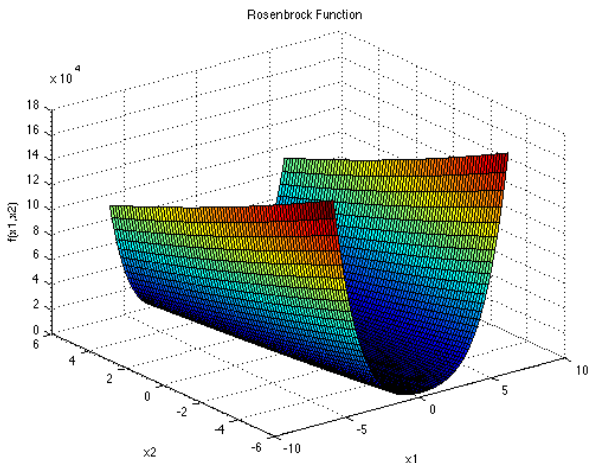
$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$



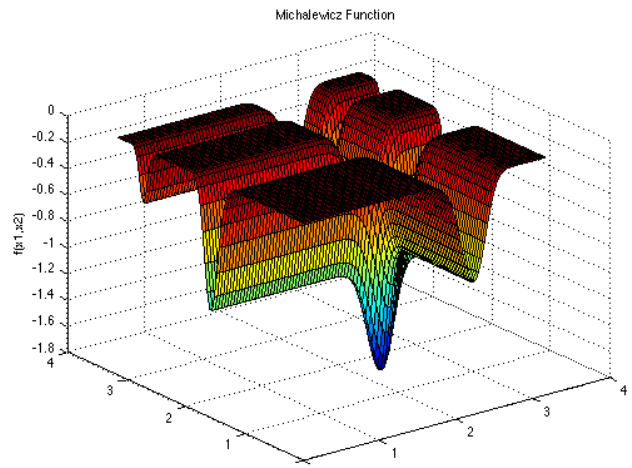
$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$



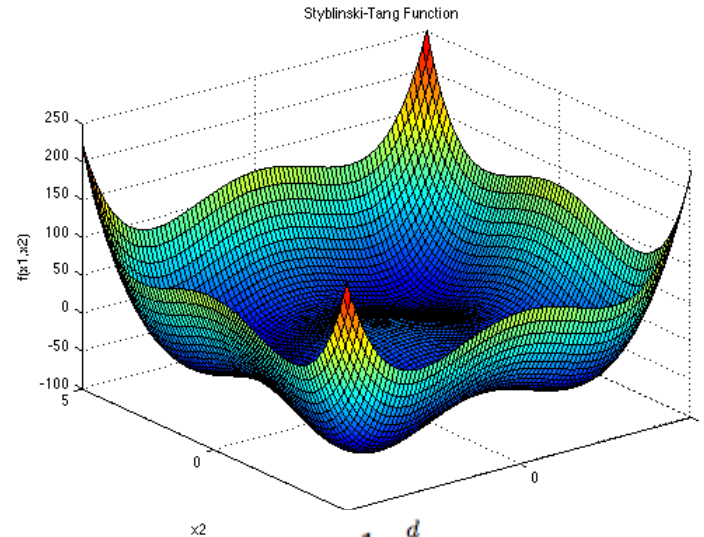
$$f(\mathbf{x}) = \sum_{i=1}^{d} i x_i^2$$



$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$



$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i)\sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$



$$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{d}(x_i^4 - 16x_i^2 + 5x_i)$$

Example – algorithm

1. Set t = 0
2. Create initial point at random: $\mathbf{x}^t = \langle \mathbf{x}_1^t,\ldots,\mathbf{x}_n^t \rangle$
3. REPEAT UNTIL (*TERMIN.COND* satisfied) DO
4. Draw $\mathbf{z}_i$ from a normal distribution for all i = 1,…,n
5. $\mathbf{y}_i^t = \mathbf{x}_i^t + \mathbf{z}_i$
6. IF fitness($\mathbf{x}^t$) < fitness($\mathbf{y}^t$) THEN $\mathbf{x}^{t+1} = \mathbf{x}^t$ ELSE $\mathbf{x}^{t+1} = \mathbf{y}^t$
7. Set t = t+1
8. OD

The operator to get a new solution is to add z to the previous solution

Solution for the next generation x^{t+1} is the "best found so far"

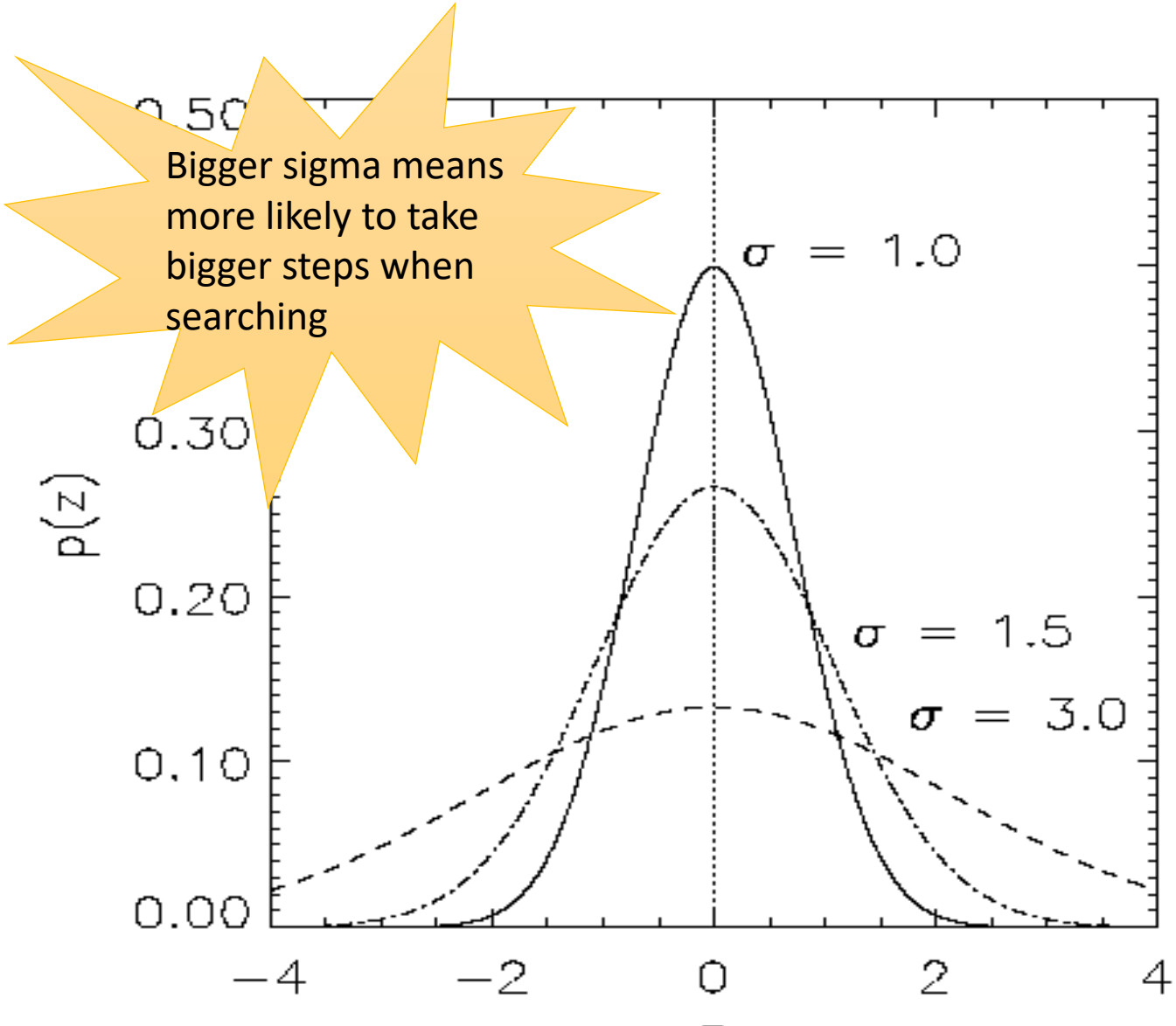
Operator: "Mutation"

- z values drawn from normal distribution $N(\xi,\sigma)$
  - mean ξ is set to 0
  - variation σ is called mutation step size
- σ is varied on the fly by the "1/5 success rule":
- This rule resets σ after every k iterations by
  - $\sigma = \sigma / c$ if $p_s > 1/5$
  - $\sigma = \sigma \bullet c$ if $p_s < 1/5$
  - $\sigma = \sigma$ if $p_s = 1/5$

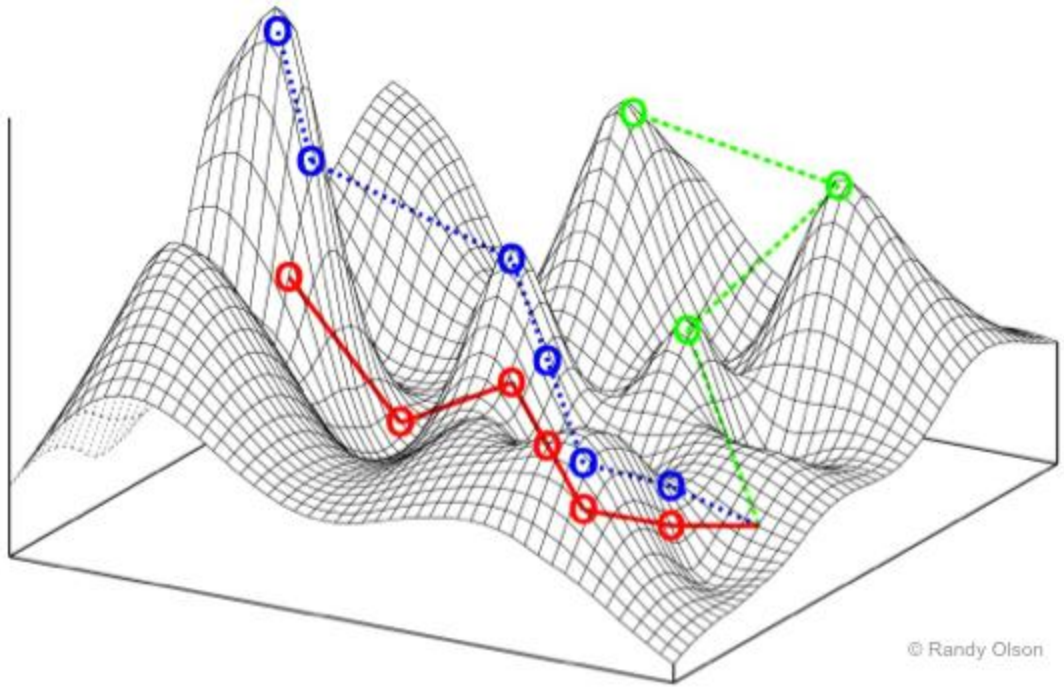
  where p_s is the % of successful mutations, $0.8 \leq c \leq 1$

Reduce the step size if finding good solutions near the current solution, increase to explore different places if p_s low

Illustration of Normal Distribution



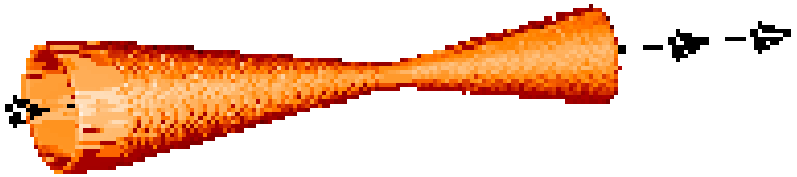
Small changes VS Bit changes



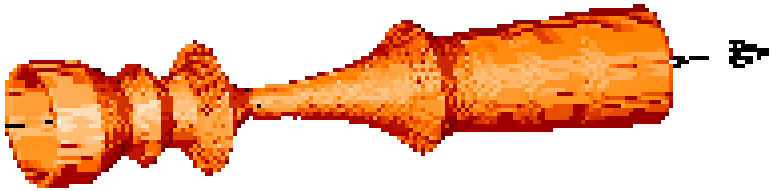
Another Example: Jet Nozzle Design

Task: to optimize the shape of a jet nozzle to produce maximum propulsion

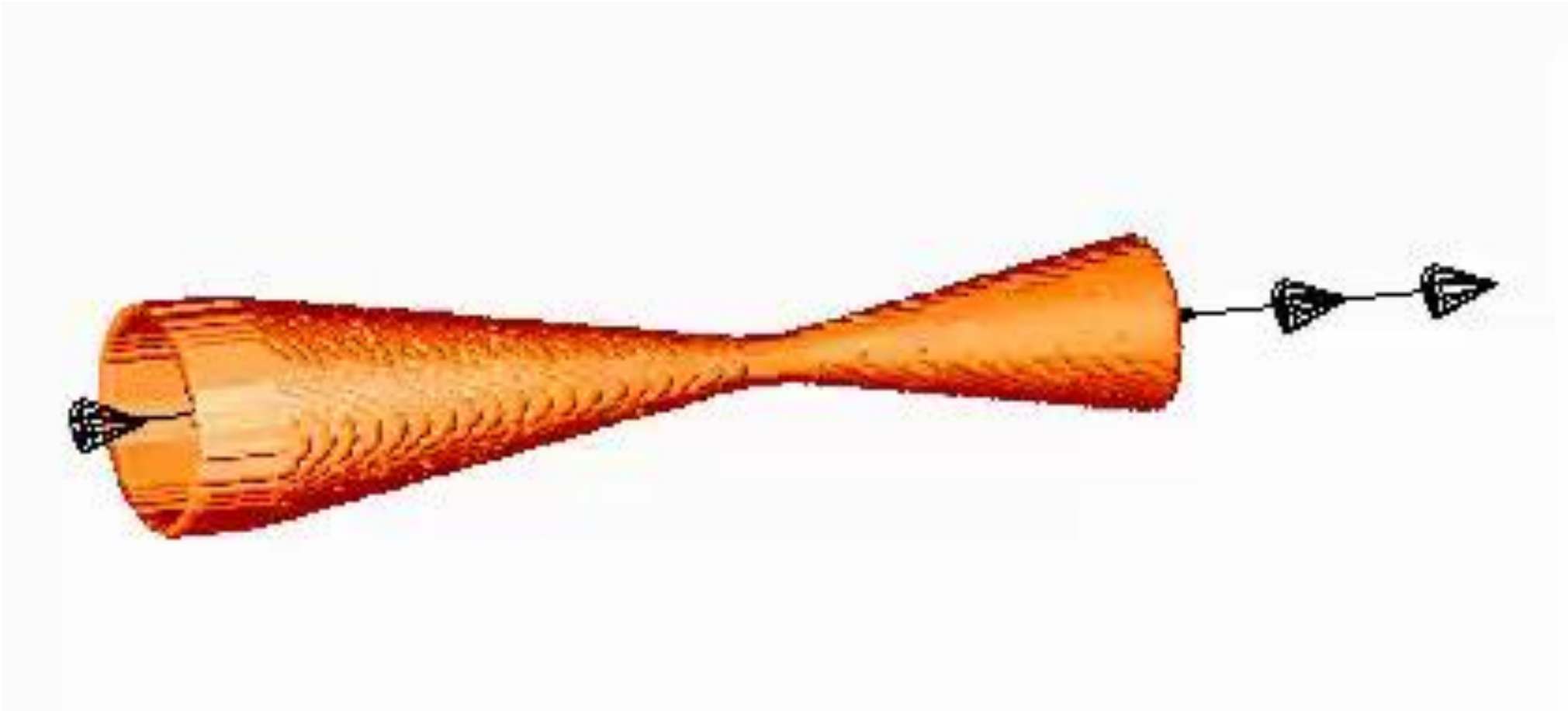
Approach: random mutations to shape + selection



Initial shape



Final shape



Advanced Representation

- Chromosomes consist of two parts:
  - Object variables: $x_1,\ldots,x_n$
  - Mutation step sizes: $\sigma_1,\ldots,\sigma_{n_\sigma}$
- Chromosome: $\langle x_1,\ldots,x_n, \sigma_1,\ldots,\sigma_n \rangle$

Each "gene" has it's own variation step size

x	0.5	0.2	0.6	0.8	0.7	0.4	0.3	0.2	0.1	0.9
σ	.1	.01	.2	.6	.01	.5	.5	.5	.7	.02

Mutation

- Main mechanism: changing value by adding random noise drawn from normal distribution
- $x'_i = x_i + N(0,\sigma)$
- Key idea:
  - σ is part of the chromosome $\langle x_1,\ldots,x_n, \sigma \rangle$
  - σ is also mutated into σ'
  - Thus: mutation step size σ is also evolving with the solution x

Mutate σ first

- Net mutation effect: $\langle x, \sigma \rangle \rightarrow \langle x', \sigma' \rangle$
- Order is important:
  - first $\sigma \rightarrow \sigma'$
  - then $x \rightarrow x' = x + N(0,\sigma')$
- Rationale: new $\langle x', \sigma' \rangle$ is evaluated twice
  - Primary: x' is good if f(x') is good
  - **Secondary: σ' is good if the x' it created is good**
- Reversing mutation order this would not work because we want to see how much the *new* step size influences the search

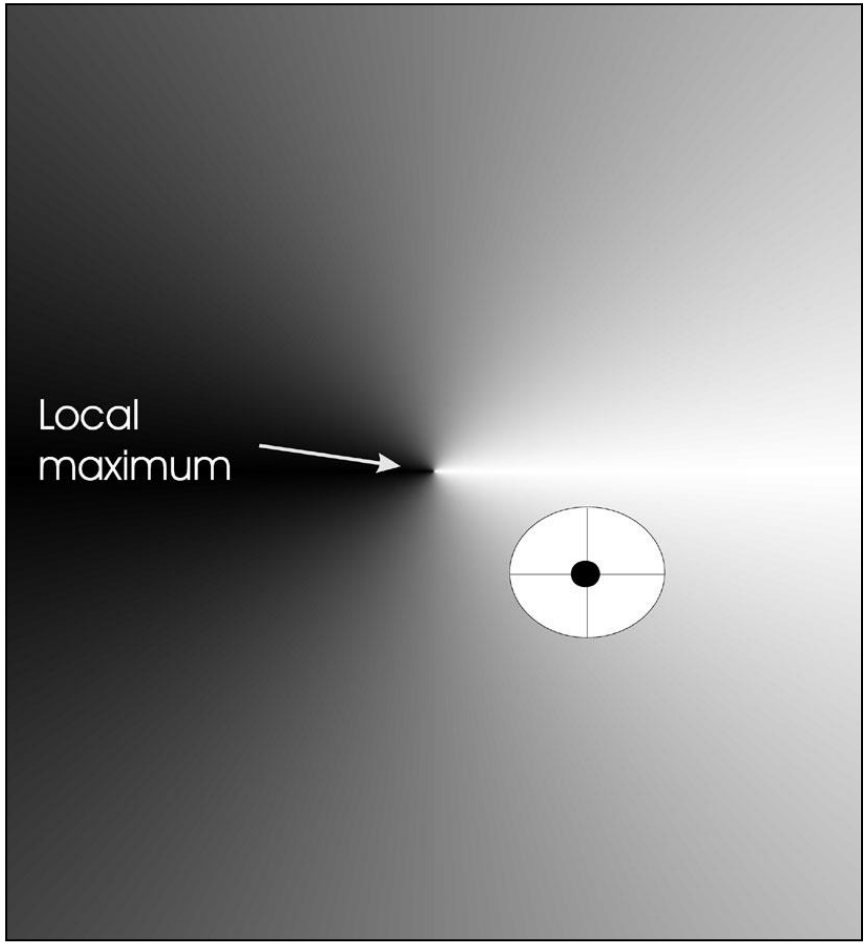
Mutation case 1: Uncorrelated mutation with one σ

Chromosomes: $\langle x_1,\ldots,x_n, \sigma \rangle$

- $\sigma' = \sigma \bullet \exp(\tau \bullet N(0,1))$
- $x'_i = x_i + \sigma' \bullet N(0,1)$

τ Is like a step size for the mutation rate change

Mutants with equal likelihood



Circle: mutants having the same chance to be created

Mutation case 2: Uncorrelated mutation with n σ's (one for each chromosome "gene")

