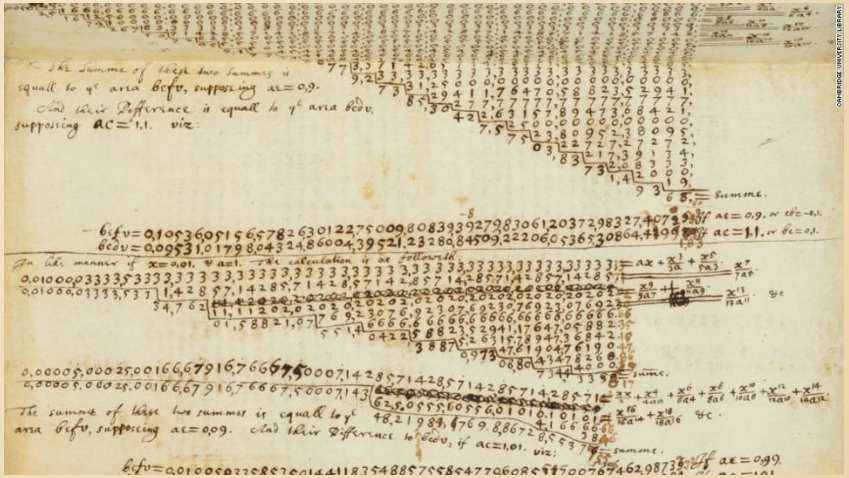
Lecture 3-1

Approximation Algorithms



Approximation algorithms

Example: Compute $\sqrt{x}$ for $x > 1$

Strategy

Ideally: Find g such that $g \times g = x$.

Instead: we'll find an approximate value which is *close enough* to the correct value.

Specifically, we'll search for a number g such that, for a given small *epsilon*,

$| g \times g - x | \leq epsilon$

Example: Compute $\sqrt{169}$ for *epsilon* = 0.01

We use some algorithm and find $g = 12.9999$

Since $|12.9999 \times 12.9999 - 169| \leq 0.01$,

we say that $g = 12.9999$ is a ***good enough approximation*** of $\sqrt{169}$

```
// Computing √x
epsilon = 0.01
g = initial guess
while | g × g – x | > epsilon {
    g = improve(g)
}
return g
```

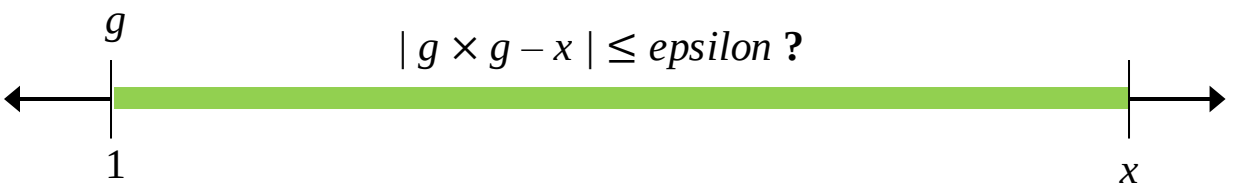
Relevant algorithms

- Sequential search
- Bisection
- Newton - Raphson

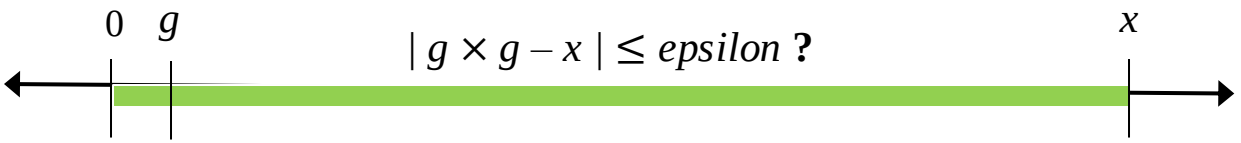
Sequential search ("brute force")

Task: Compute $\sqrt{x}$ for $x > 1$

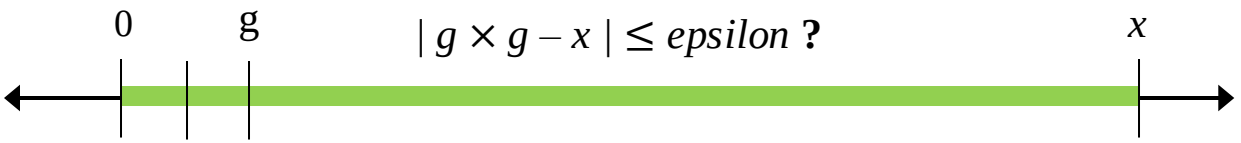
Itr. 0: The square root of x must be between 1 and x, so let's start with $g = 1$:



Itr. 1: g = g + *increment* (*increment* = a small number, like 0.001)



Itr. 2: g = g + *increment*



Continue until $| g \times g - x | \leq epsilon$

Performance

- The smaller is *epsilon*, the more accurate is the answer
- The smaller is *increment*, the slower is the computation
- **Running time:** Less than $x / increment$ iterations.