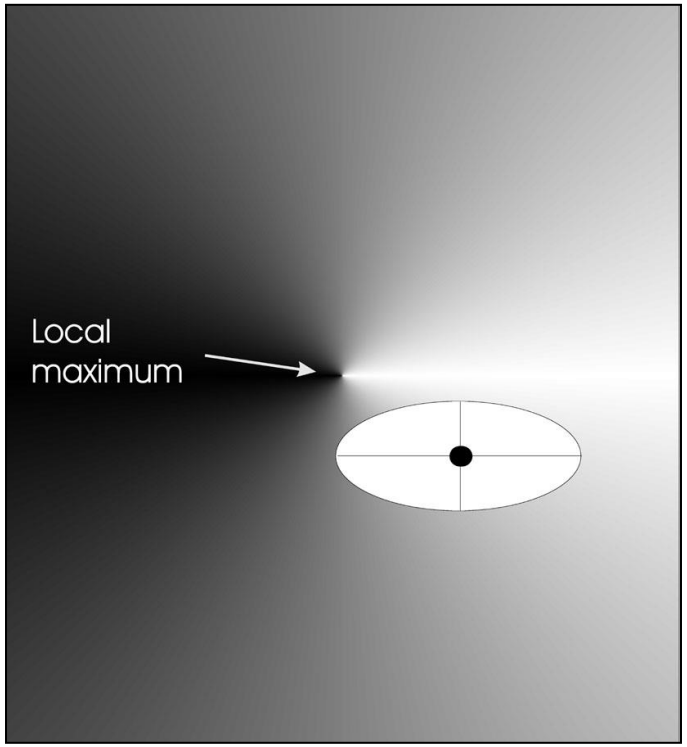
Chromosomes: $\langle x_1,\ldots,x_n, \sigma_1,\ldots, \sigma_n \rangle$

- $\sigma'_i = \sigma_i \bullet \exp(\tau' \bullet N(0,1) + \tau \bullet N_i(0,1))$
- $x'_i = x_i + \sigma'_i \bullet N_i(0,1)$
- two learning rate parameters:
  - τ' overall learning rate
  - τ per "gene" wise learning rate

and $\sigma_i' < \varepsilon_0 \Rightarrow \sigma_i' = \varepsilon_0$

Mutants with equal likelihood

- More change in only some variables but not others



Ellipse: mutants having the same chance to be created

More advanced representation

- Chromosomes consist of three parts:
  - Object variables: $x_1,\ldots,x_n$
  - Strategy parameters:
    - Mutation step sizes: $\sigma_1,\ldots,\sigma_{n_\sigma}$
    - Rotation angles: $\alpha_1,\ldots, \alpha_{n_\alpha}$
- Not every component is always present
- Full size: $\langle x_1,\ldots,x_n, \sigma_1,\ldots,\sigma_n ,\alpha_1,\ldots, \alpha_k \rangle$
- where $k = n(n-1)/2$ (no. of i,j pairs)

α - another set of parameters to control which variables are changed

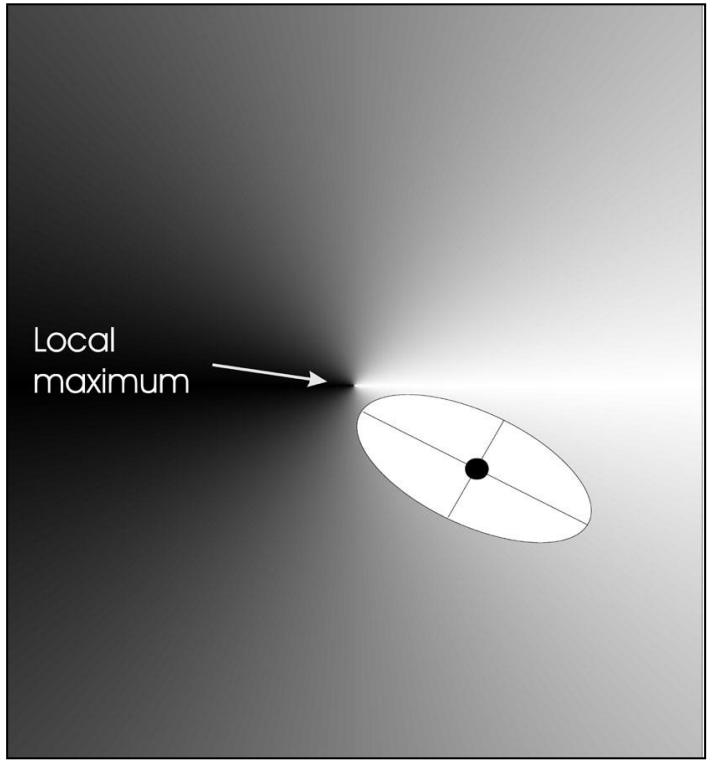
Mutation case 3:
Correlated mutations

Chromosomes: $\langle x_1,\ldots,x_n, \sigma_1,\ldots, \sigma_n, \alpha_1,\ldots, \alpha_k \rangle$

- where $k = n \bullet (n-1)$
- and the covariance matrix C is defined as:
  - $c_{ii} = \sigma_i^2$
  - $c_{ij} = 0$ if i and j are not correlated
  - $c_{ij} = ½ \bullet (\sigma_i^2 - \sigma_j^2) \bullet \tan(2\alpha_{ij})$ if i and j are **correlated**

α_m : correlation between step sizes (1 for each pair)

Mutants with equal likelihood

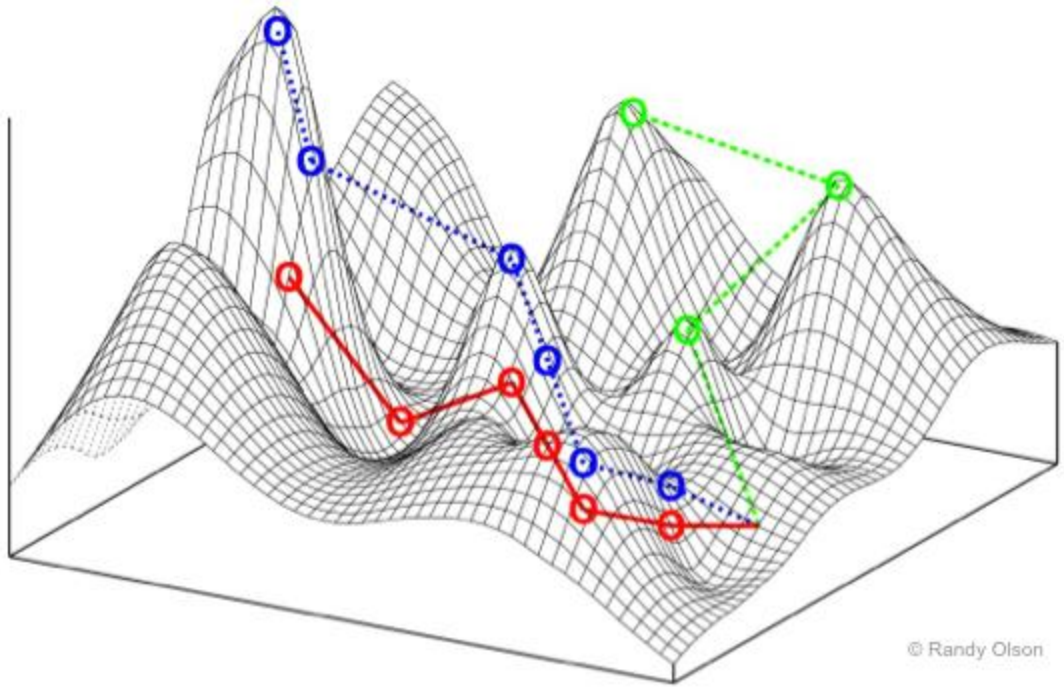


Ellipse: mutants having the same chance to be created

Recombination

- Creates one child
- Acts per variable / position by either
  - Averaging parental values, or
  - Selecting one of the parental values
- From two or more parents by either:
  - Using two selected parents to make a child
  - Selecting two parents for each position anew

Crossover (combining optima)



Recombination Types

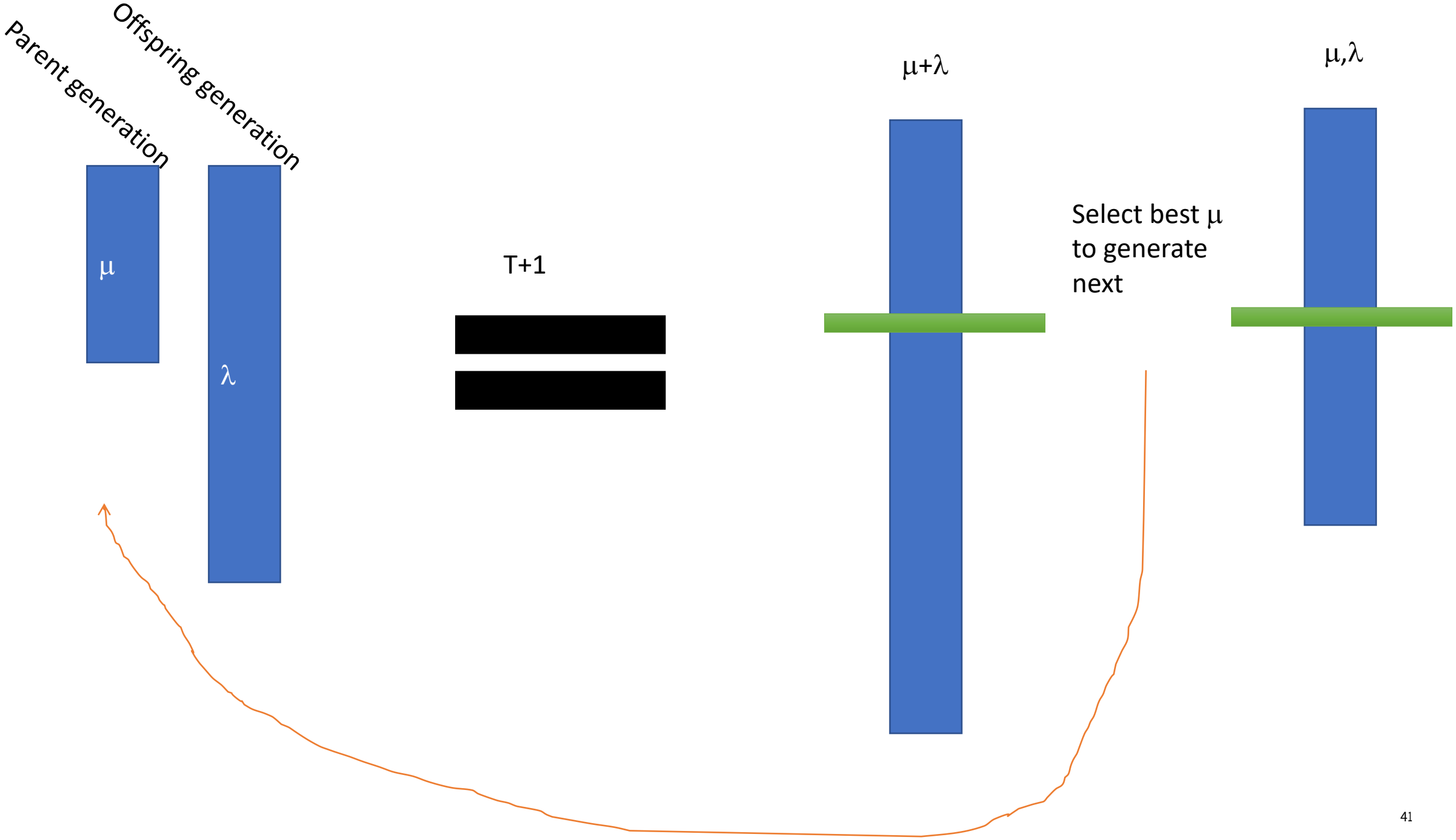
	Two fixed parents	Two parents selected for each i
$z_i = (x_i + y_i)/2$	Local intermediary	Global intermediary
z_i is x_i or y_i chosen randomly	Local discrete	Global discrete

Choosing carents for crossover

- Parents are selected by uniform random distribution whenever an operator needs one/some
- Thus: ES parent selection is unbiased - every individual has the same probability to be selected (can add bias to prefer better individuals in some schemes but still has to be random)
- Note that in ES “parent” means a population member

Survivor selection

- Applied after creating λ children from the μ parents by mutation and recombination
- Deterministically chops off the “bad stuff”
- Basis of selection is either:
  - The set of children only: (μ,λ)-selection
  - The set of parents and children: $(\mu+\lambda)$-selection


Parent generation
Offspring generation
μ
λ
T+1
μ+λ
Select best μ
to generate
next
μ,λ

Survivor selection cont'd

- $(\mu+\lambda)$-selection is called an **elitist** strategy because the better solutions are kept across generations
- (μ,λ)-selection can "forget"
- Often (μ,λ)-selection is preferred for:
  - Better in leaving local optima
  - Better in following moving optima
  - Using the + strategy bad σ values can survive in $\langle x,\sigma\rangle$ too long if their host x is very fit
- Selective pressure in ES is very high ($\lambda \approx 7 \bullet \mu$ is the common setting)

Local Search Algorithm – Key Points

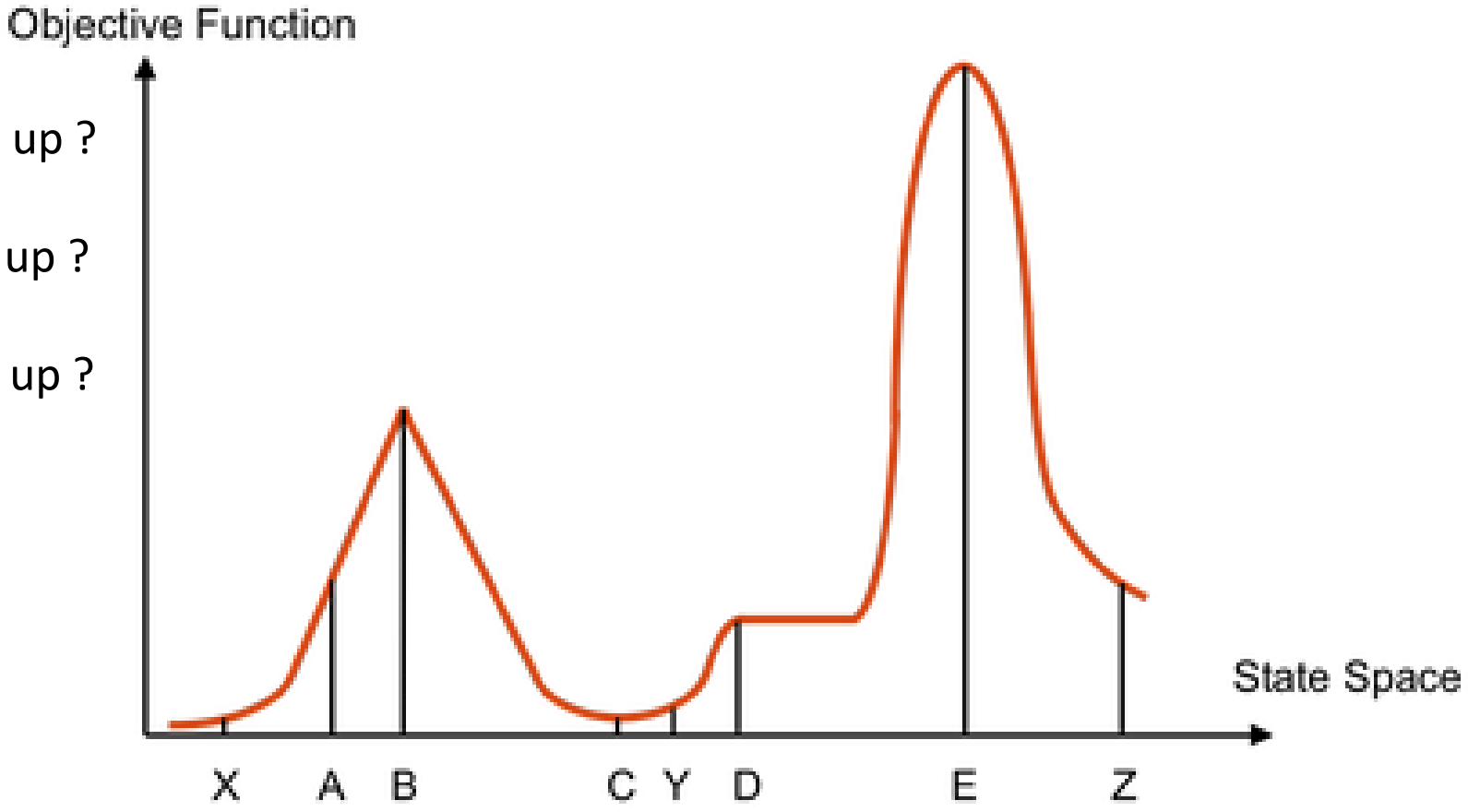
- Keep a current single state or a set of states “candidate solutions”
- Iteratively improve by selecting new “solutions” stochastically
- Very efficient in terms of memory

Hill Climbing Quiz

Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?



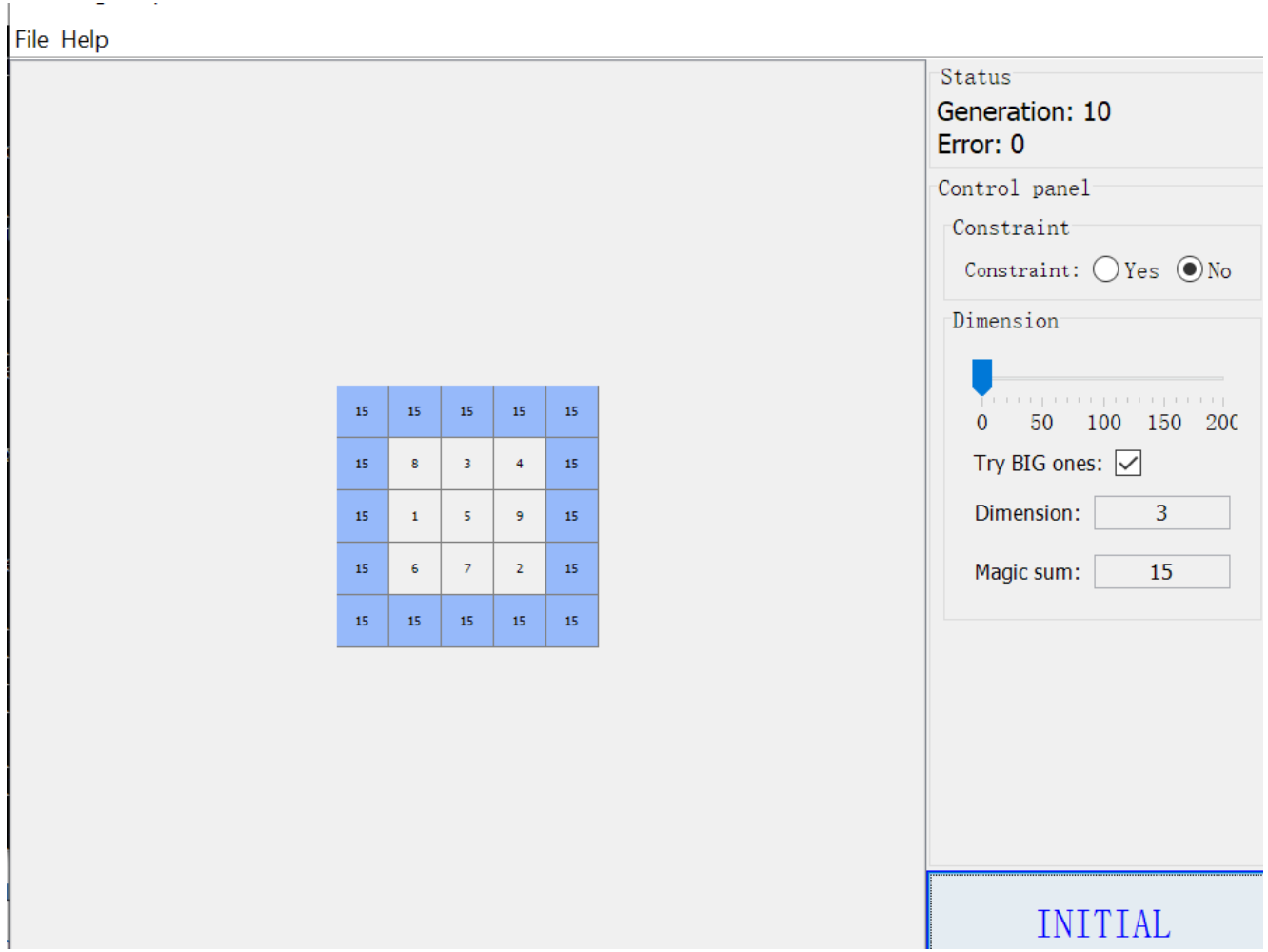
Stochastic Local Search concepts

- Greedy descent: move to best neighbour
- Random walk: taking random steps
- Random restart: reassigning all values of variables

- Stochastic local search = greedy descent + random walk + random restart

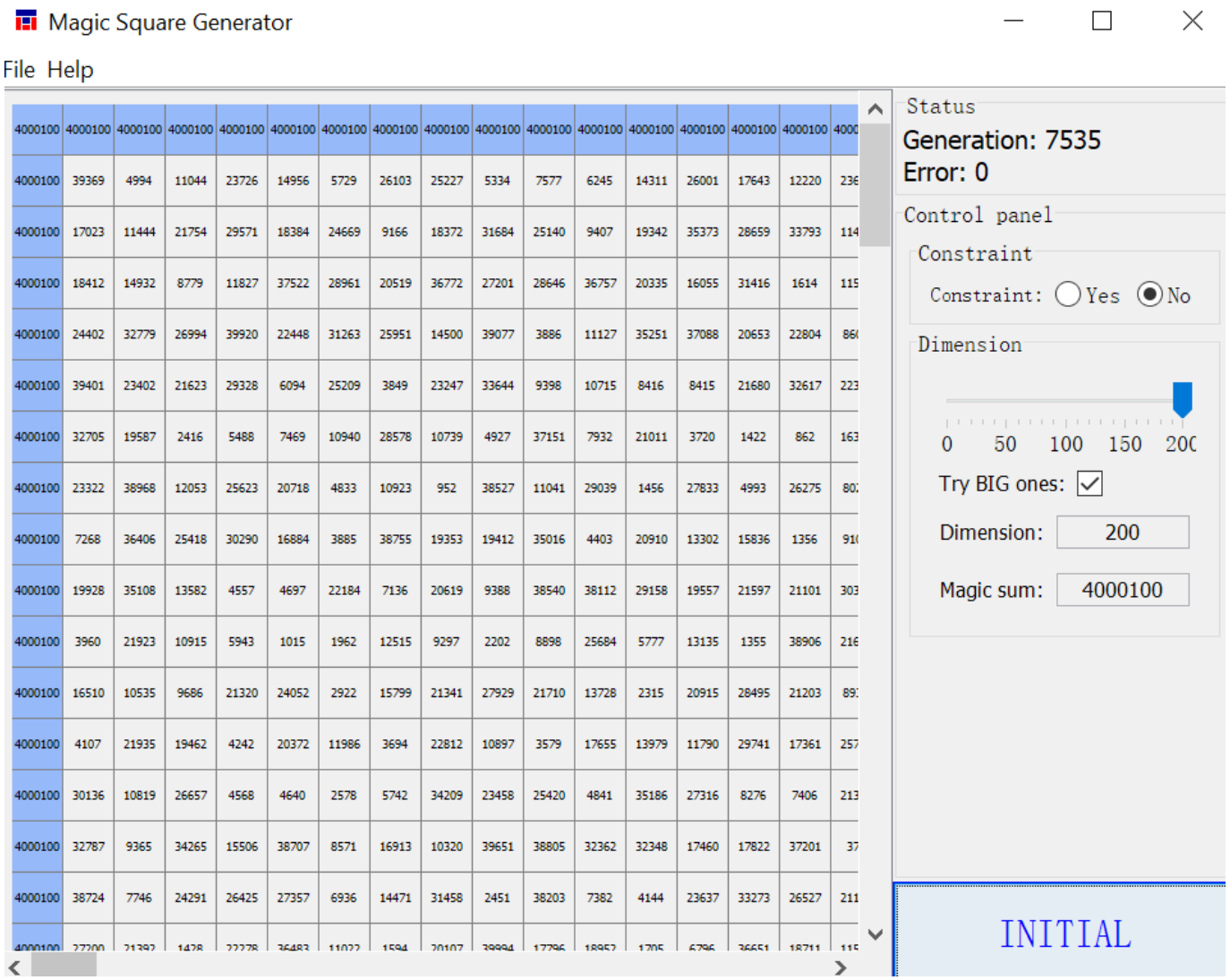
Magic Square

magic sum $S = 0.5 \times n \times (n^2 - 1)$



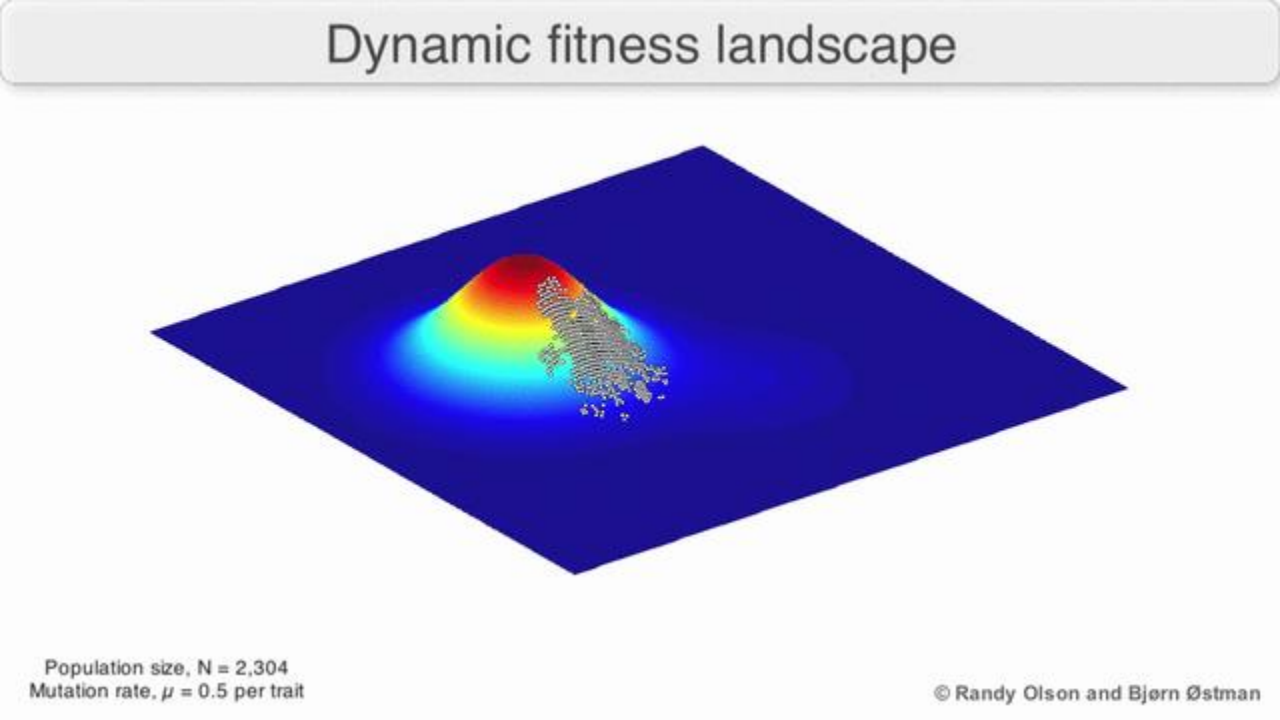
Magic Square

magic sum $S = 0.5 \times n \times (n^2 - 1)$



Self-adaptation illustrated

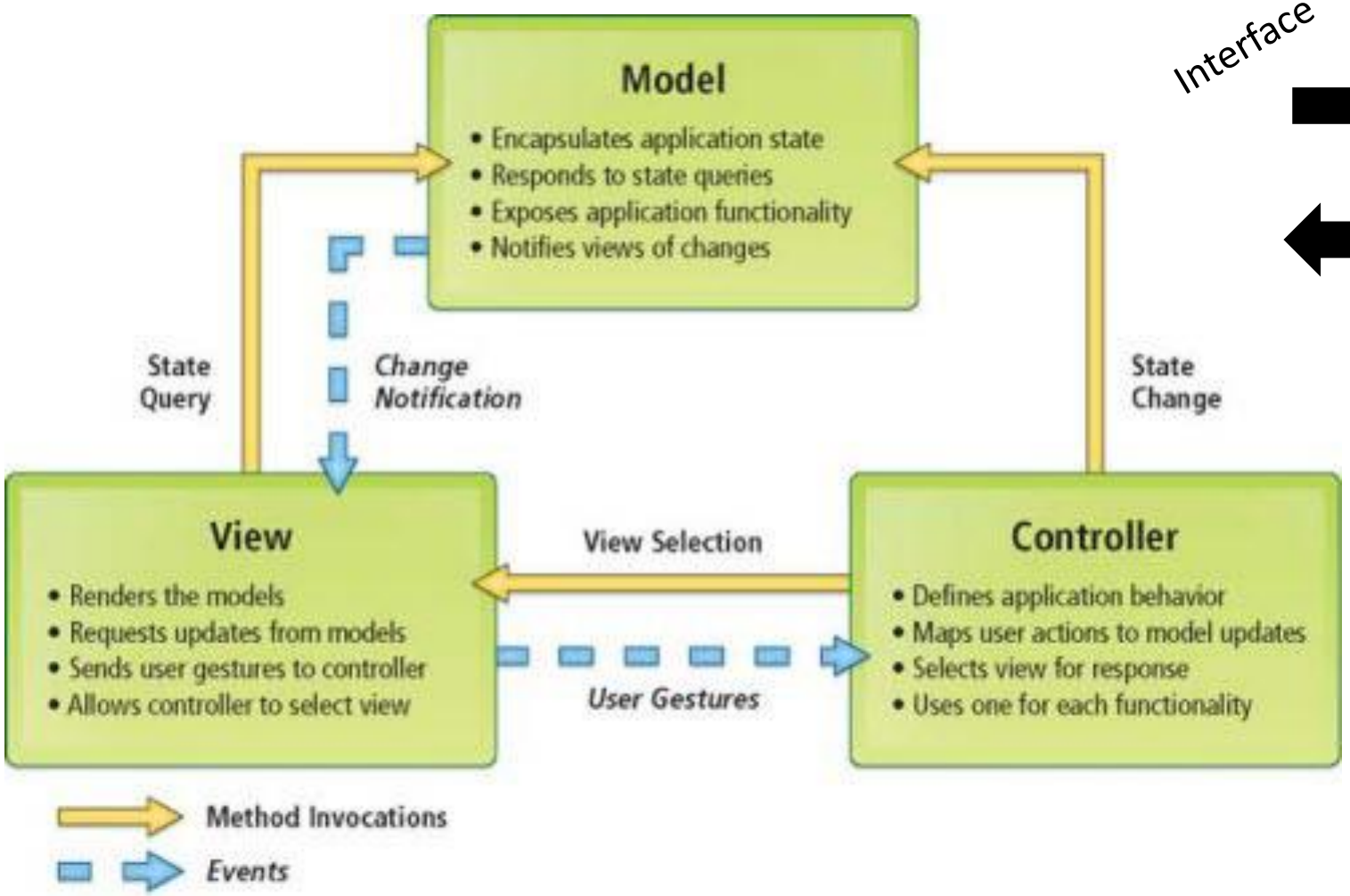
- Given a dynamically changing fitness landscape (optimum location shifted every 200 generations)
- Self-adaptive ES is able to
  - follow the optimum and
  - adjust the mutation step size after every shift !



General Comments on Project Architecture

- Key Design decisions
  - Solver representation needs to be fast because the when the solver runs many iterations are needed to produce new solutions and then evaluate them
  - The requirements for the representation of the model used in the user interface needs to be able to model the problem and allow different puzzles to be represented
  - There is a tradeoff between these, the solver can be highly specific, the main model has to capture all aspects in a well structured way that allows for extending
  - Different model for optimizer/solver and general application?
- How to organize the team to have distinct modules where members can work? How to coordinate the interface between the modules?

Main application modules



Interface

Solver modules

Specialized solver 1

- Model (specific set of constraints and fast data structures)
- Algortihm

Specialized solver 2

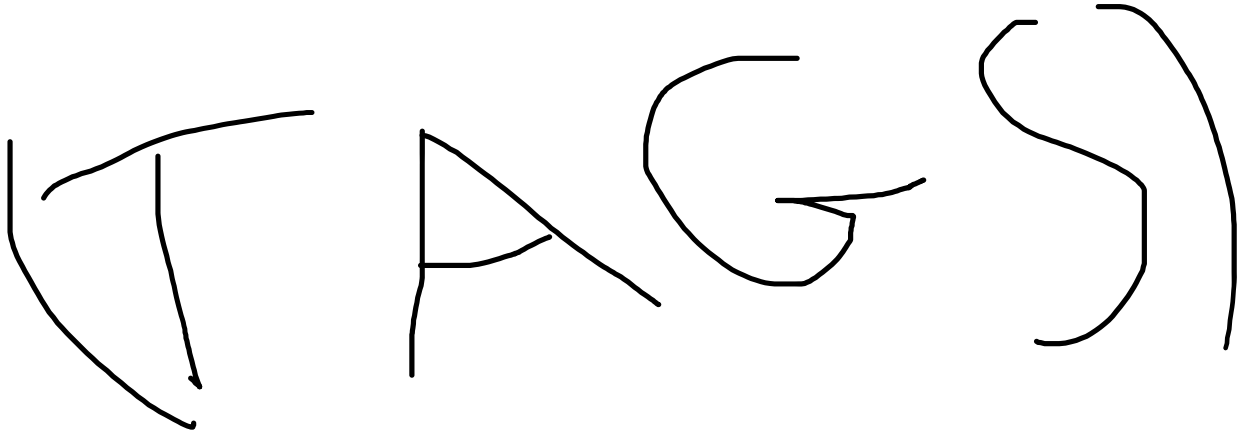
- Model (specific set of constraints and fast data structures)
- Algortihm

Java Code Annotations

Annotations

- You may have noticed some annotations already; it's common when you use inheritance and extend an abstract class. Here you redefine in the child class a method defined in the parent class to precede the child class definition with @Override (which means *replace* the existing method).
- This is an annotation, which is completely optional but warns javac of your intent. If you mistype the name, it will enable javac to detect an error if there is no such method in the parent class.

Annotations



- Completely optional
  Change nothing to what the program does
- Help Javac – or the program
- Much used by code-generating tools

As annotations can be accessed by programs, many tools that generate code – for tests, for instance – use annotations to collect information they cannot get otherwise.

Annotations

Annotations can take different forms, from the simple marker to some kinds of function calls that are outside the program itself.

- Marker e.g. @Override
- Single parameter
- Multiple parameters

```
@Select (sql ="SELECT userId, firstName,...
```

Metadata

= DATA about the CODE

Metadata is a big concern in real life. Companies consider programs as assets, on which several generations of developers can work, which must be written in an easy-to-comprehend, standard way, and well documented. Metadata allows, among many other things, to industrialize code production and to standardize everything.

Standard Annotations

- @Override
- @Deprecated = old / obsolete
- @SuppressWarnings

Java provides three standard annotations, which are all a way to give hints to javac.

e.g. @SuppressWarnings({"deprecation", "unchecked"})

2 Annotations were added in Java 7

- @SafeVarargs
- @FunctionalInterface

"Varargs" stands for "Variable [number of] Arguments"

FunctionalInterface marks a functional interface as we saw can be assigned to anonymous functions

Making Annotations

- You can create your own annotations!

Declared as interfaces

```
import java.lang.annotation.*;

public @interface MyAnnotation {
 }
```

Annotation-based tools use their own set of annotations, which you just need to import before using. EG JUNIT.

Making Annotations

- As annotations are a bit special (it's a kind of program in the program) they are constrained by a number of rules.

- **Methods** should not have any parameters.
- **Methods declarations** should not have any throws clauses.
- ***Return*** *type must be one of:*
  - primitive type, String, enum, Class or an array of one of these

https://docs.oracle.com/javase/tutorial/java/annotations/

Custom Annotations Example: Structured Documentation

```
class SomeClass {
// Created by S Faroult // Creation date: 21/03/2017
// Revision history:
//
// 24/05/2017–Constructor with String parameter
// 26/02/2018 – toString() rewritten
```

What can you use annotations for in practice? Any Software Development Manager dreams of seeing comments like this. But every developer will not write them, and those who do may use a different format.

Custom Annotations Example: Structured Documentation

```
import java.lang.annotation.*;

public @interface ClassDoc {
        String author();
        String created();
        String[] revisions();
}
```

ClassDoc.java

Annotations could help turning readable but unparsable comments into usable data…

Custom Annotations Example: Structured Documentation

```
@ClassDoc (		author="S Faroult",
			created="21/03/2017",
			revisions={"24/05/2017 – Constructor
		with String parameter",
			"26/02/2018 – toString() rewritten"})
class SomeClass {
```

As the annotation is defined and checked by javac it becomes possible to ensure a standard way for documenting the code. This information can then be retrieved to document programs (such as with Javadoc – will cover later).

Meta Annotations (Annotations about Annotations)

- **@Retentions:** says whether the annotation is available to javac or available at runtime.
- **@Documented**: Make it appear in the docs generated by Javadoc tool.
- **@Target**: What it applies to: Constructor, Method, Parameter…
- **@Inherited**: Passed to child classes (false by default).
- **@Repeatable**: Can be applied multiple times.

Testing

JUnit

- JUNIT handles tests for checking your programs. Frameworks are software tools that try to generate automatically the boring bits of a program (which are often a lot of copy-and-paste).

https://junit.org/junit5/

@Test

@BeforeClass – Run once before any of the test methods in the class, public static void
@AfterClass – Run once after all the tests in the class have been run, public static void
@Before – Run before @Test, public void
@After – Run after @Test, public void
@Test – This is the test method to run, public void

Reflection

Reflection

- Generally speaking, "reflection" is your program asking the JVM what it knows about it – and the JVM knows a lot of things.
- As all this happens of course while the program is running, it allows for a lot of on-the-fly operations that would be impossible with a compiled program written in C, for instance.
- Reflection is considered rather advanced programming, but some of its features are frequently used, for instance with JDBC which is the standard Java way to access a database.

Reflection

- Works because of the JVM. Once again, it only works because of the JVM
- The JVM stores the description of the classes when it loads them
- The loading subsystem needs to read a lot of information to make the program runnable, and this information is stored and made available when the program runs

Reflection

- The JVM stores objects, of class **Class**, that describe every class used in the application

- Class called **Class**
  - *Metadata*
- The objects represent classes in the running application

- Class objects have no constructor – they are built by the JVM

Class objects

- There are two ways two retrieve class information from the JVM.

ClassName obj = new ClassName();

Method inherited from object

1. Method inherited from the object:

   The get class method of the object: obj.getClass()

2. The .class attribute when there is no object: ClassName.class

Static version

no constructor – built by the JVM

Uses of Reflection #1: Getting Class Names

```
class OuterClass {
        private int dummy;
        OuterClass(){}
}
public class MyClass {

        class InnerClass {
                private int dummy;
                InnerClass(){}
        }
public static void main(String[] args) {
        OuterClass obj = new OuterClass();
        System.out.println(obj.getClass().getName());
        System.out.println(InnerClass.class.getName());
}
```

For instance, you can retrieve class names.

```
$
$ java MyClass
OuterClass
MyClass$InnerClass
$
```

Uses of Reflection #2: Locating Files used by Your Program

- One common problem is locating files used by your program – the properties file to start with if there is one.
- Location of files read by your program
  - parameter file
  - data file
    multimedia, etc

When people click on an icon to launch your program, the idea of "current directory" becomes extremely hazy. If you want to start by reading a properties file, of if you want to display the logo of your company (an image) while initialization is going on, where should you look?

The default directory for installing programs varies from system to system (and don't forget that a Java application can run on Windows as well as on Linux or Mac OSX), and additionally users often have the option of installing software elsewhere than the default location. Your only hope to find out is to get it when the program runs.