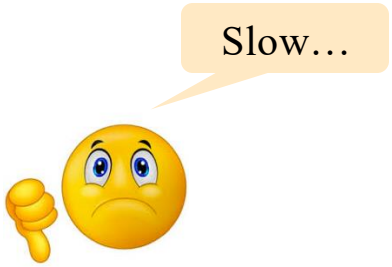
Sequential search ("brute force")

```
// Computes an approximate square root by sequential search
public class Sqrt1 {
   public static void main(String args[]) {
      double x = 16; // we'll find the (approx.) square root of x
      double epsilon = 0.01, increment = 0.0001;
      double g = 1.0;
      int stepCounter = 0;
      while (Math.abs(g * g - x) >= epsilon) {
         g += increment;
         stepCounter++;
      }
      System.out.println("Square root (approx.) = " + g);
      System.out.println("Number of iterations = " + stepCounter);
   }
}
```

```
% java Sqrt1 (x = 16)
Square root (approx.) = 3.998800000004108
Number of iterations = 29988
```

Sequential search ("brute force")

```
// Computes an approximate square root by sequential search
public class Sqrt1 {
   public static void main(String args[]) {
      double x = 16; // we'll find the (approx.) square root of x
      double epsilon = 0.01, increment = 0.0001;
      double g = 1.0;
      int stepCounter = 0;
      while (Math.abs(g * g - x) >= epsilon) {
         g += increment;
         stepCounter++;
      }
      System.out.println("Square root (approx.) = " + g);
      System.out.println("Number of iterations = " + stepCounter);
   }
}
```



Bug (nasty)

If *increment* is too large, we can miss the answer, and get into an infinite loop

```
% java Sqrt1 (x = 16)
Square root (approx.) = 3.998800000004108
Number of iterations = 29988

% java Sqrt1 (x = 100)
Square root (approx.) = 9.999499999990128
Number of iterations = 89995

% java Sqrt1 (x = 105)
Square root (approx.) = 10.246499999989553
Number of iterations = 92465
```

Sequential search ("brute force")

```
// Computes an approximate square root by sequential search
public class Sqrt1 {
   public static void main(String args[]) {
      double x = 16; // we'll find the (approx.) square root of x
      double epsilon = 0.01, increment = 0.0001;
      double g = 1.0;
      int stepCounter = 0;
      while ((Math.abs(g * g - x) >= epsilon) && (g <= x)) {
         g += increment;
         stepCounter++;
      }

      if (g > x) {
          System.out.println("Use a smaller increment");
      } else {
          System.out.println("Square root (approx.) = " + g);
      }
   }
}
```

Bug fixed

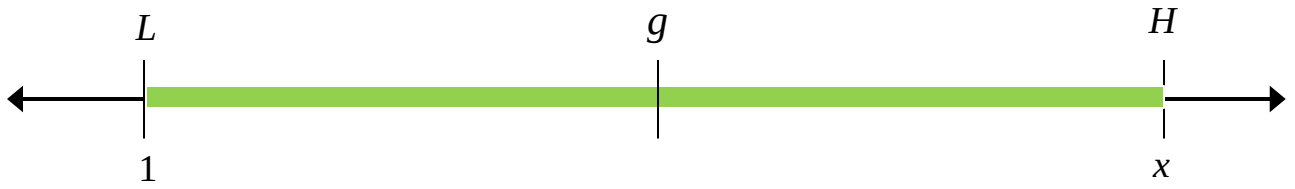
Algorithms

- Sequential search
- Bisection
- Newton - Raphson

Bi-section search

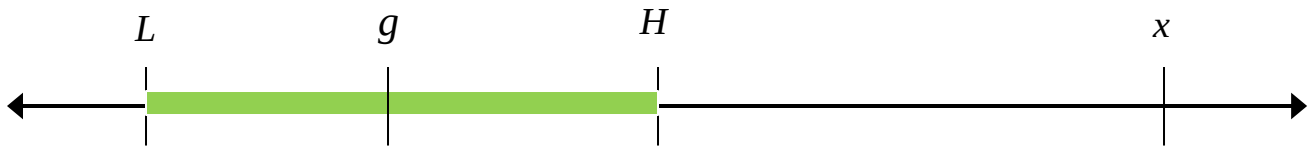
Task: Compute $\sqrt{x}$ for $x > 1$

Itr. 0: Initialize $L = 1$, $H = x$, g = $(L + H) / 2$:



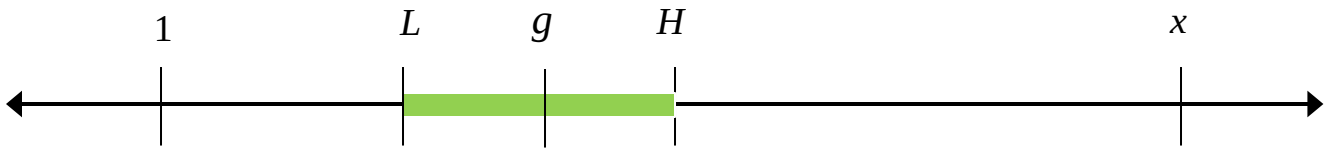
Itr. 1: Suppose that g × g > x. Therefore, the new g must be between L and g.

$H = g$, $g = (L + H) / 2$:



Itr. 2: Suppose that g × g < x. Therefore, the new g must be between g and H.

$L = g$, $g = (L + H) / 2$:

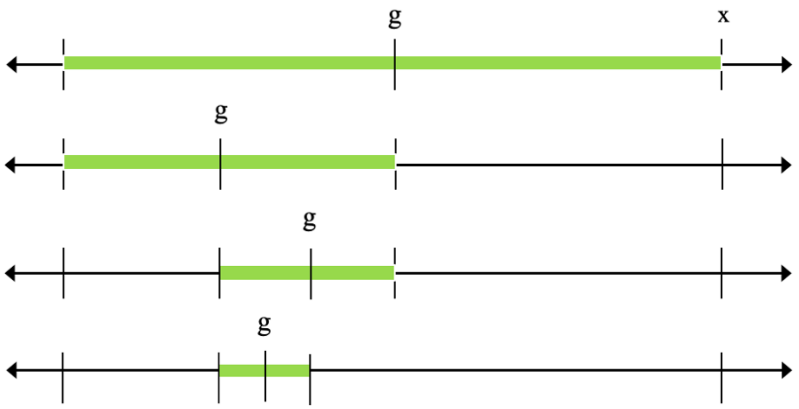


Continue: Until | g × g – x | ≤ *epsilon*

Bi-section search

Performance

- In itr. 0, search space: x
- In itr. 1, search space: $x / 2$
- In itr. 2, search space: $x / 4$
- In itr. 3, search space: $x / 8$
- In itr. k, search space: $x / 2^k$



Running time

In each step, the search space is divided by half;

Therefore, g converges to $\sqrt{x}$ after an order of $log_2 x$ iterations;

(more about running time analysis and logarithmic running time later in the course).