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.

file:/Users/... ... .../Reflection.class

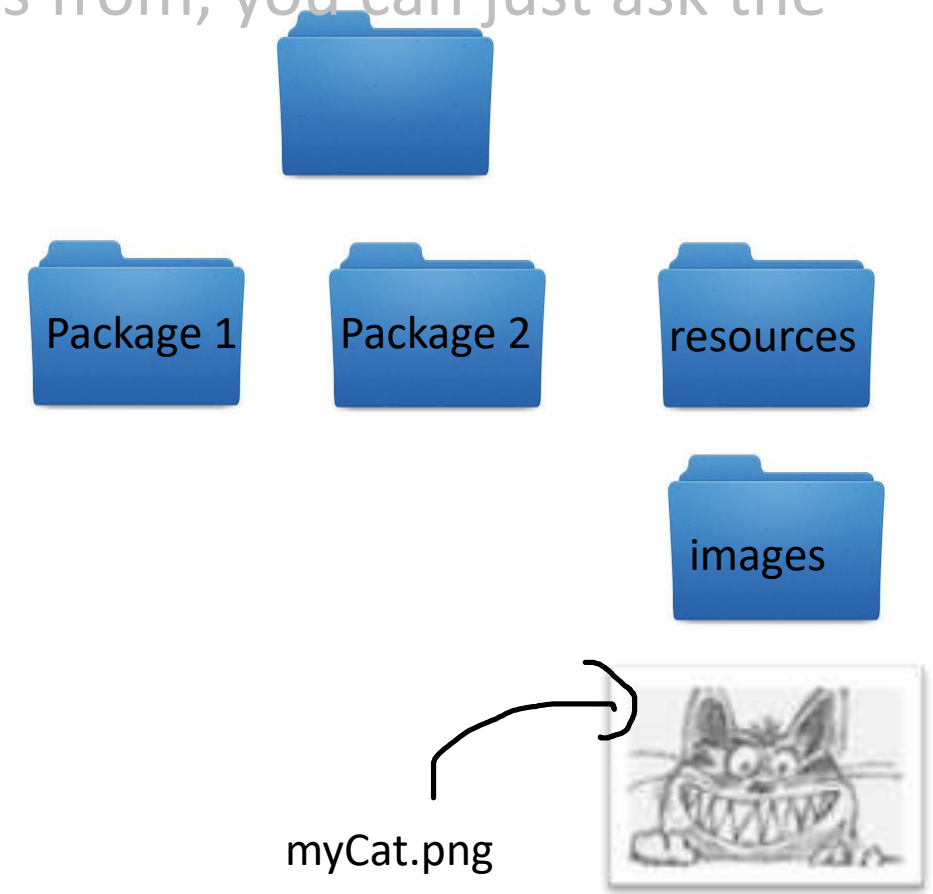
- **Solution** get location at runtime

```
public class Reflection {

        public static void main(String[] args) {
                System.out.println(Reflection
                                .class
                                .getClassLoader() .getResource("Reflection.class") .toSt

        }
}
```

Uses of Reflection #2: Locating Files used by Your Program

- As the loader knows where it got the .class from, you can just ask the JVM.
- **Solution** get location at runtime

```
URL url = this
            .getClass()
             .getClassLoader()
             .getResource("resources/images/myCat.png");
```



Uses of Reflection #3: Reading Annotations

- We saw in the previous presentation that annotations could be read by a program, it's through reflection.
- Done by many tools such as JUNIT and more we will see later.

Uses of Reflection #3: Reading Annotations

- We saw in the previous presentation that annotations could be read by a program, it's through reflection.
- Done by many tools such as JUNIT and more we will see later.

- By default annotations are **NOT** visible at runtime so we make them so as follows

@Retention(RetentionPolicy.RUNTIME) `

Uses of Reflection #3: Reading Annotations

- By default annotations are **NOT** visible at runtime so we make them so as follows

@Retention(RetentionPolicy.RUNTIME) `

Remember that @Retention() is a meta-annotation, an annotation that applies to annotations.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface ClassDoc {
        String author();
        String created();
        String[] revisions();
}
```

Uses of Reflection #3: Reading Annotations

- If SomeClass is annotated with an annotation available at runtime ...

```
@ClassDoc(
        author="S Faroult",
        created="21/03/2017",
        revisions={"24/05/2017 – Constructor with String parameter",
                "26/02/2018 – toString() rewritten"})
class SomeClass {
}
```

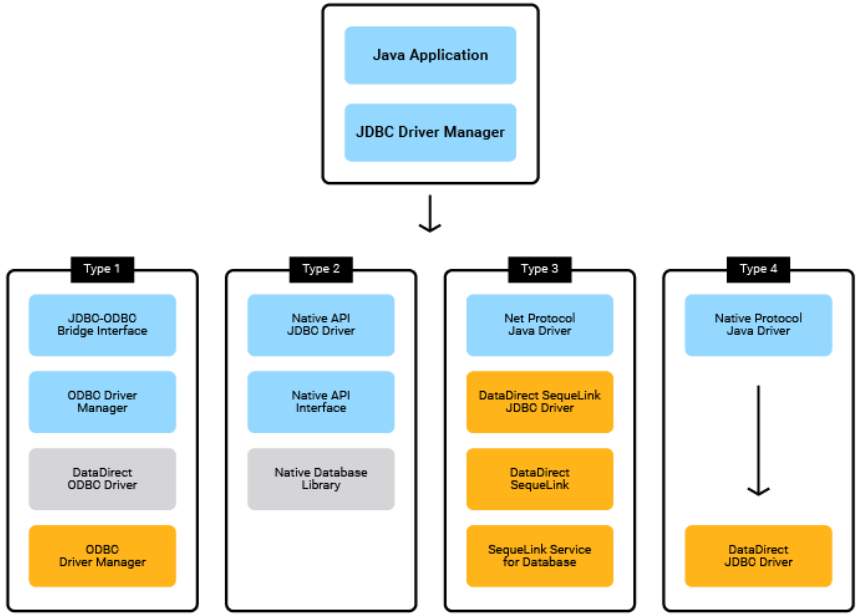
Note: it must be recompiled if ClassDoc is changed for getAnnotations to see it…

Uses of Reflection #3: Reading Annotations

- Then annotations gets it…

```
$ java ReadingAnnotations
@ClassDoc(author=S Faroult, created=21/03/2017
revisions=[24/05/2018 - Constructor with Strin
parameter, 26/02/2018 - toString() rewritten])
$
```

```
import java.lang.annotation.Annotation;
public class ReadingAnnotations {

        public static void main(String[] args) {
                Annotation[] annotations = SomeClass .class
                                                .getAnnotations();
                for (Annotation annot: annotations) {
                        System.out.println(annot.toString());
                }
        }
}
```

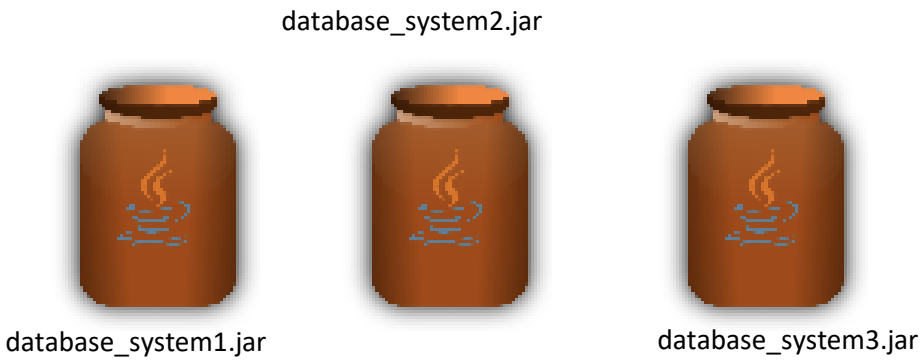


Uses of Reflection #4: Dynamically loading a class

- This is another very useful application of reflection
- Much used for "drivers" of hardware
- Because of the many different standards (National, International, Proprietary, etc) identical functionality is often achieved by different classes, that work with one special piece of hardware or software.

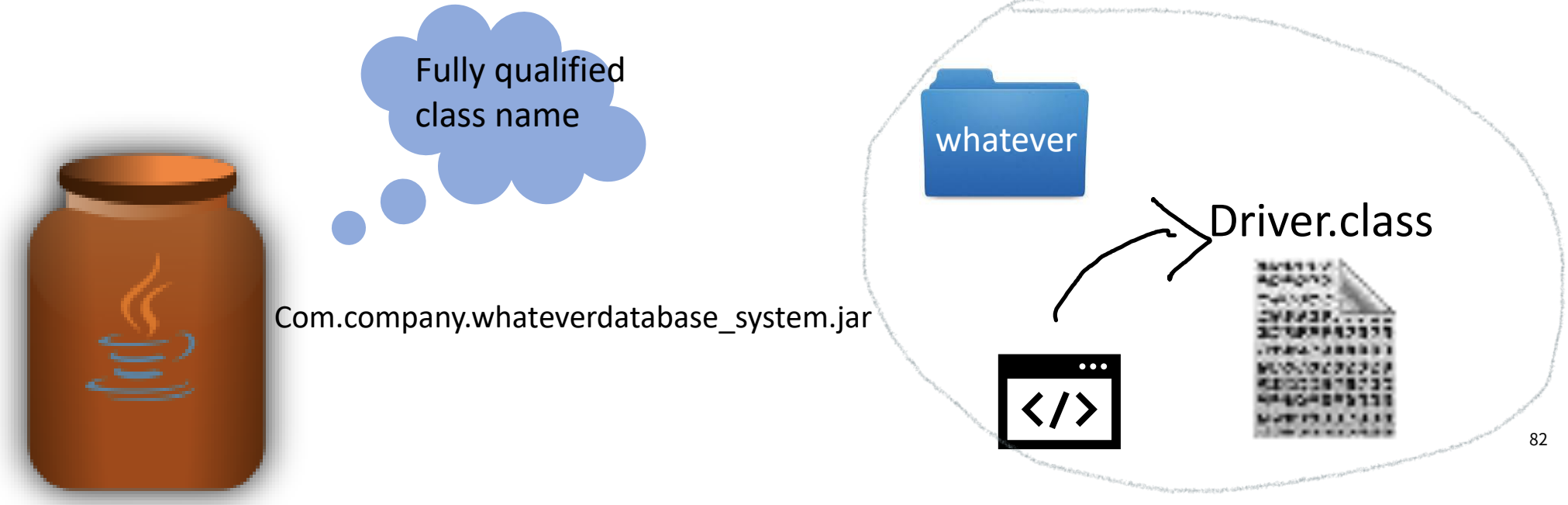
Uses of Reflection #4: Dynamically loading a class

- This is particularly useful with database access. Although there is a common language for accessing databases, database providers supply (as java archives) classes that implement the required methods to talk to THEIR system.
- More later when we discuss JDBC.

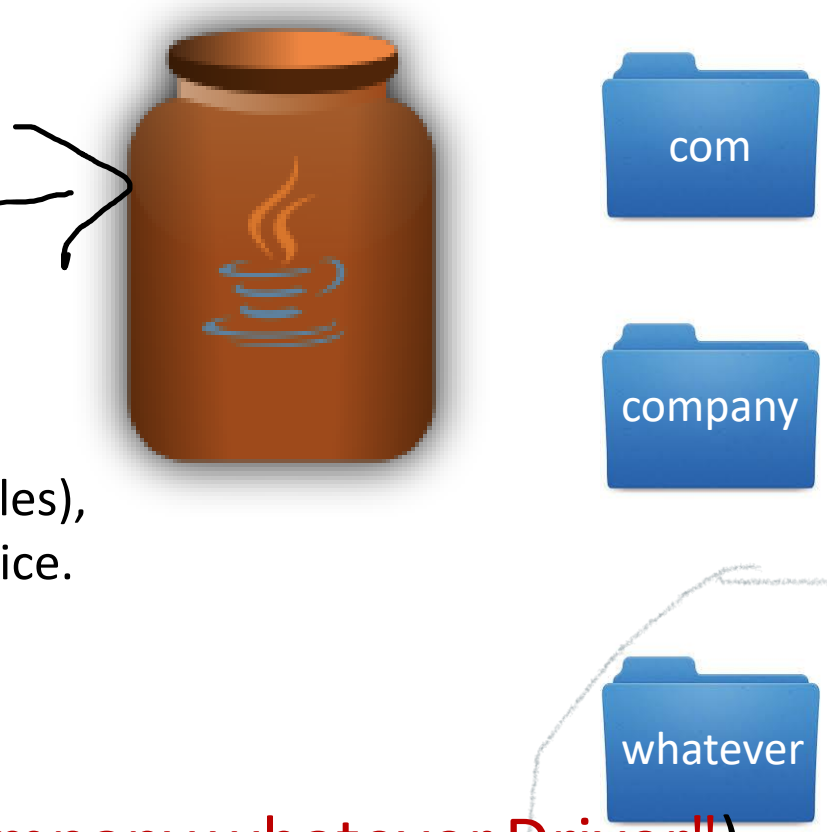


Uses of Reflection #4: Dynamically loading a class

Usually the driver has a long complicated name to ensure that there is no conflict (two different drivers cannot have the same name).



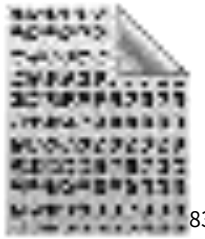
Uses of Reflection #4: Dynamically loading a class



On the CLASSPATH

If the name of the .jar file is included in the CLASSPATH (where the loader looks for .class files), then the program can load the driver of its choice.

```
Class c = Class.forName("com.company.whatever.Driver");
Driver d = (Driver) c.newInstance();
```



Exercise: Download and run Squirrel SQL*

*You should have already done this last week!

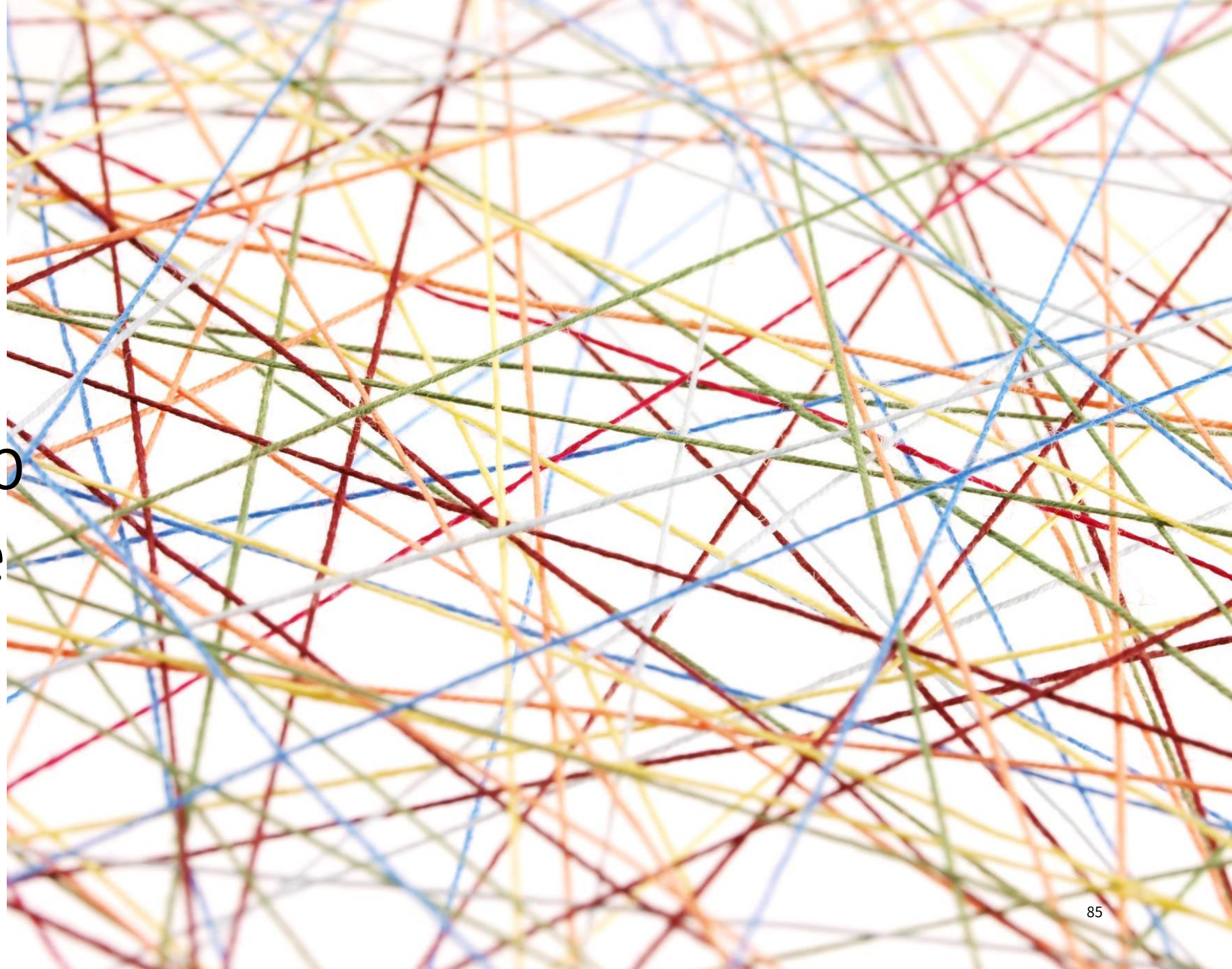
There is a Java graphical tool called Squirrel SQL that uses this to let you query almost any database system, as long as you have the suitable .jar file added to your CLASSPATH. You can switch between very different systems.

Download link here: http://squirrel-sql.sourceforge.net

Follow to the tutorial here to view some databases:

http://squirrel-sql.sourceforge.net/kulvir/tutorial.html

HashMap Example



Maps

Table 5 Working with Maps	
`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

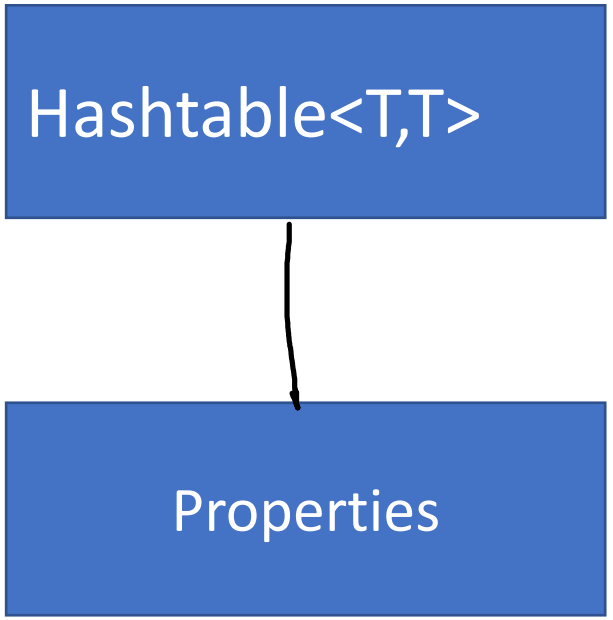
`for (String key : scores.keySet()) {`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

Properties

- One interesting (and very useful) example of hash tables is the Properties class, which associates two strings.
- java.util.Properties

- java.lang.Object
  - java.util.Dictionary<K,V>
    - java.util.Hashtable<Object,Object>
      - java.util.Properties



Key: String

Value: String

```
#Location of data files
data_dir = C:\Users\Public\Data
# Remote server
server = 192.168.1.214
# Theme name
theme = Funky
```

preferences.cnf

Example 1

Properties

When you install a piece of software on your computer, you are usually asked a lot of questions, such as where you want to install the program, the location of other resources, possibly a theme. All this information is stored in a .ini, .conf or whatever file. Each parameter is a name associated with a value. Properties objects deal with these files, can read them, write them, and ignore for instance lines starting with #.

You can define default values. Calling prop.get() would return null for a value not read from the file, prop.getProperty() returns the default if there is one.

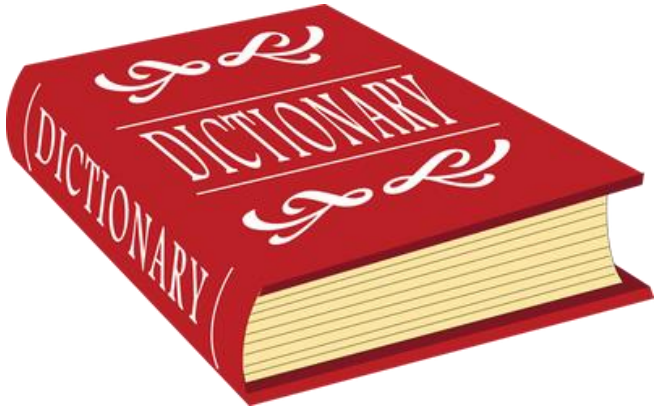
```
1
2  import java.io.BufferedReader;
3  import java.io.FileReader;
4  import java.io.IOException;
5  import java.util.Properties;
6
7  public class PropertiesEx {
8
9      public static void main(String args[]) {
10
11         Properties defprop = new Properties();
12         defprop.put("data_dir", ".");
13         defprop.put("theme", "classic");
14         Properties prop = new Properties(defprop);
15
16         try (BufferedReader conf = new BufferedReader(new FileReader("preferences.cnf"))) {
17
18             prop.load(conf);
19
20         }catch (IOException e) { // Ignore
21             System.err.println("Warning: using default preferences");
22         }
23
24         // Display the preferences
25         System.out.println(prop.getProperty("data_dir"));
26         System.out.println(prop.getProperty("theme"));
27     }
28
29 }
```

Try with resources. Remember?

Example 2

A *multimap* is like a Map but it can map each key to multiple values.

Problem: how can we find all the anagrams of every word in a dictionary?

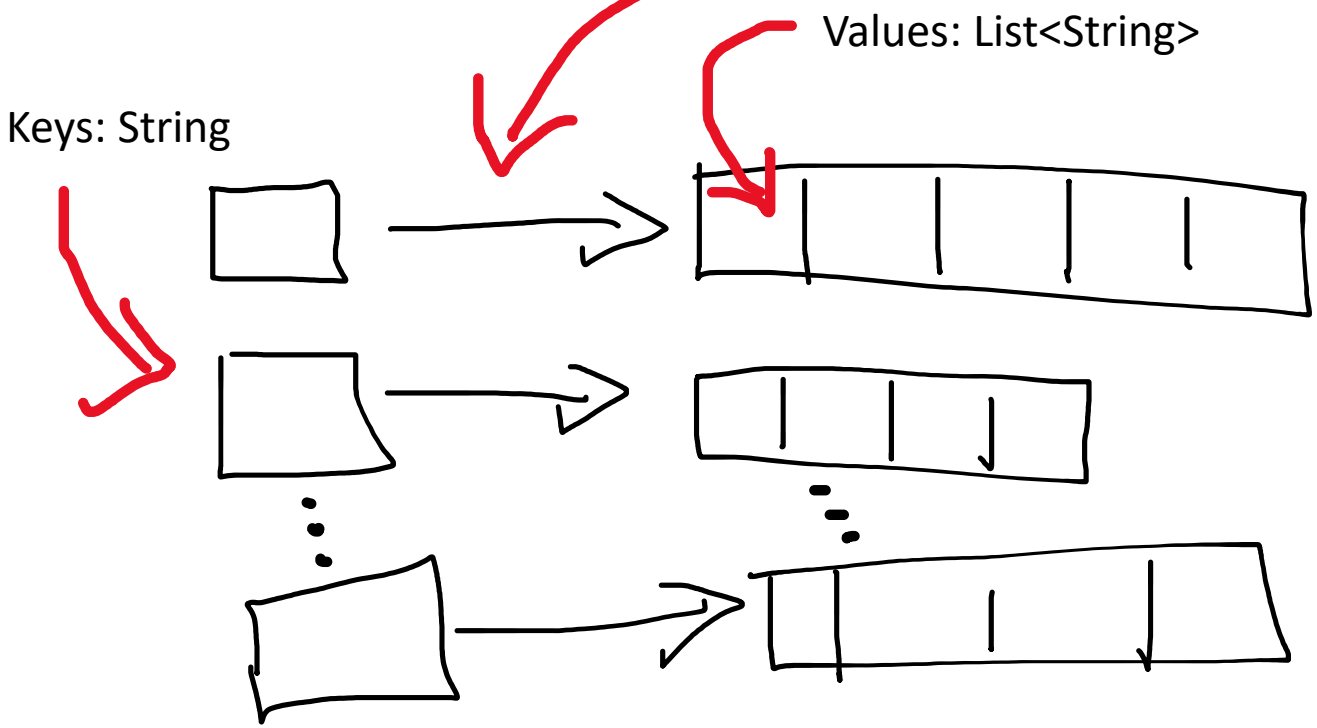


Anagrams are words that are created by rearranging the letters in a word – for example “late” is an anagram of “tale”.

A *multimap* is like a Map but it can map each key to multiple values.

- Here the map links a string to a list of strings.

```
Map<String, List<String>> m = new HashMap<String, List<String>>();
```



```java
import java.util.*;

public class Anagrams {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);

        // Read words from file and put into a simulated multimap
        Map<String, List<String>> m = new HashMap<String, List<String>>();

        try {
            Scanner s = new Scanner(new File(args[0]));
            while (s.hasNext()) {
                String word = s.next();
                String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        }

        // Print all permutation groups above size threshold
        for (List<String> l : m.values())
            if (l.size() >= minGroupSize)
                System.out.println(l.size() + ": " + l);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the **alphabetized** word to the original word.

For example, the word bad causes an entry mapping abd into bad to be put into the multimap.

A moment's **reflection** will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

```
> java Anagrams "../dictionary.txt"
5
```

5: [dates, sated, stade, stead, tsade]
6: [dearths, hardest, hardset, hatreds, threads, trashed]
6: [dater, derat, rated, tared, trade, tread]
5: [entasis, nasties, sestina, tansies, tisanes]
5: [actors, castor, costar, scrota, tarocs]
5: [delist, idlest, listed, silted, tildes]
5: [intreat, iterant, nattier, nitrate, tertian]
9: [anestri, antsier, nastier, ratines, retains, retinas, retsina, stainer, stearin]

See: https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html

Example 3

Another example uses a Tokenizer class that counts words from a text file one by one, ignoring space and punctuation.

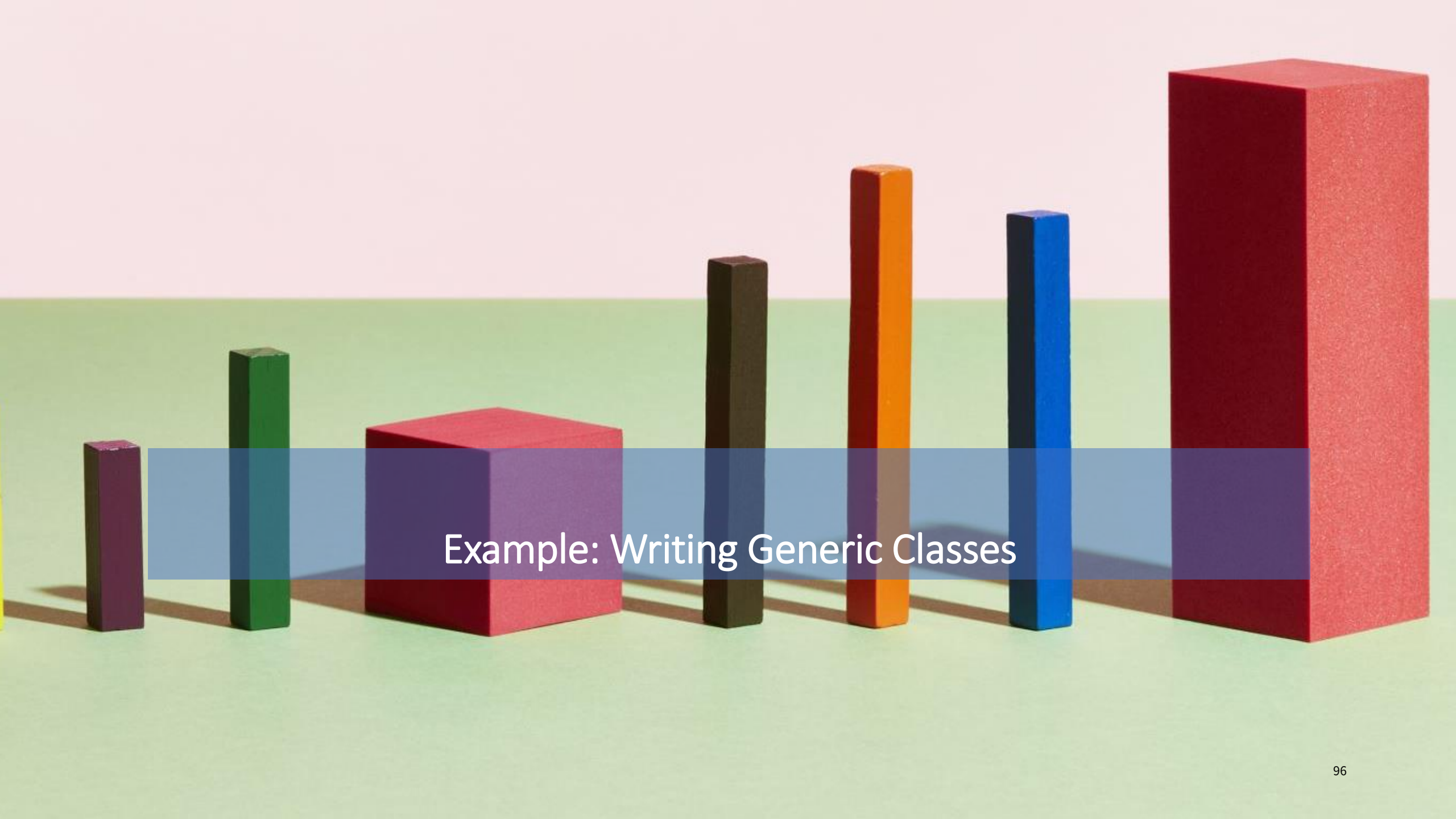
Goal: finding the 10 most used words in a speech.

We need to associate with each word a counter (so, we need a Map<String,Integer>). If we don't know the word, we store it with "1". Otherwise, we retrieve the counter, increase its value by one, and store it back.

When we have counted words, we need to find the 10 most used – we can use a map (associating a number of occurrences to a list of words, as there may be ties) but we also need some ordering. We need to go through our hash map and store its objects in, for instance, a TreeMap<Integer,TreeSet<String>>. Then we can iterate on the tree map and retrieve the most common words.

The result is usually very disappointing, because in an English speech the most common words are likely to be "the", "is", "a", and so forth. Those words are not very significant words and are usually called "stop words" (search engines on the Internet ignore them).
What we need to do is have a list of stop words, read it into an easily searchable structure such as a tree, and start counting words only when we cannot find them in this list of not important words. It gives a completely different vision of a speech.

Exercise: implement this algorithm.



Example: Writing Generic Classes

A Queue as a Linked List

To ensure that the only operations that are performed on the list are the queue operations **enqueue**, **dequeue**, and **isEmpty**, we can create a new class that contains the linked list as a private instance variable.

```
class QueueOfStrings {
   private LinkedList<String> items = new LinkedList<>();
   public void enqueue(String item) {
      items.addLast(item);
   }
   public String dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

if we want queues of *Integers* or *Doubles* or *Colors* or any other type, we can write a **generic** *Queue* class that can be used to define queues of any type of object.

Generic Class: A Queue as a LinkedList

Given this class definition, we can use parameterized types such as *Queue<String>* and *Queue<Integer>* and *Queue<Color>*.

```
class Pair<T,S> {
   public T first;
   public S second;
   public Pair (T a, S b) {  // Constructor.
      first = a;
      second = b;
   }
}
```

can be used to declare variables and create objects such as:

```
Pair<String,Color> colorName = new Pair<>("Red", Color.RED);
Pair<Double,Double> coordinates = new Pair<>(17.3,42.8);
```

It's also easy to define generic classes and interfaces that have two or more type parameters, as is done with the standard interface *Map<K,V>*. A typical example is the definition of a "Pair" that contains two objects, possibly of different types. A simple version of such a class can be defined as the class Pair here

```java
/**
 * Returns the number of times that itemToCount occurs in list. Items
 * in the list are tested for equality using itemToCount.equals(),
 * except in the special case where itemToCount is null.
 */
public static int countOccurrences(String[] list, String itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (String listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (String listItem : list)
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

For a generic method, the "<T>" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (T listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (T listItem : list)
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

For a generic method, the "<T>" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for (T listItem : list)
         if (listItem == null)
            count++;
   }
   else {
      for (T listItem : list)
         if (itemToCount.equals(list
            count++;
   }
   return count;
}
```

If `wordList` is of type `String[]` and `word` is of type *String*, then

```
int ct = countOccurrences( wordList, word );
```

will count the number of times that `word` occurs in `wordList`.

If `palette` is of type `Color[]` and `color` is of type *Color*, then

```
int ct = countOccurrences( palette, color );
```

will count the number of times that `color` occurs in `palette`.

If `numbers` is a variable of type `Integer[]`, then

```
int ct = countOccurrences( numbers, 17 );
```

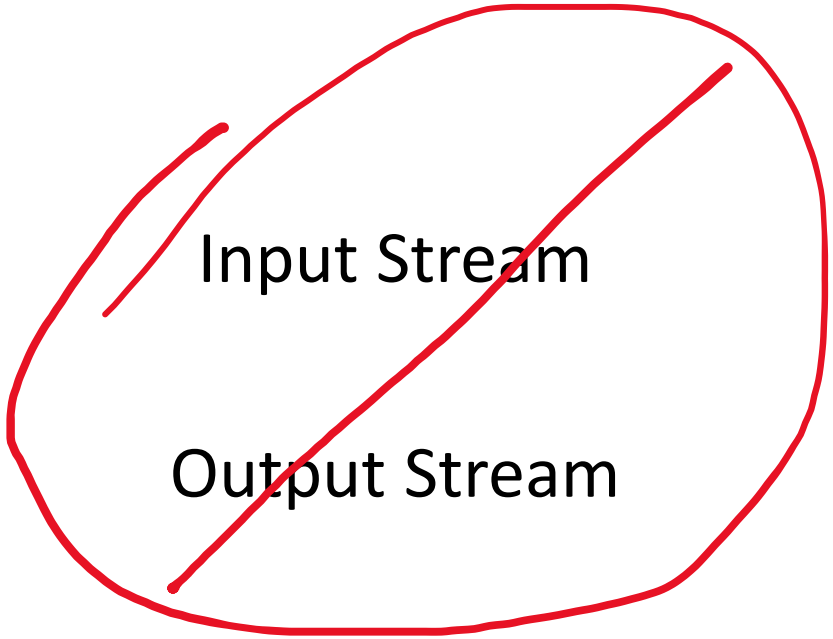
will count the number of times that `17` occurs in `numbers`.

```
public static <T> int countOccurrences(Collection<T> collection, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
   return count;
}
```

The countOccurrences method operates on an array. We could also write a similar method to count occurrences of an object in any collection…

Streams

Streams



They are not input streams or output streams

Beware that in spite of the name, it's unrelated to files.
It's about chaining processing.

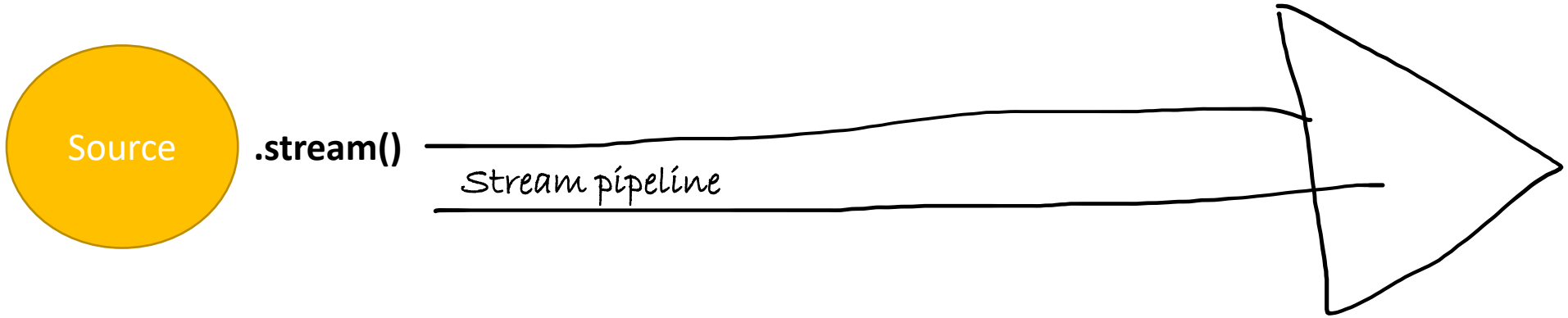
Streams

- Can demonstrate the concept with String operations
- When you apply a method to a string you can apply a new method to the result… and so on.

```
String str = “now let’s have some fun”;
Str.toUpperCase().replace(’S’, ‘D’).substring(15, 19).replace(‘M’, ‘N’);
Str                                                    "now let's have some fun"
        .toUpperCase()                                  "NOW LET'S HAVE SOME FUN"
                .replace(’S’, ‘D’)                     "NOW LET'D HAVE DOME FUN"
                        .substring(15, 19)             "DOME"
                                .replace(‘M’, ‘N’);    "DONE"
```

Streams

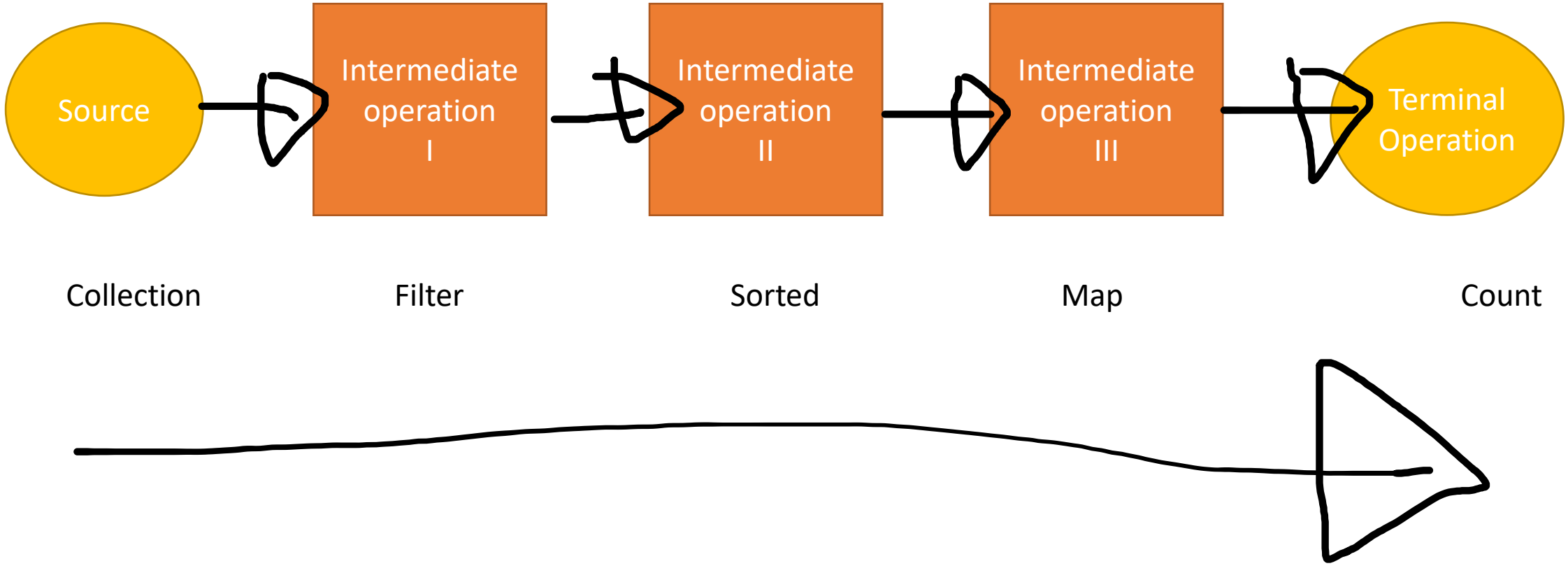
- The same idea may be applied to collections, arrays, functions, IO and others:



- Because serial processing is often applied to collections of data, the idea is to do something similar in Java
- If you know SQL this will not be new as it is also based on similar ideas

```sql
SELECT students.name, students.lastname
FROM students
WHERE students.lastname IS NOT NULL
UNION

SELECT students.name, 'N/A'
FROM students
WHERE students.lastname IS NULL;
```

Stream Pipeline



Streams

- Intermediate operations
  - Filter
  - Distinct
  - Sorted
  - Map

With stream operations you have methods that return a stream (which can be fed into something else), and those that don't and terminate the streaming process.

- Terminal Operations (return a value by combing stream contents)
  - count
  - min/max
  - average
  - reduce (you set your own accumulator function – can use anonymous functions!) **(https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html)**

Another Example

Suppose we have an array of Film objects read from a file:

```
class Film {
        private String      title;
        private String      countries;
        private int         year
        private double      billionRMB

public Film(String title, String countries, int year, double billionRMB){
        …
}
…
public toString() …
}
…
ArrayList<Film> films = new ArrayList<Film>();
```

CSV

We can build a collection read from a file, and then the problem is how to search this collection. We can search on many different criteria – film title, year of release, country, how much it made so far.

Retrieve information using different conditions…

Remember predicate – built in functional interface?