Bi-section search

```
// Computes an approximate square root by bi-section search
public class Sqrt2 {
   public static void main(String args[]) {
      double x = 16;
```

```
% java Sqrt2 (x = 16)
Square root (approx.) = 3.999267578125
Number of iterations = 11

% java Sqrt2 (x = 105)
Square root (approx.) = 10.2469482421875
Number of iterations = 15
```

Bi-section search

```
// Computes an approximate square root by bi-section search
public class Sqrt2 {
   public static void main(String args[]) {
      double x = 16;
      double epsilon = 0.01, L = 1.0, H = x;
      double g = (L + H) / 2.0;
      int stepCounter = 0;
      while (Math.abs(g * g - x) >= epsilon) {
         if (g * g < x)
            L = g;
         else
            H = g;
         g = (L + H) / 2;
         stepCounter++;
      }
      System.out.println("Square root (approx.) = " + g);
      System.out.println("Number of iterations = " + stepCounter);
   }
}
```

Fast!

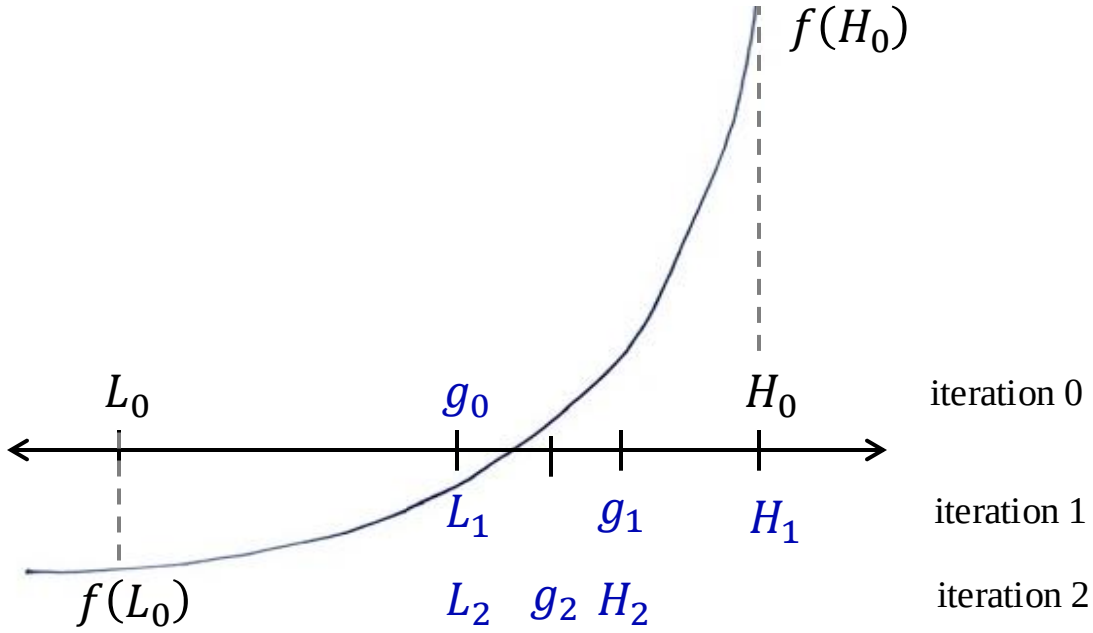
```
% java Sqrt2 (x = 16)
Square root (approx.) = 3.999267578125
Number of iterations = 11

% java Sqrt2 (x = 105)
Square root (approx.) = 10.2469482421875
Number of iterations = 15
```

Bi-section search: Calculus intuition

Computing, say, $\sqrt{17}$, is like finding the root of the function $f(x) = x^2 - 17 = 0$

Intermediate Value Theorem (IVT): If f is monotonic and $f(L) \cdot f(H) < 0$,
the root of f is between L and H.



Bisection:

Start with $g_0 = (L_0 + H_0) / 2$

In each iteration i: if $f(g_{i-1}) \cdot f(H_{i-1}) < 0$ set $L_i = g_{i-1}$, $H_i = H_{i-1}$, $g_i = (L_i + H_i) / 2$