```
import java.util.function.Predicate;
static void filter (Predicate<Film> pred){
        Film f;
        ListIterator<Film> iter = films.listIterator();
        while(iter.hasNext()){
                f = iter.next();
                if (pred.test(f)){
                        System.out.println(f);
                }
        }
}
```

The predicate functional interface has a single method called test

Remember predicate – built in functional interface?

- Using this method we can very concisely apply different filters for country, year, revenue, etc:

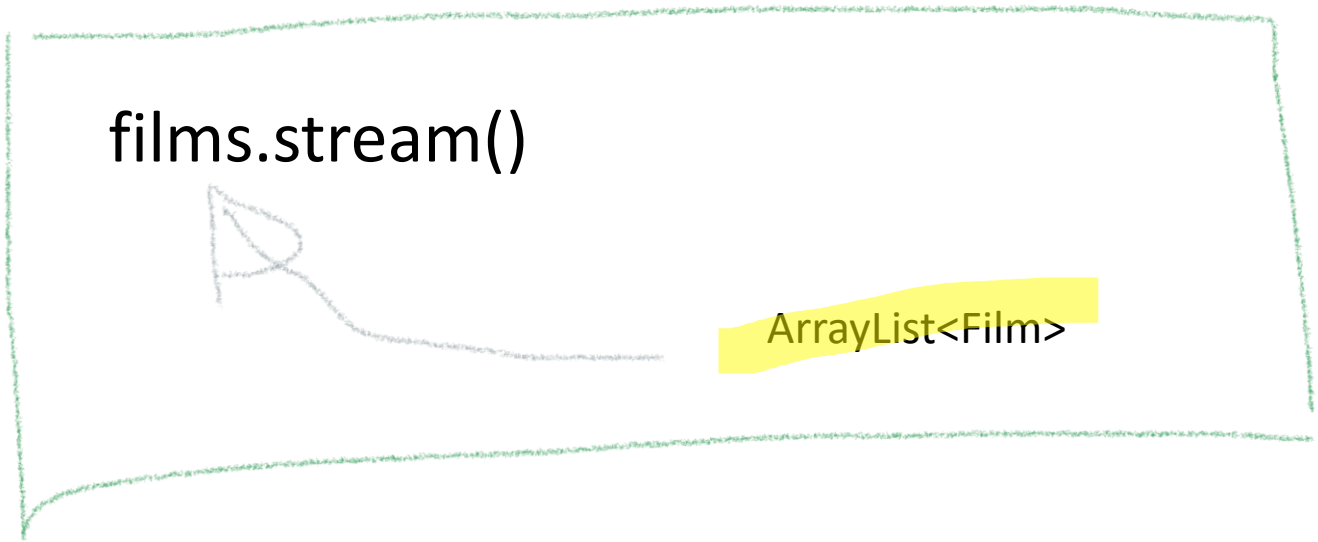
filter((film)->film.getYear() == 2014);

- To extract the films from a collection meeting the condition that the year is 2014

```
import java.util.function.Predicate;
static void filter (Predicate<Film> pred){
        Film f;
        ListIterator<Film> iter = films.listIterator();
        while(iter.hasNext()){
                f = iter.next();
                if (pred.test(f)) {
                        System.out.println(f);
                }
        }
}
```

Streams

- First we take the collection and turn it into a stream:



- Then apply serial operations ending in a termination.

```
films.stream()
        .filter((film)->film.getYear() == 2014)
        .forEach(System.out::println);
```

We can display any film that "gets through" with a forEach() call (a terminal operation) that applies println() to it.

Note the use of the double colon notation that is used here to specify a method applied to each element.

You can insert intermediate operations before the terminal one, for instance sort the output, if of course Java knows how to sort Film objects.

Note that it's usually FAR more efficient to sort AFTER filtering rather than BEFORE filtering because the set to sort is smaller, even if both are possible ...

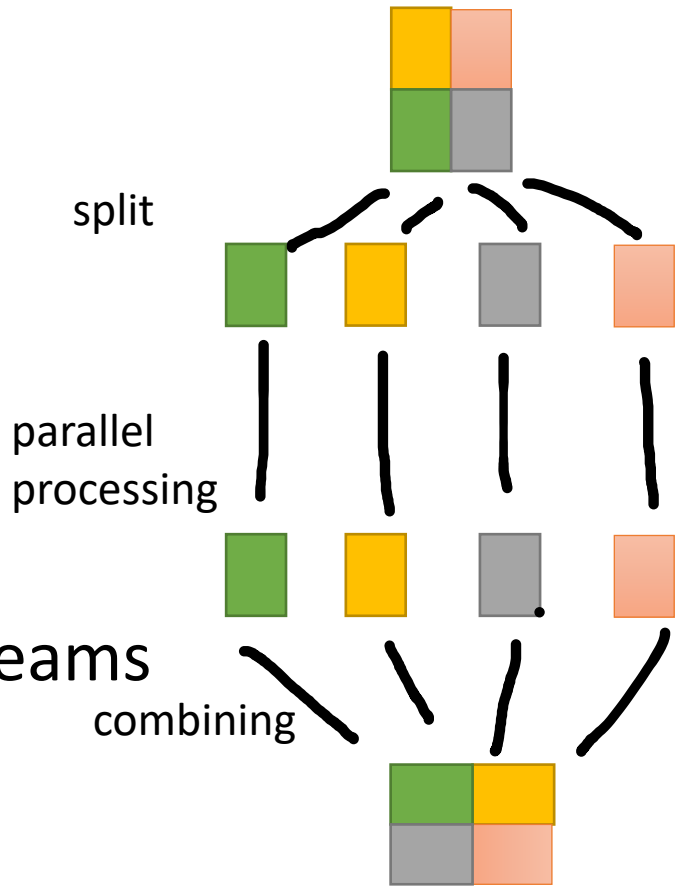
```
films.stream()
        .filter((film)->film.getYear() == 2014)
        .sorted()
        .forEach(System.out::println);
```

NOTE: needs to implement Comparable

Parallel Streams

.parallelStream()

Like a river that reaches the sea with a delta, streams can be split into multiple parallel streams for faster processing. This is important in applications of big data.



https://docs.oracle.com/javase/tutorial/collections/streams/index.html

further examples and discussions of streams can be found in the docs here

Anonymous Classes for Event Handlers

Nested Classes

```
class OuterClass {
        …
        class NestedClass {
            …
        }
}

class OuterClass {
        …
        public void doSomething() {
                class LocalClass {
                    …
                }
        }
}
```

Nested classes are classes declared inside other classes….

In Java you can also have local classes which are classes declared inside a method.

Why?

- Encapsulation
- Grouping related functionality together

- These nested or local objects are not needed outside the context in which they are declared (within another class or in a method)

It is useful for code clarity, but it also means that the objects are kind of "technical objects" that aren't expected to be used anywhere else and you won't have a reference to them generally available elsewhere.

Anonymous Classes

```
Class NamedClass implements Interface{
    …
}

Interface anObject = new NamedClass(…)
```

We saw a similar case with anonymous classes where some functionality requires an interface. For example an object that implements the “comparable” interface is often not really needed by a caller or other program parts generally – all that is needed here often is to provide a sorting utility method with a comparator. Other interfaces may have additional attributes and methods that are also not needed by the caller so would be unnecessary work to implement the object completely.

Of course one can however define everything and create an object however…

Anonymous Classes

```
Interface anObject = new Interface() {
                    // attribute and method implementation
                    };
```

And the Java syntax also allows creating an interface on the fly, and you can also directly instantiate an object by implementing the interface methods here…

This is very convenient for passing as a method parameter

Anonymous Classes

Also works for inheritance

```
class NamedClass extends ParentClass{
   …
}
NamedClass anObject = new NamedClass(…);
```

or

```
ParentClass anObject = new ParentClass(){
                    // attribute and method defitions
                    };
```

GUI: Event Handlers

This is common for graphical user interface programming for instance event listers. Here is an example of listening for a button press…

```
…
Button but = new Button();
but.setText(”Say ‘Hello’”);
But.setOnAction(new EventHandler<ActionEvent>(){
        @Override
        public void handle(ActionEvent e){
                System.out.println(“Hi”);
        }
});
…
```

Only one method!

Only one method!

- If there is only one method then we can use Lambda expressions instead

In the very common case where your interface requires a single method, you can use lambda expressions. Lambda expressions come from functional programming, where you try not to store any state (which is completely opposed to attributes that store the state of an object ...)

- Cool!

Lambda Expressions

- Lambda expression

(parameter list) -> {statements}

- Recall that Lambda expressions are assigned to functional interfaces

```
@FunctionInterface
```

Functional interfaces have only one abstract method

- If there is only one method to define for an interface you don't have to give it a name

Using Lambda Expression in Event Listener

```
…
Button but = new Button();
but.setText(”Say ‘Hello’”);
But.setOnAction(new EventHandler<ActionEvent>(){
        @Override
        public void handle(ActionEvent e){
                System.out.println(“Hi”);
        }
});
…
```

Anonymous Class

Using Lambda Expression in Event Listener

Using Lambda Expression in Event Listener

```
...
Button but = new Button();
but.setText("Say 'Hello'");
But.setOnAction(new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent e){
        System.out.println("Hi");
    }
});
...
```

Anonymous Class

As "handle()" is the only method of an event handler, it can also be written like this:

```
Button btn = new Button();
btn.setText("Say 'Hi'");
btn.setOnAction((e)->{
    System.out.println("Hi!"); });
```

Much shorter

Other common uses are in searching and sorting collections where if you recall the comparable interface has only one method.

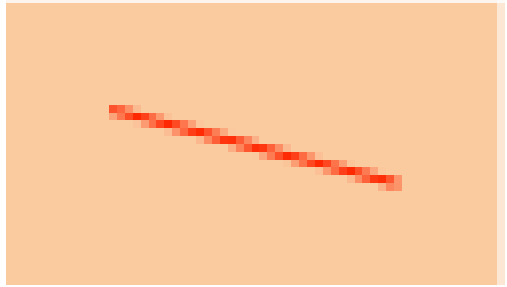
Graphical User Interfaces I

Graphical User Interface Programming

- Event driven programming is a different paradigm for creating a sequence of instructions and possible interactions with a user
- Event handlers / ”callbacks” are functions associated with certain events
- Different from other types of programming you are familiar with
- You rely on packages which you must import and use in your own program (there are many options)
- Need to interact closely with an environment (operating system, windows, display, etc)

Many Different Packages

- First of all you don't code everything by yourself, and instead use functions from packages that you must import when writing your program.
- There are **low-level** packages with functions (called "primitives") for performing tasks such as drawing a rectangle, a line or a curve or 3d rendering.
- You also have **high-level** packages that use the previous ones to draw for instance buttons, and automatically change them when they are clicked.



Historical Overview of Java GUI Packages

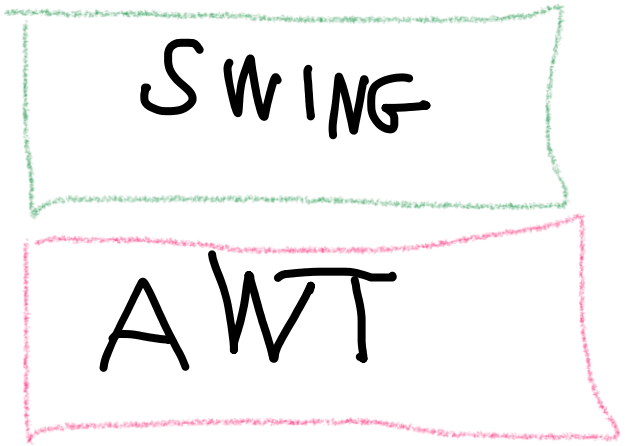
- 1995 Abstract Windows Toolkit (‘**AWT**’)
  Looks like other applications on the system

- Dec 1996 Java Foundation Classes (‘**Swing**’)
  Looks the same on all systems

In Java, several packages allow you to code a GUI. The first one was AWT, followed by "Java Foundation Classes" quiclky renamed "Swing".

Historical Overview of Java GUI Packages

```
import java.awt.*;
import javax.swing.*;
import javax.imageio.*;
```

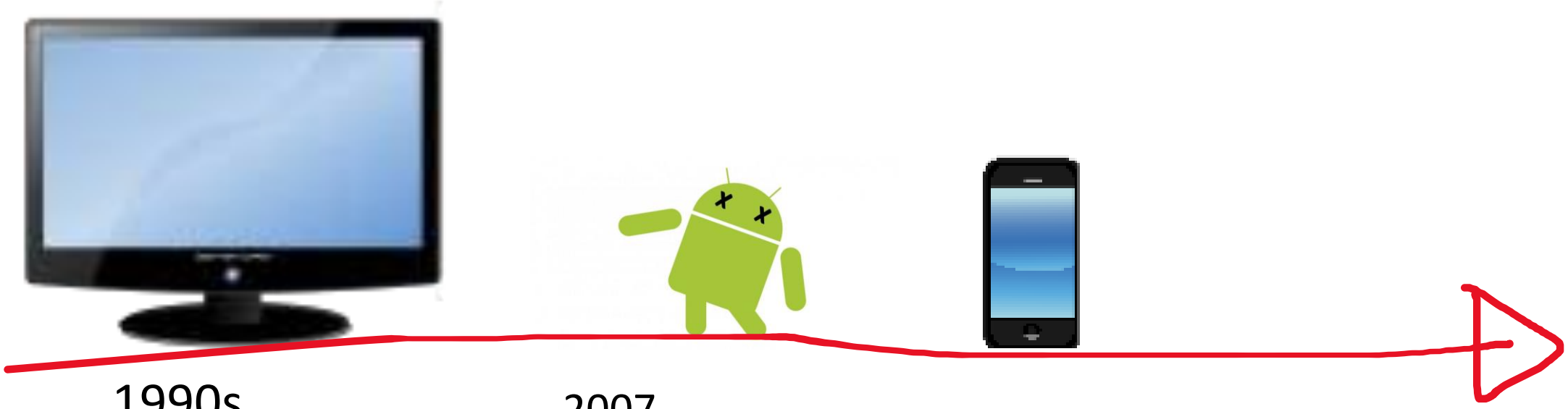


Swing relies on AWT, and whenever you code a Swing application you also need to import classes from AWT, as well as from other packages for images.

Historical Overview of Java GUI Packages

- A new package, JavaFX, was introduced in 2008.

2008 JavaFX `import javafx.*;`



JavaFX

JavaFX, with which you import classes from a single package (but many subpackages) supports other devices than computer screens for which AWT and Swing were written – mobile phones in particular. It also allows to define the looks of applications in external files called "style sheets" or "CSS" files (CSS means "Cascading Style Sheet" – 'cascade' is French for 'Waterfall'), a technique borrowed from web programming. However, because software has a long life, there is a lot of Swing around, Swing is still much in use and will probably stay around for quite a while. It's good to know both Swing and JavaFX (they aren't VERY different, class names change, basic ideas are the same).

Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

The Basic Structure of a JavaFX Program

The `javafx.application.Application` *class defines the essential framework for writing JavaFX programs.*

Historical Overview of Java GUI Packages

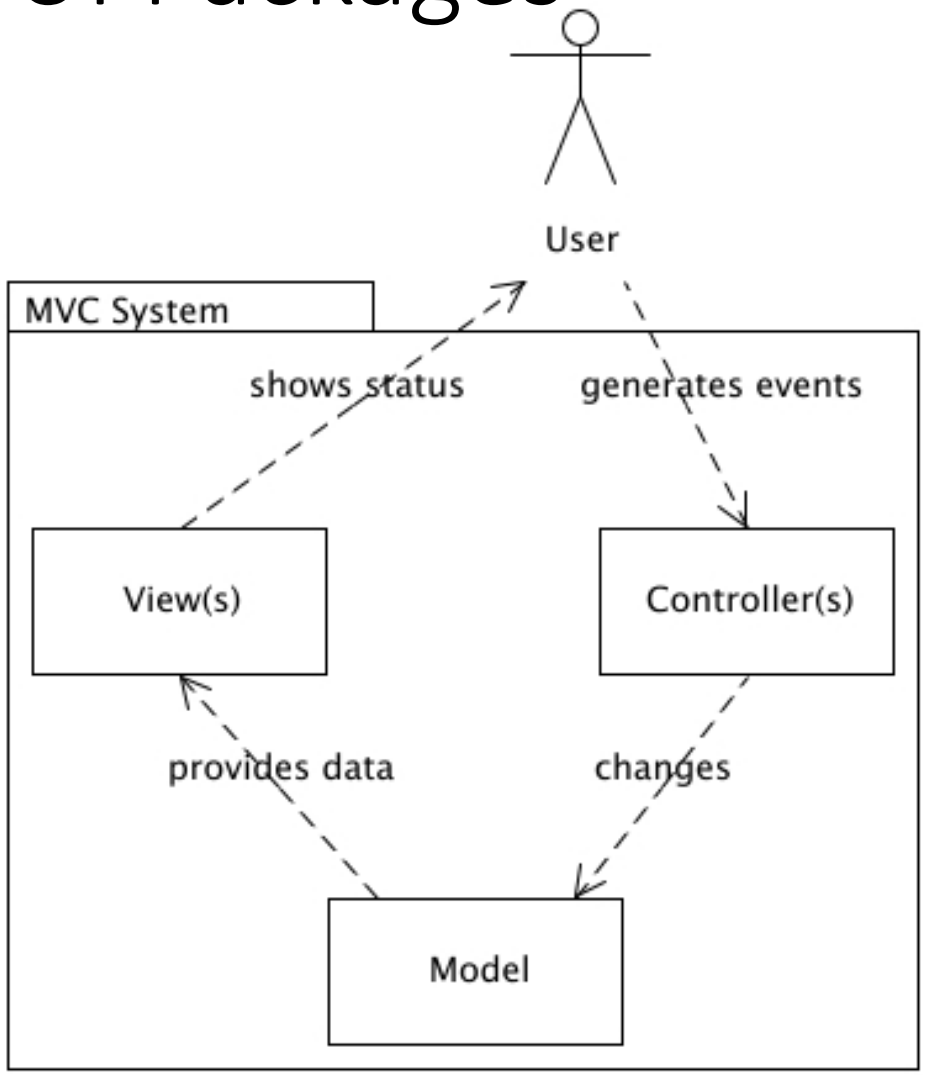
2008 JavaFX import javafx.*;

Model = data management

view = user interface (visual elements/looks)

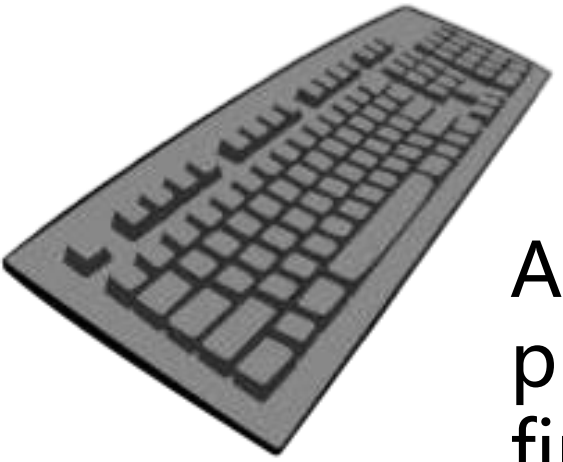
Controller = Logic

JavaFx applications often follow a popular structure known as "Model/View/Controller" (or MVC) in which data management, user interface and logic are clearly separated.

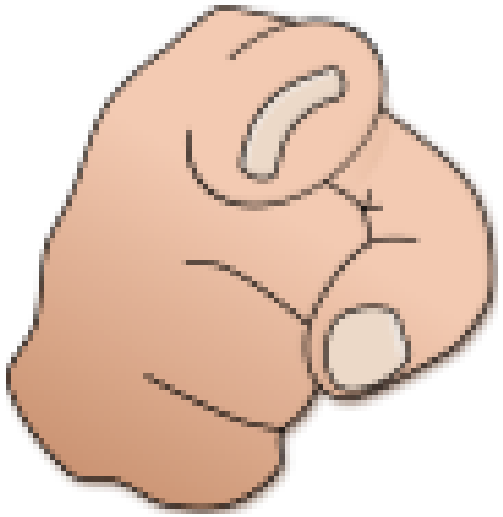


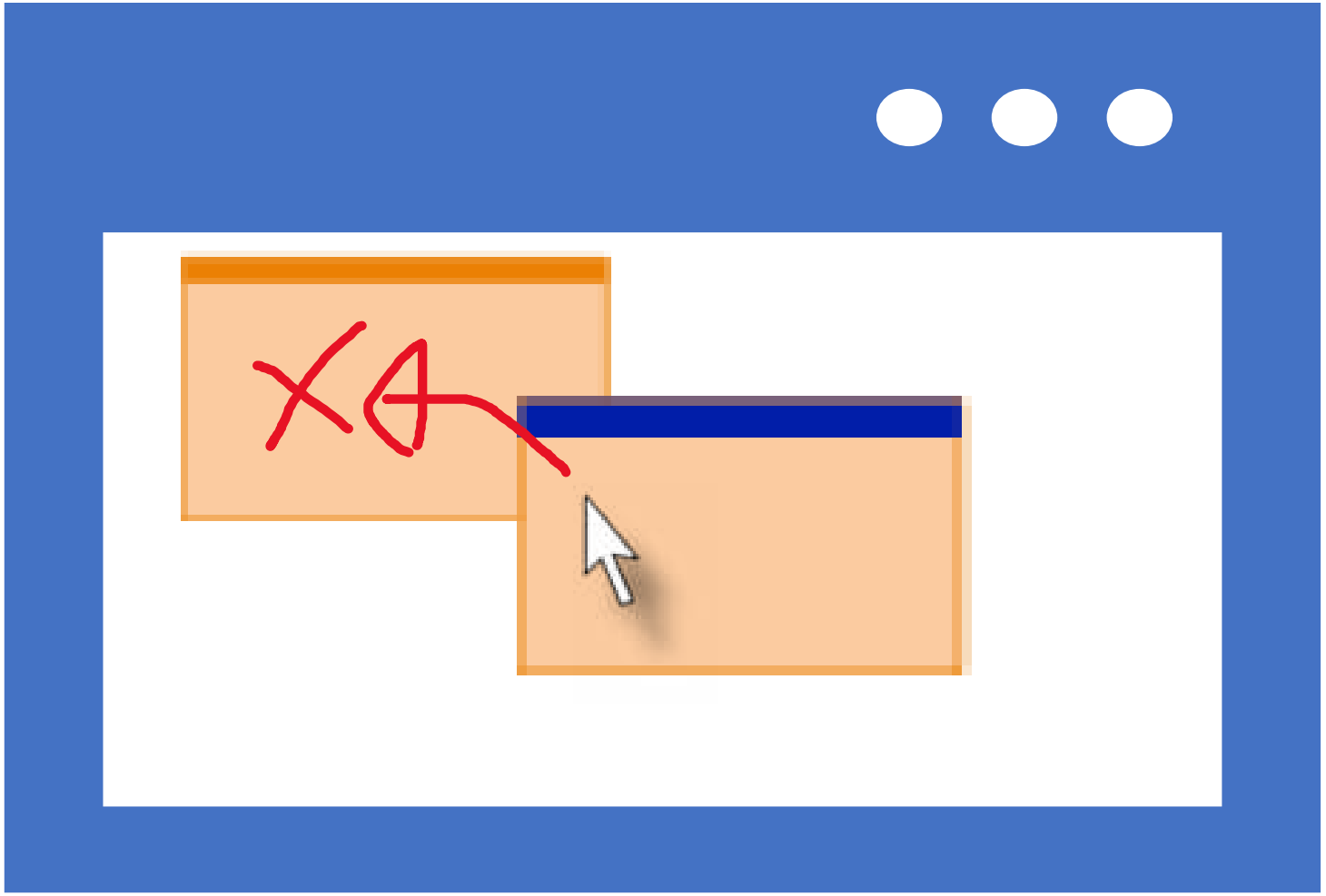
Event Driven Progamming

A Graphical Application is just a big loop



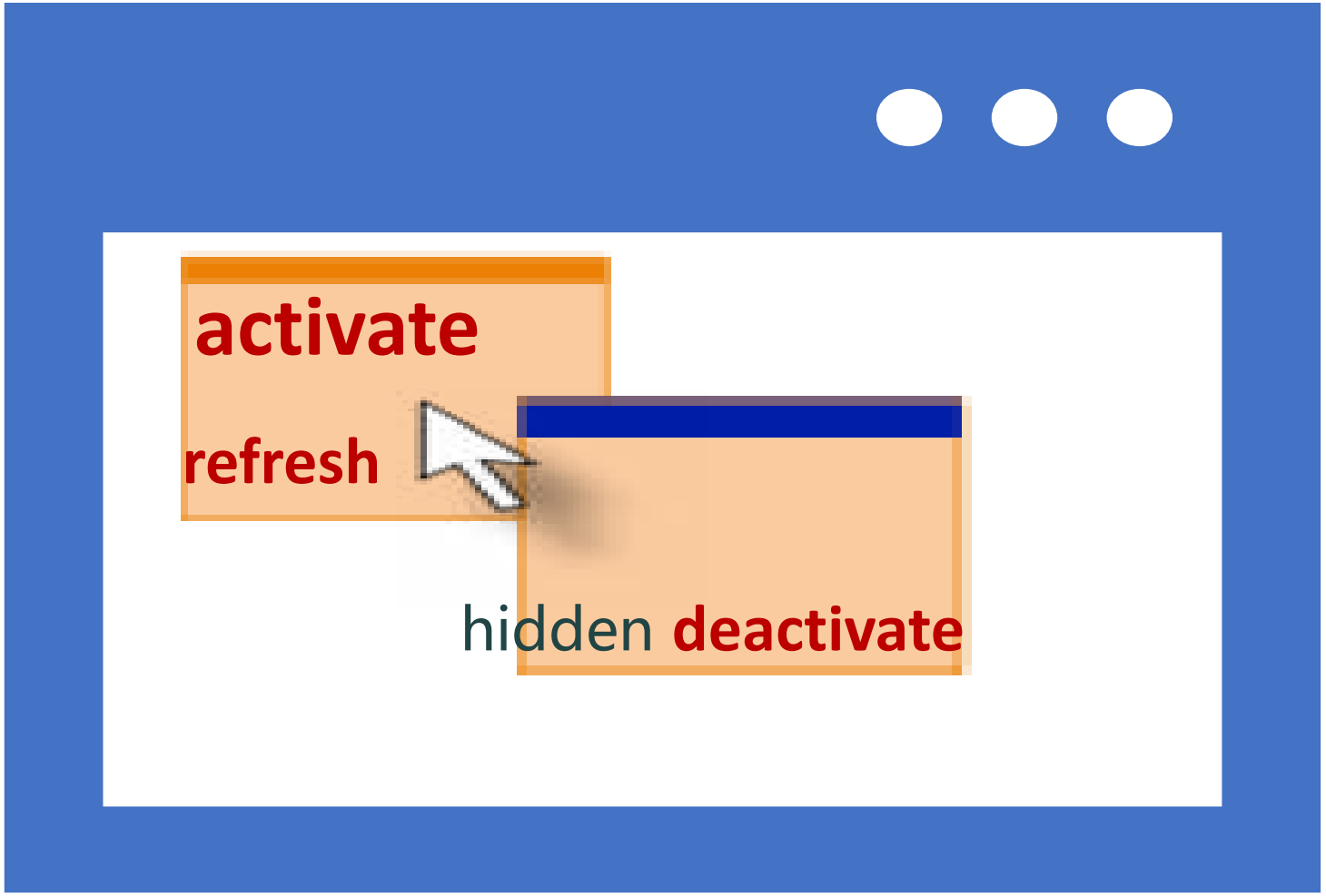
An event can be anything. A key pressed, a move of the mouse, a finger swiping a touch screen, somebody jumping in front of a webcam ... Anything that can be translated into an electrical signal reaching the computer.





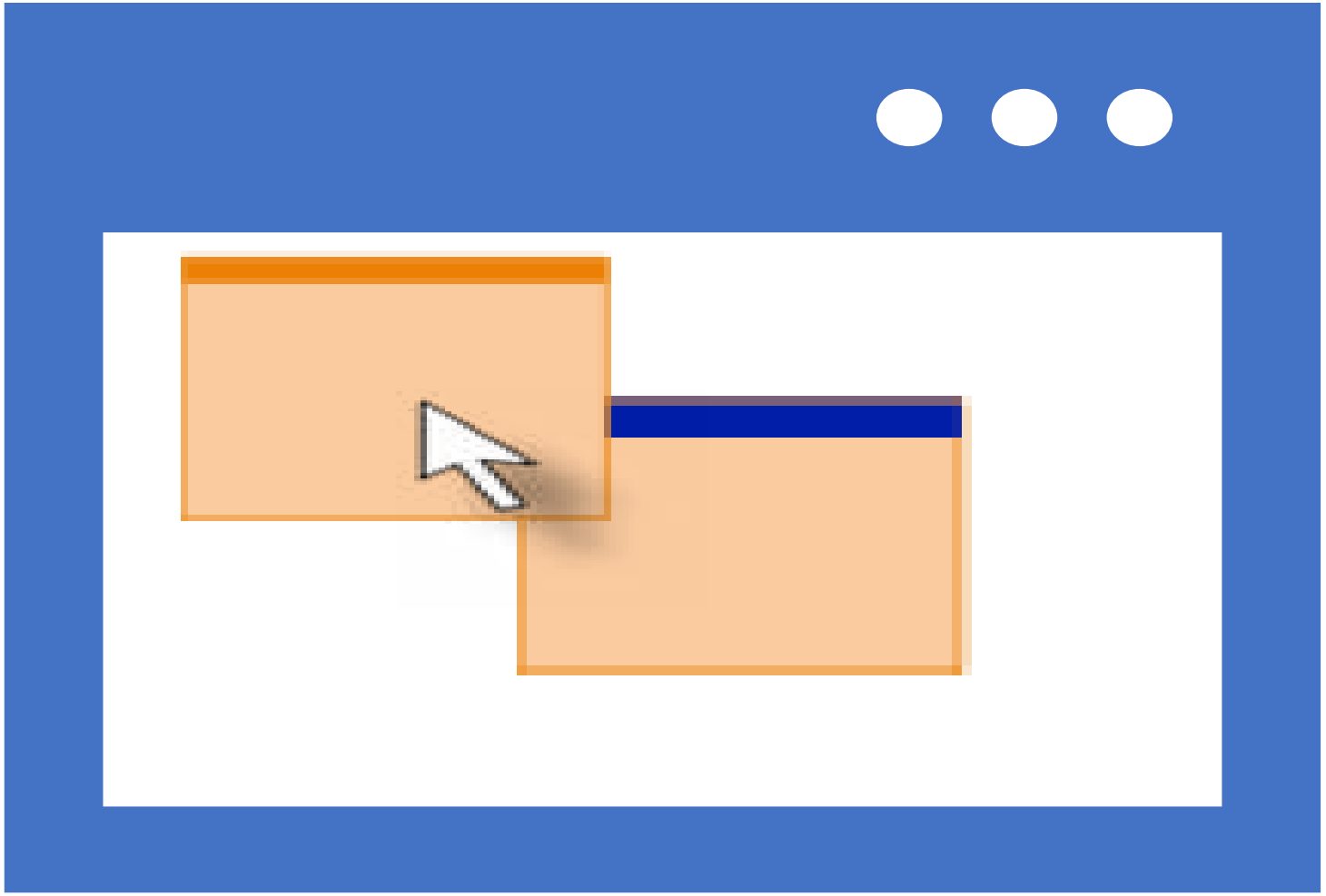
The user may move the mouse and click it outside the active window. The windows management system will get the coordinates and discover that another window is at this location.

This is an event



This means the inactive window has to be brought forwards. Note: there is a stack of objects to represent this internally.

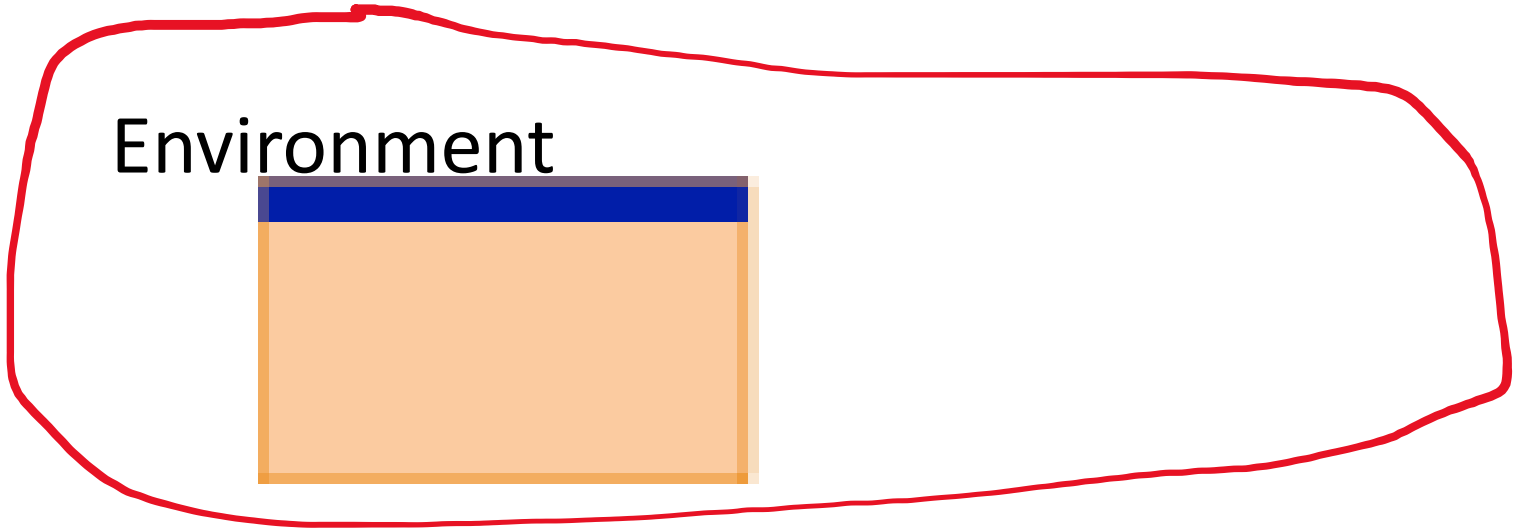
The previously active window must be redrawn to show it is inactive, the newly active window must be redrawn too. Including what was previously hidden.



Then your screen will look like this. All this stuff is performed by the window manager and doesn’t require you to write any code.

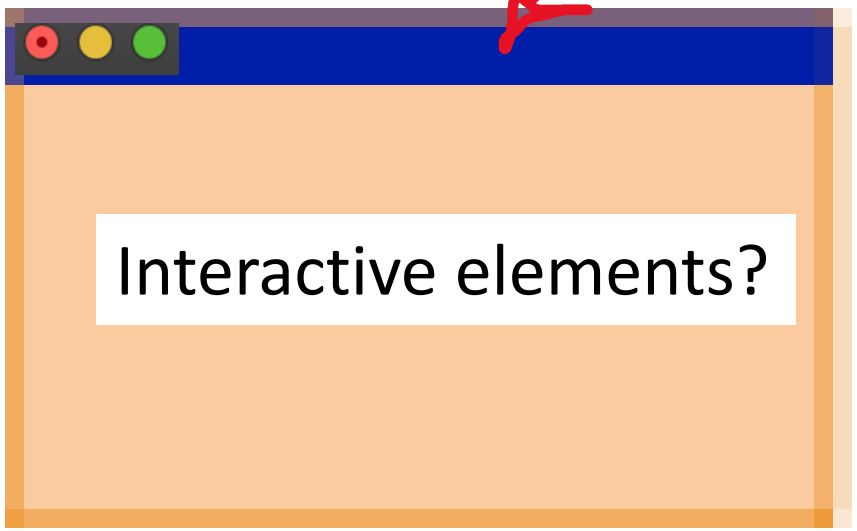
Environment

Observe that the GUI is running within an environment that provides services for managing windows (resizing, destruction, etc).



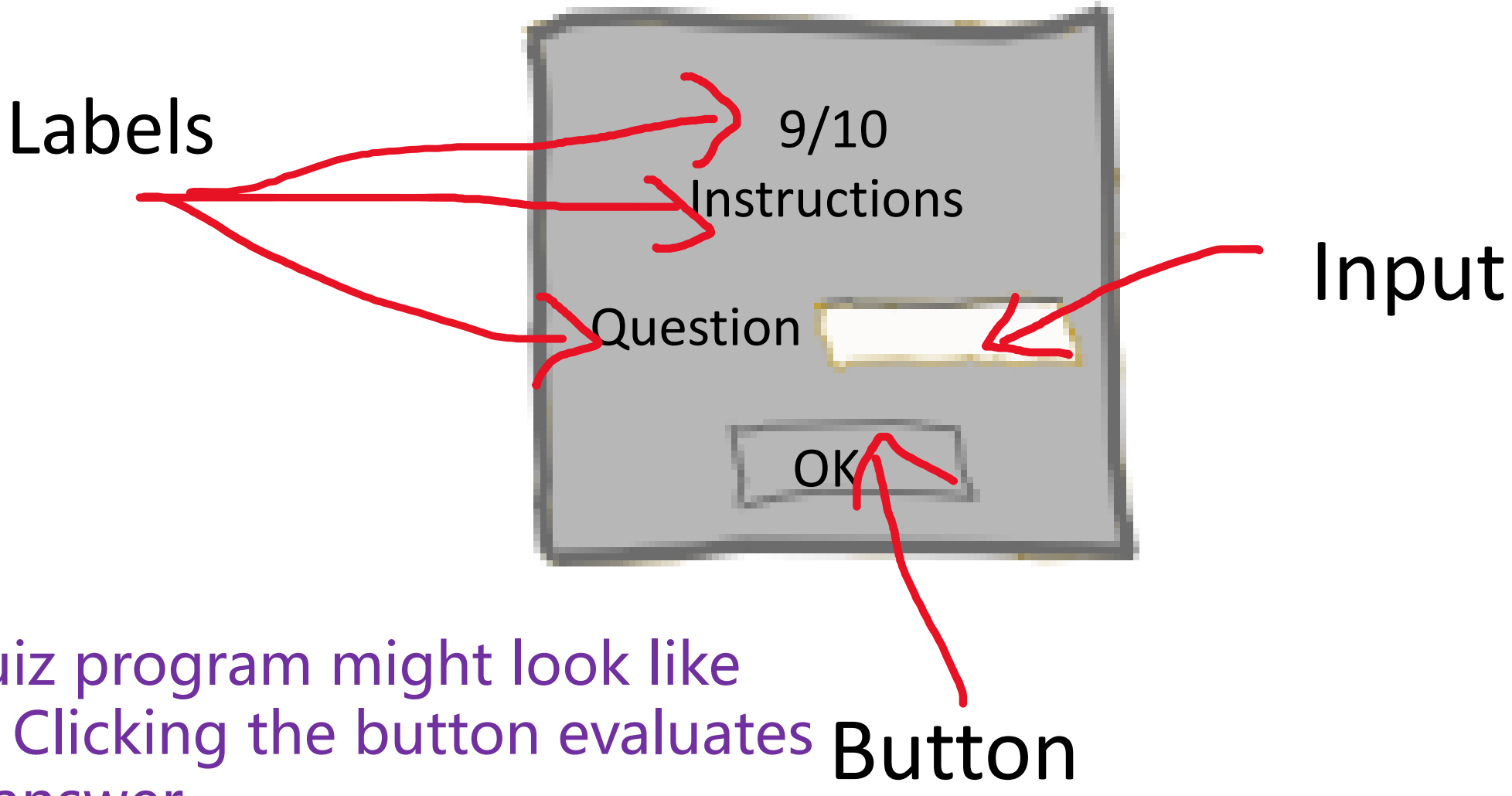
When you design your application you must decide on what the user will see: will your window have a title, will it be resizable, which elements will the user interact with in the window?

title?



What does the user see?

Components of a window…

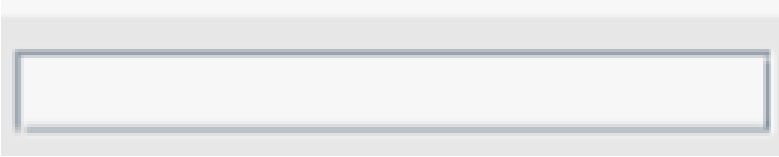


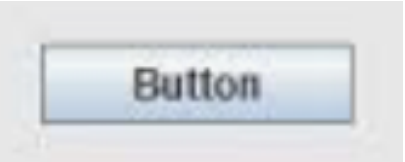
A quiz program might look like this. Clicking the button evaluates the answer.

Your program must prepare everything in advance and, like a conjuror, only reveal elements at the right time when they should be visible.

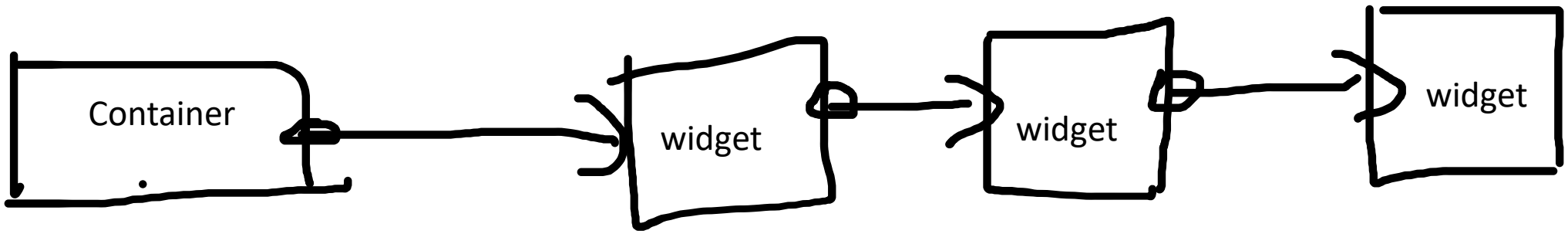
Widget = Window Gadget

- Visual elements are called “widgets” - short for window gadgets
- Some widgets include:
  - labels
  - entry fields
  - drop down lists
  - check boxes
  - and buttons



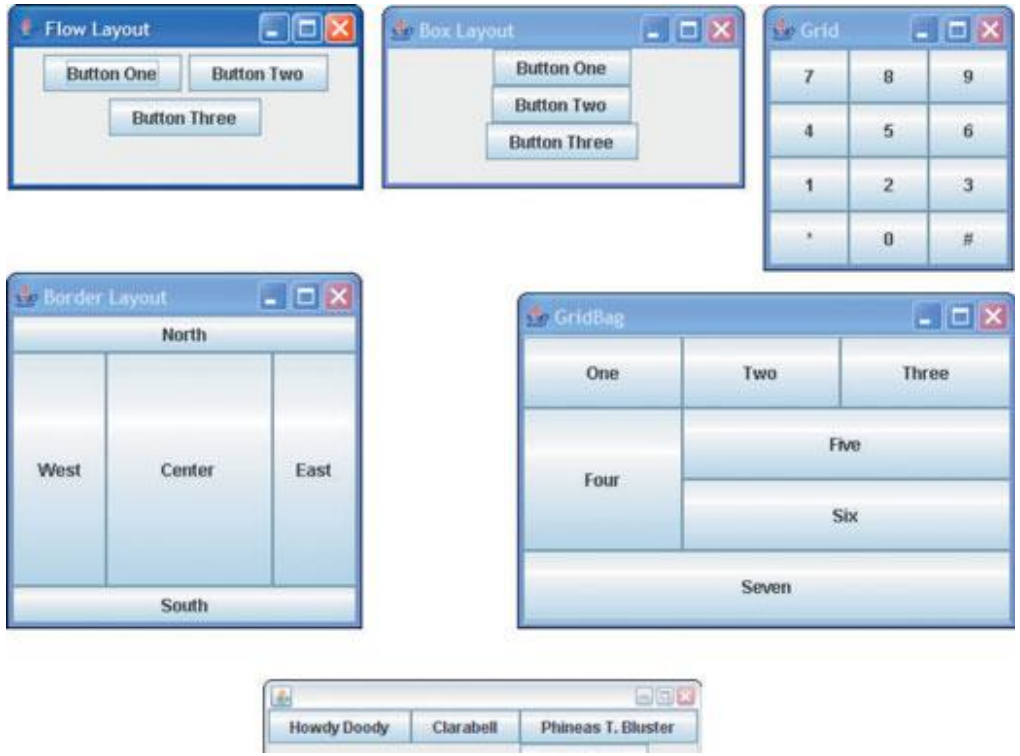


Containers



Something that you are probably not familiar with is the notion of "container". If you see widgets, you don't see containers that are nothing more than linked lists of widgets and (more about this soon) other containers.

- The purpose of containers is to make creating a layout easier.
- A layout means how the various widgets are displayed on the screen in relation to each other.



If you have a fixed-size window, things are easy. You can say "I want this widget to appear at these coordinates relative to the upper-left corner of the window".

Unfortunately the easy case isn’t the most common…

Some containers are fixed

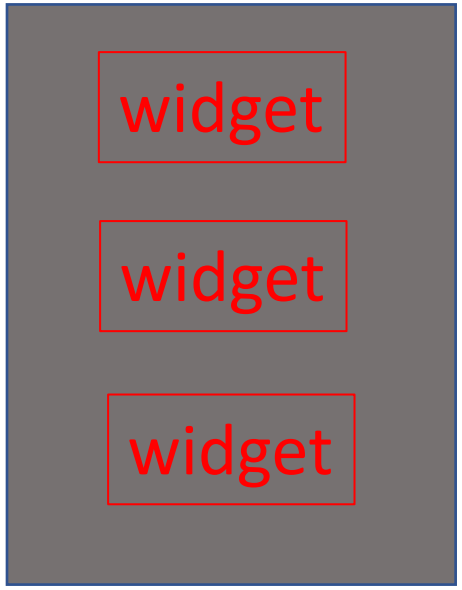
Usually people can and do resize windows and you want a "fluid layout" . If you gave absolute coordinates assuming a given window size it will soon be a big mess.

Most are not

Boxes

Containers are for solving these issues. Boxes come in two flavours and display widgets next to each other (with some padding in between) in only one direction.

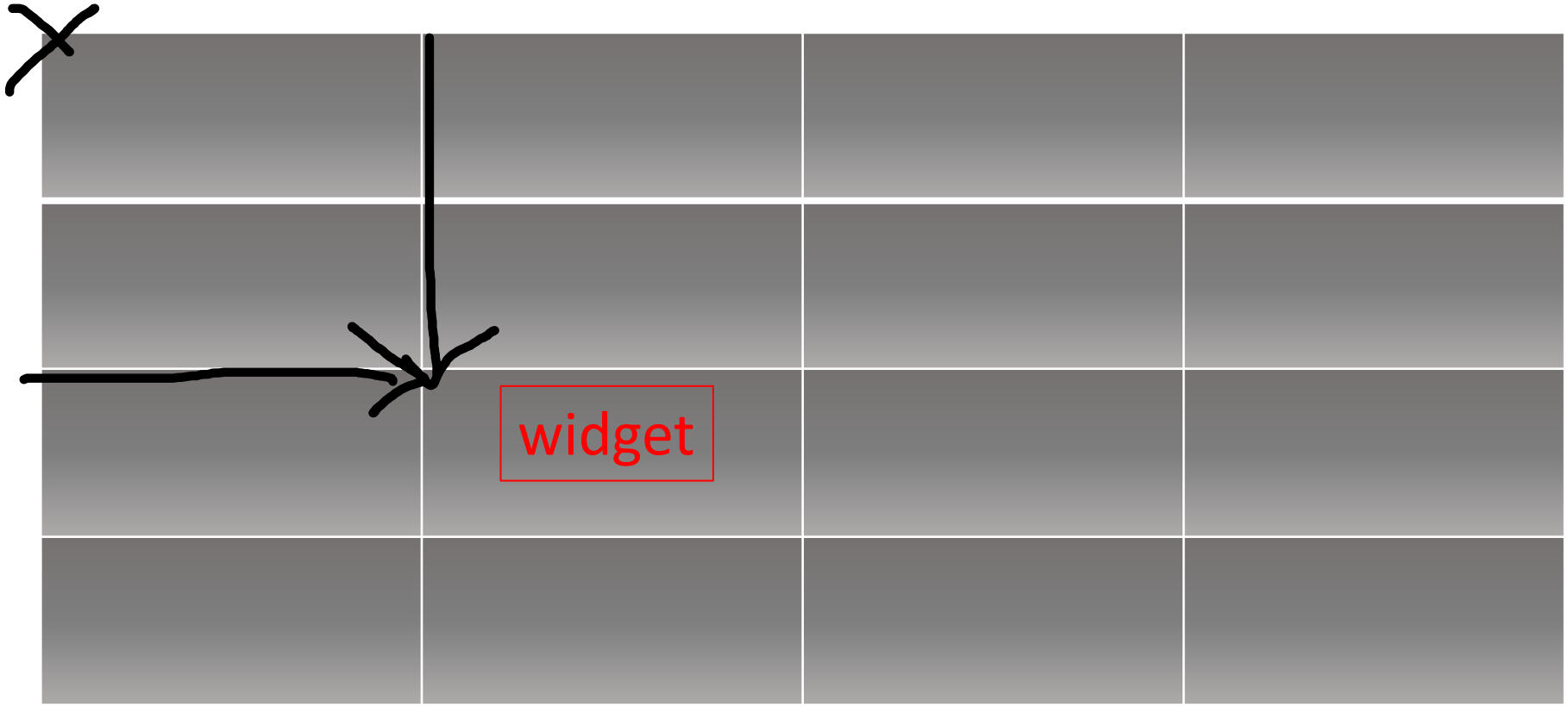
Horizontal Boxes



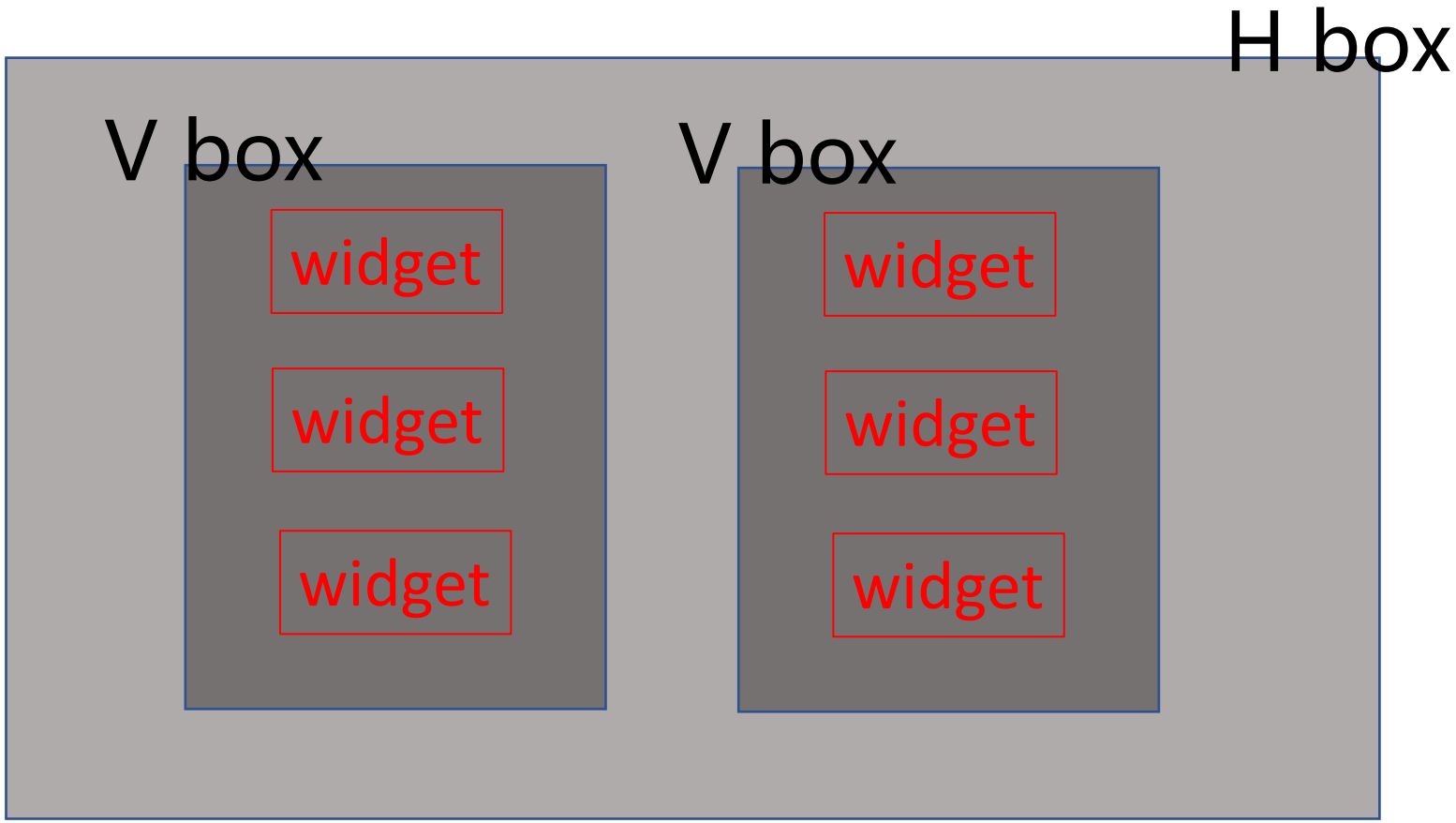
Vertical Boxes

Grids

Other frequently used containers are grids which allow you to place widgets at **relative** row column coordinates instead of absolute distance coordinates



For instance you can have two vertical boxes (each one showing widgets vertically) and add them to a horizontal box (side by side). When the window is resized, the global layout is respected and it still looks (more or less) as intended



Callback

- The last important idea to understand with graphical user interfaces is the one of "callbacks", often called "handlers" in Java, which is the name given to a function associated with an event. For instance, clicking a button might trigger a search inside a database. This is a function that you write, and associate with the button.

- A “callback” is a function associated with an event

Predefined Events

There is an extensive set of predefined events. You only handle those that matter to you. You must often perform a number of checks when the window is destroyed. For instance a text editor will ask you whether you want to save your changes.

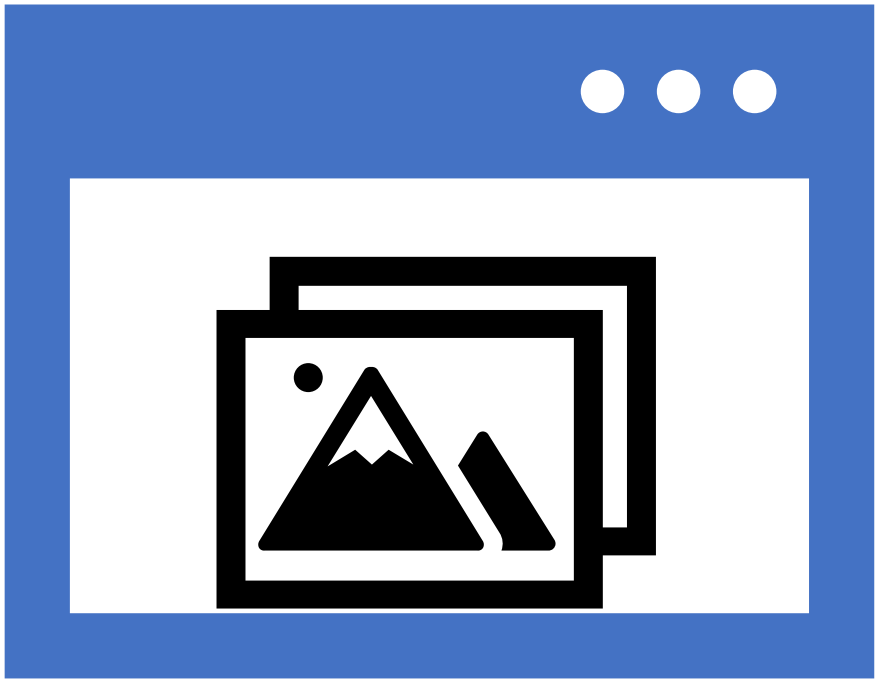
Destroy window

Button press / release

Key press / release

Focus in / out

Move in / out

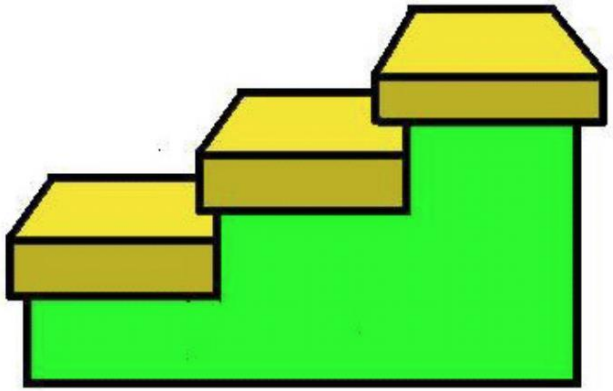


Your graphical application will run in a Window Manager that also reacts to events. For instance you may have an active window and an inactive window on the screen.

Graphical user Interfaces II
First Application

The previous presentation on GUIs (“gooey”) summarized the way Event Driven Programming is applied to Graphical User Interface Development

This presentation starts to describe how to implement all this in practice with JavaFX



Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

The Basic Structure of a JavaFX Program

The `javafx.application.Application` *class defines the essential framework for writing JavaFX programs.*

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MyJavaFX extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place a button in the scene
    Button btOK = new Button("OK");
    Scene scene = new Scene(btOK, 200, 250);
    primaryStage.setTitle("MyJavaFX"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    Application.launch(args);
  }
}
```