else set $L_i = L_{i-1}$, $H_i = g_{i-1}$, $g_i = (L_i + H_i) / 2$

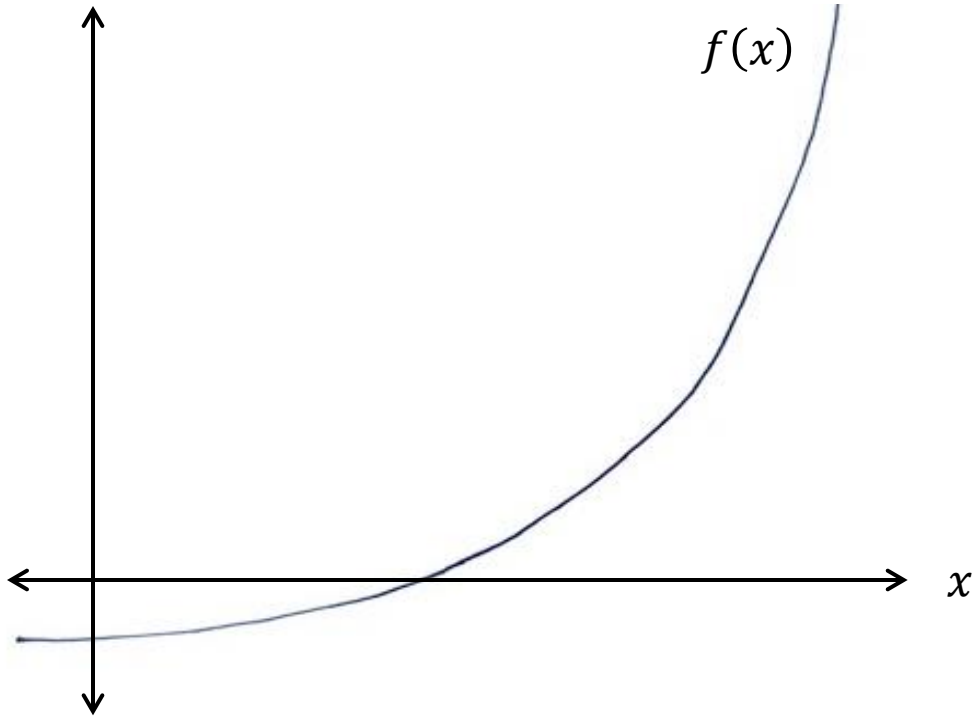
Algorithms

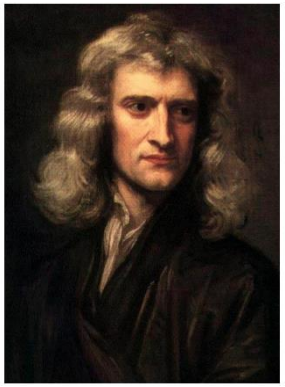
- Sequential search
- Bisection
- Newton - Raphson

Newton–Raphson

The Newton-Raphson discovery

If $f(x)$ is a continuous and differentiable function, we can approach $f(g) = 0$ as follows:



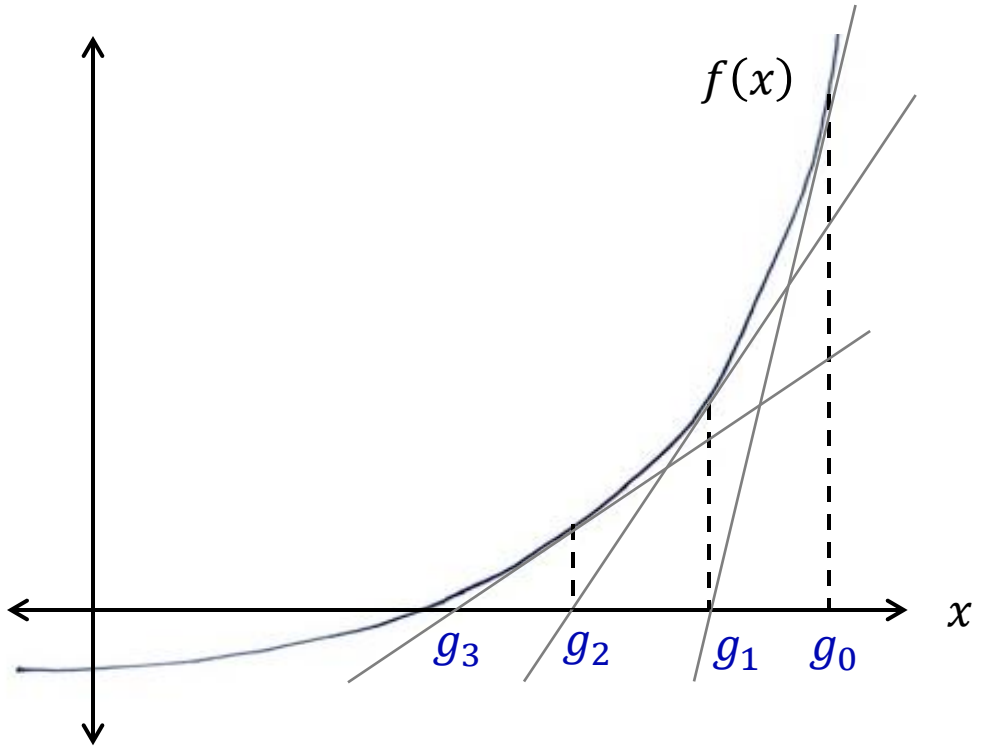


Isaac Newton
(1643 - 1727)

Newton–Raphson

The Newton-Raphson discovery

If $f(x)$ is a continuous and differentiable function, we can approach $f(g) = 0$ as follows:



Formally: If g is an approximate solution of $f(x) = 0$, then $g - \frac{f(g)}{f'(g)}$ is a better solution.

These insights led to the discovery of such concepts as *derivative* (נגזרת), *tangent* (שיפוע), *convergence* (התכנסות), *limit* (גבול), ..., and the invention / discovery of modern Calculus.

Newton–Raphson

Example: find $x = \sqrt{719}$

Same as solving $f(x) = x^2 - 719 = 0$

Newton-Raphson

If g is an approximate solution of $f(x) = x^2 - 719$, then $g - \frac{f(g)}{f'(g)}$ is a better solution.

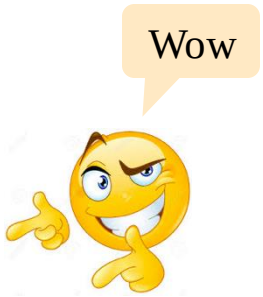
Algorithm

```
// Computing √719
epsilon = 0.01
g = initial guess
while | g × g – x | >          epsilon {
    g ← g – (g² – 719) / (2 · g)
}
return g
```

Newton–Raphson

```
// Uses Newton-Raphson search to approximate sqrt(x).
public class Sqrt3 {
   public static void main(String args[]) {
      double x = 16;
      double epsilon = 0.01;
      int stepCounter = 0;
      double g = x / 2;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
         System.out.println("Step " + stepCounter + ": g = " + g);
         stepCounter++;
      }
      System.out.println("Square root (approx.) = " + g);
   }
}
```

```
% java Sqrt3 (x = 16)
Step 0: g = 5.0
Step 1: g = 4.1
Step 2: g = 4.001219512195122
Square root (approx.) = 4.001219512195122
```




```
% java Sqrt3 (x = 105)
Step 0: g = 27.25
Step 1: g = 15.551605504587156
Step 2: g = 11.151659989960418
Step 3: g = 10.283649283522411
Step 4: g = 10.247016247636102
Square root (approx.) = 10.247016247636102
```

Approximation algorithms

Task: Find x so that $f(x) = 0$

Sequential search

- For any function
- Linear running time

Bi-section search

- For monotonic functions
- Logarithmic running time

Newton-Raphson

- For continuous and differentiable functions
- Fastest running time

Newton–Raphson: proof (informal)

The equation for the tangent at x_n is $L(x) = f'(x_n)x + b$,
where b is the intersection of the tangent with the y-axis.
To find b we set $x = x_n$ which should give $L(x_n) = f(x_n)$,

so:

$$f'(x_n)x_n + b = f(x_n) \Rightarrow b = f(x_n) - f'(x_n)x_n$$

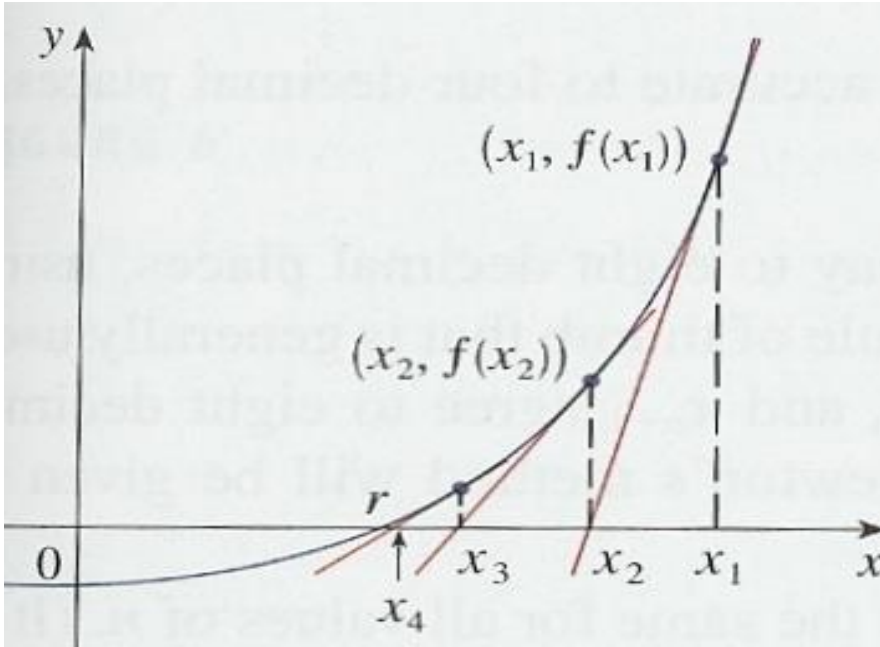
Therefore

$$L(x) = f'(x_n)(x - x_n) + f(x_n)$$

Now we look for x_{n+1} such that $L(x_{n+1}) = 0$:

$$0 = f'(x_n)(x_{n+1} - x_n) + f(x_n) \Rightarrow$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