extend Application

override start

create a button

create a scene

set stage title

set a scene

display stage

main method

launch application

Life of a javafx application

Step 1 - Create an instance of the Application class

The program class must extend Application

A JavaFX application derives from the Application class in the JavaFx package. It means that it automatically inherits standard attributes and methods.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MyJavaFX extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place a button in the scene
    Button btOK = new Button("OK");
    Scene scene = new Scene(btOK, 200, 250);
    primaryStage.setTitle("MyJavaFX"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    Application.launch(args);
  }
}
```

extend Application

override start

create a button

create a scene

set stage title

set a scene

display stage

main method

launch application

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Does nothing by default

JavaFx will automatically call a function called init(). By default, this function does nothing. You can write your own version, and connect to a network or a database, or read a parameter file.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                          extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                            override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                                   create a button
11     Scene scene = new Scene(btOK, 200, 250);                          create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title         set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage    set a scene
14     primaryStage.show(); // Display the stage                        display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                           main method
22     Application.launch(args);                                         launch application
23   }
24 }
```

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Must be rewritten!

Step 3 - Call the start(javafx.stage.Start) method

What you must write (and override) is a function called "start()" that takes a "Stage" (the name given to windows in JavaFx) as parameter. The function adds the widgets to the window and defines how it looks, and how widgets will react.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");
11     Scene scene = new Scene(btOK, 200, 250);
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title
13     primaryStage.setScene(scene); // Place the scene in the stage
14     primaryStage.show(); // Display the stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {
22     Application.launch(args);
23   }
24 }
```

extend Application

override start

create a button

create a scene

set stage title

set a scene

display stage

main method

launch application

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Step 3 - Call the start(javafx.stage.Stage) method

A simple JavaFX displays a button in the window.

Step 4 - Wait for the application to finish:
the application calls Platform.exit()
or window closed

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {                    extend Application
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {                      override start
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");                            create a button
11     Scene scene = new Scene(btOK, 200, 250);                   create a scene
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title  set stage title
13     primaryStage.setScene(scene); // Place the scene in the stage  set a scene
14     primaryStage.show(); // Display the stage                 display stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {                     main method
22     Application.launch(args);                                  launch application
23   }
24 }
```

You must write the event handlers you need – *and nothing else* – JavaFx will run the application until it calls an exit routine (perhaps associated with a "Quit" button) or it receives the event "Window destroyed".

Life of a javafx application

Step 1 - Create an instance of the Application class

Step 2 - Call the init() method

Step 3 - Call the start(javafx.stage.Stage) method

Step 4 - Wait for the application to finish: the application calls Platform.exit() or window closed

Step 5 - Call the stop() method

It will then call a stop() method where you can undo what you have done in init() – for example disconnect from a database or network. Like with init(), and exit() rewriting stop() is only done if you *need* to.

A simple JavaFX displays a button in the window.

MyJavaFX.java

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Button;
4  import javafx.stage.Stage;
5
6  public class MyJavaFX extends Application {
7    @Override // Override the start method in the Application class
8    public void start(Stage primaryStage) {
9      // Create a scene and place a button in the scene
10     Button btOK = new Button("OK");
11     Scene scene = new Scene(btOK, 200, 250);
12     primaryStage.setTitle("MyJavaFX"); // Set the stage title
13     primaryStage.setScene(scene); // Place the scene in the stage
14     primaryStage.show(); // Display the stage
15   }
16
17   /**
18    * The main method is only needed for the IDE with limited
19    * JavaFX support. Not needed for running from the command line.
20    */
21   public static void main(String[] args) {
22     Application.launch(args);
23   }
24 }
```

extend Application

override start

create a button

create a scene

set stage title

set a scene

display stage

main method

launch application

NOTICE

No explicit test

No explicit loop

<u>Just events</u>

In a GUI application is that you just declare everything, and there is no procedural logic (if ... and loops) outside event handlers.