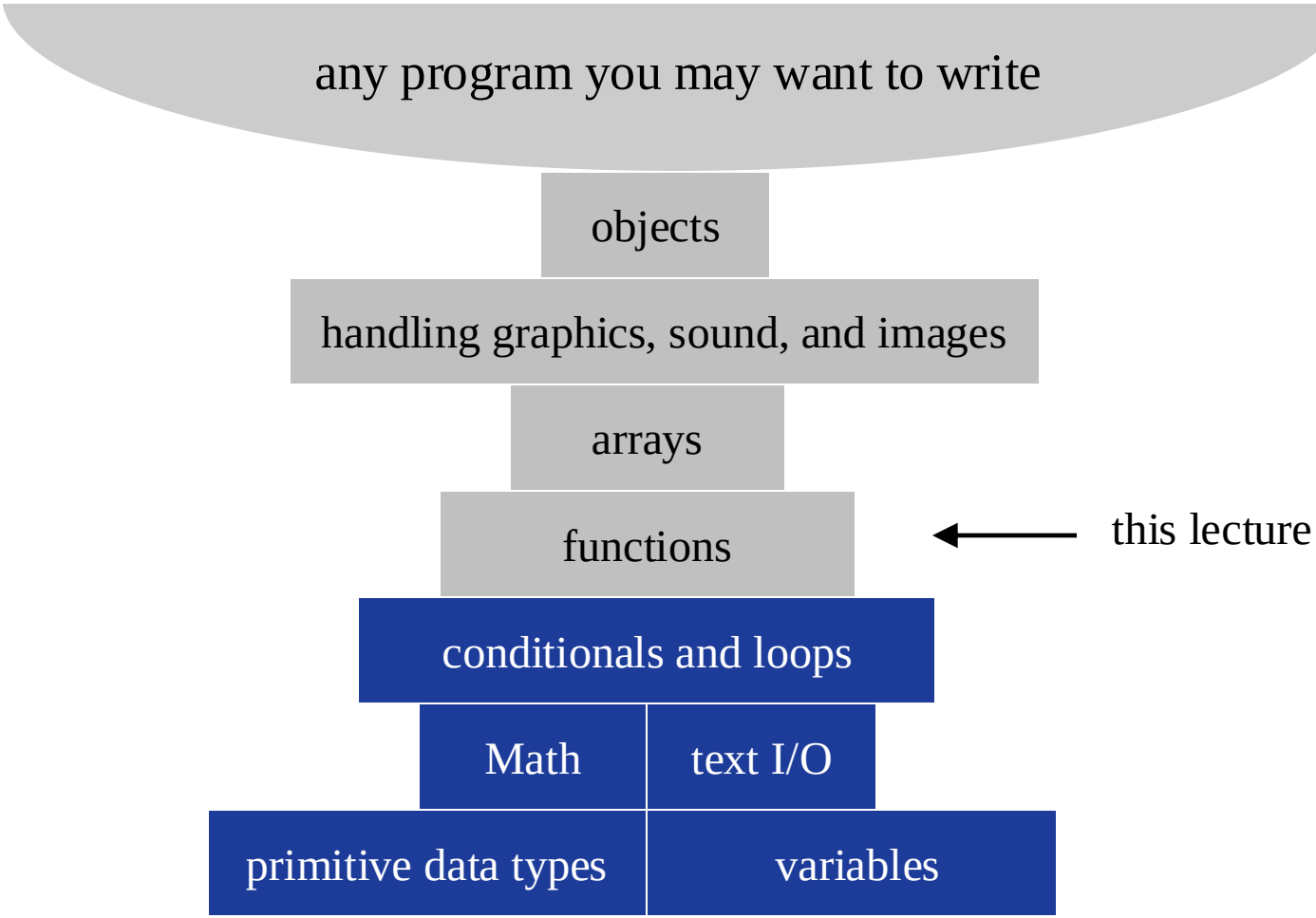


Lecture 3-1

Functions

(Functions in programming are related to,
but quite different from, mathematical functions)

The big picture



Overview

```
public class Sqrt3 {
   public static void main(String args[]) {
      double x = Double.parseDouble(args[0]);
      double epsilon = 0.01;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      System.out.println("Square root (approx.) = " + g);
   }
}
```

Question

How can we make this computation accessible to *any program* that needs to compute square roots?

Answer

- We can turn this computation into a reusable, stand-alone, *function.*
- We'll put this function, along with other mathematical functions, in a library named, say, `MyMath`

Overview

```
public class MyMath {
   public static void main(String args[]) {
      ...
      System.out.println(sqrt(x));
   }

   // Computes sqrt(x)
   public static double sqrt(double x) {
      double epsilon = 0.01;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
   // Other library functions come here
}
```

function call

function declaration

return value

Any other class

```
public class Foo {
   public static void main(...) {
      ...
      a = MyMath.sqrt(719);
      ...
   }
}
```

function call

- A Java program normally consists of several classes
- A *public method* can be called by any class
- In a class like `MyMath`, which serves as a library of functions, a `main` method is not required
- But, library classes often feature a `main` method designed to test / illustrate the library's functions.

Overview

```
public class MyMath {

    /** Computes square root with precision epsilon */
    public static double sqrt(double x, double epsilon) {
        if (x < 0) return Double.NaN;
        double g = x;
        while (Math.abs(g * g - x) > epsilon) {
            g = g - (g * g - x) / (2 * g);
        }
        return g;
    }
    // More MyMath functions
}
```

This version of `sqrt` takes `epsilon` as a parameter

Overview

```
public class MyMath {
   public static void main(String args[]) {
      // Tests the square root function with various precisions:
      System.out.println(Math.sqrt(2));    // benchmark value
      System.out.println(sqrt(2, 0.5));
      System.out.println(sqrt(2, 0.1));
      System.out.println(sqrt(2, 0.001));
      System.out.println(sqrt(2, 1e-7));  // 0.00000001
   }

   /** Computes square root with precision epsilon */
   public static double sqrt(double x, double epsilon) {
      if (x < 0) return Double.NaN;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
    // More MyMath functions
}
```

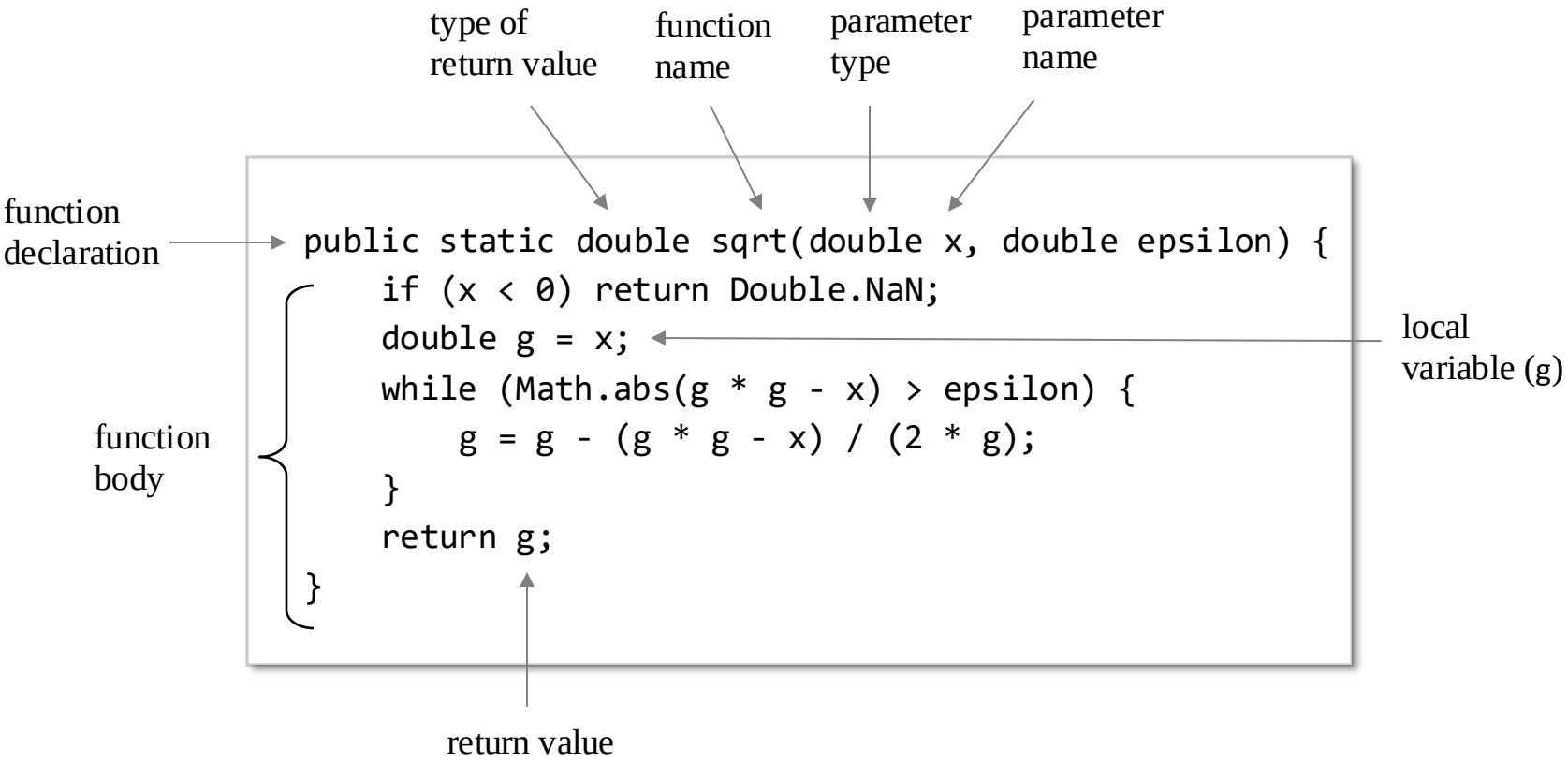
```
% java MyMath
1.4142135623730951
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
```

This version of `sqrt` takes `epsilon` as a parameter

Function anatomy

```
public static double sqrt(double x, double epsilon) {
    if (x < 0) return Double.NaN;
    double g = x;
    while (Math.abs(g * g - x) > epsilon) {
        g = g - (g * g - x) / (2 * g);
    }
    return g;
}
```

Function anatomy



A function

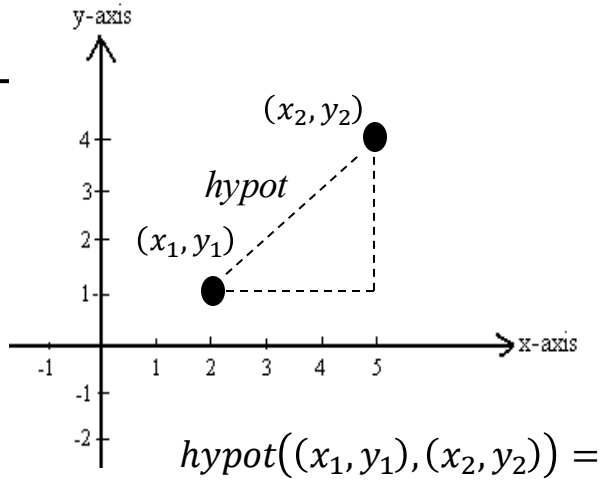
- Takes zero, one, or more inputs ("arguments")
- Returns a value, which may be `void`
- Has a *type*, which is the type of the returned value.

Function examples

```
public class MyMath {




    /** Computes the Euclidean distance between (x1,x2) and (y1,y1) */
    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }
}
```



$$hypot\big((x_1, y_1), (x_2, y_2)\big) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Euclidean distance

Widely used; Therefore, it makes sense to have a *function* that computes it

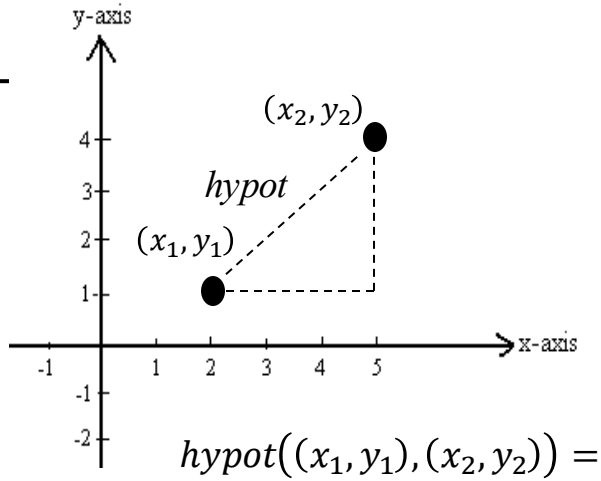
Function examples

```
public class MyMath {
    public static void main(String args[]) {
        /** Computes and prints a Euclidean distance */
        double d = hypot(2,1,5,4);
        System.out.println(d);
    }

    /** Computes the Euclidean distance between (x1,x2) and (y1,y1) */
    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }
    /** Computes sqrt(x) */
    public static double sqrt(double x) {
      double epsilon = 0.01;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
   ... // More functions come here
}
```

```
% java MyMath

4.242642472890547
```



$$hypot\big((x_1, y_1), (x_2, y_2)\big) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Euclidean distance

Widely used; Therefore, it makes sense to have a *function* that computes it

Function examples

```
public class MyMath {
    public static void main(String args[]) {
        /** Computes and prints a Euclidean distance */
        double d = hypot(2,1,5,4);
        System.out.println(d);
    }

    /** Computes the Euclidean distance between (x1,x2) and (y1,y1) */
    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }
    /** Computes sqrt(x) */
    public static double sqrt(double x) {
      double epsilon = 0.01;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
   ... // More functions come here
}
```

```
% java MyMath

4.242642472890547
```

Observations

- Class: A collection of one or more methods
- This particular class: A library of common math functions
- The `main` method is often used to test/demo the other methods in the class.

Functions


Function anatomy


Function call and return

- Overloading
- Software engineering issues

Calling a function

```
public class MyMath {
    public static void main(String args[]) {
        ...
        double d = hypot(2,1,5,4);
        ...
    }

    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }

    public static double sqrt(double x) {
        double epsilon = 0.01;
        double g = x;
        while (Math.abs(g * g - x) > epsilon) {
            g = g - (g * g - x) / (2 * g);
        }
        return g;
    }
    ...
}
```

main calls hypot

hypot calls sqrt

sqrt calls Math.abs

Terminology

The calling function = *caller*

The called function = *callee*

Calling a function:

- The caller passes arguments to the callee
- The caller suspends its execution, and waits for the callee to terminate
- Control transfers to the callee, which starts executing.

Calling a function

```
public class MyMath {
    public static void main(String args[]) {
        ...
        double d = hypot(2,1,5,4);
        ...
    }

    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }

    public static double sqrt(double x) {
        double epsilon = 0.01;
        double g = x;
        while (Math.abs(g * g - x) > epsilon) {
            g = g - (g * g - x) / (2 * g);
        }
        return g;
    }
    ...
}
```

x1, x2, y1, y2, are initialized to 2, 1, 5, 4, respectively

x is initialized to the value of dx*dx + dy*dy

Terminology

Arguments:

The values that the caller passes

Parameters:

Similar to local variables; initialized by the arguments supplied by the caller.

Returning from a function call

```
public class MyMath {
    public static void main(String args[]) {
        ...
        double d = hypot(2,1,5,4);
        ...
    }

    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }

    public static double sqrt(double x) {
        double epsilon = 0.01;
        double g = x;
        while (Math.abs(g * g - x) > epsilon) {
            g = g - (g * g - x) / (2 * g);
        }
        return g;
    }

    ...
}
```

RETURN VALUE

RETURN VALUE

When the callee returns:

- The return value replaces the function call in the caller's code
- The caller continues its execution.

Function call and return: simulation

```
public class MyMath {
    public static void main(String args[]) {
        ...
        double d = hypot(2,1,5,4);
    6   system.Out.println(d);
    }

    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
    5   return sqrt(dx*dx + dy*dy);
    }

    public static double sqrt(double x) {
        double epsilon = 0.01;
        double g = x;
        while (Math.abs(g * g - x) > epsilon) {
            g = g - (g * g - x) / (2 * g);
        }
    4   return g;
    }
    ...
}
```

% java MyMath

4.242642472890547

1. `MyMath.main` starts running
2. `main` calls `hypot`
   - `main` suspends its execution
   - `hypot` starts running
3. `hypot` calls `sqrt`
   - `hypot` suspends its execution
   - `sqrt` starts running
4. `sqrt` returns
   - `hypot` resumes its execution
   - the next executed instruction:
     `return` (*the value that* `sqrt` *returned*)
5. `hypot` returns
   - `main` resumes its execution
   - the next executed instruction:
     `d =` (*the value that* `hypot` *returned*)
6. `MyMath.main` prints, and then returns; the program terminates.

Functions

- ✓ Function anatomy
- ✓ Function call and return
- ➡ Overloading
- Software engineering issues

Overloading

```
public class MyMath {
   public static void main(String args[]) {
      ...
      System.out.println(sqrt(2, 0.1));
      System.out.println(sqrt(2, 0.001));
      System.out.println(sqrt(2));
      ...
   }
   /** Square root of x with precision epsilon */
   public static double sqrt(double x, double epsilon) {
      if (x < 0) return Double.NaN;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
}
```

Huh?

```
% java MyMath
1.4166666666666665
1.4142156862745097
1.4142156862745097
```

Overloading

```
public class MyMath {
   public static void main(String args[]) {
      ...
      System.out.println(sqrt(2, 0.1));
      System.out.println(sqrt(2, 0.001));
      System.out.println(sqrt(2));
      ...
   }
   /** Square root of x with precision epsilon */
   public static double sqrt(double x, double epsilon) {
      if (x < 0) return Double.NaN;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
   /** Square root of x with default precision 0.001 */
   public static double sqrt(double x) {
      return sqrt(x, 0.001);
   }
}
```

Function signature

Consists of:

The *number* of the function's parameters;

The *types* of the function's parameters.

Overloading

Defining functions that have the same name, but different signatures

Note: These are *different functions*

Usage

- Overloading is *widely used*
- Supports many different needs

In this example: Overloading is used to implement a computation that has both a default precision, and a user-defined precision.

Overloading: Example 2

println()
Terminates the current line by writing the line separator string.

println(boolean x)
Prints a boolean and then terminate the line.

println(char x)
Prints a character and then terminate the line.

println(char[] x)
Prints an array of characters and then terminate the line.

println(double x)
Prints a double and then terminate the line.

println(float x)
Prints a float and then terminate the line.

println(int x)
Prints an integer and then terminate the line.

println(long x)
Prints a long and then terminate the line.

println(Object x)
Prints an Object and then terminate the line.

println(String x)
Prints a String and then terminate the line.

(Java's `println` function API)

Here overloading is used to create a printing service that can be used to print many different data types using "the same" `println` function.

Overloading: Example 3

indexOf(int ch)

Returns the index within this string of the first occurrence of the specified character.

indexOf(int ch, int fromIndex)

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

indexOf(**String** str)

Returns the index within this string of the first occurrence of the specified substring.

indexOf(**String** str, int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

(Java's `indexOf` function API)

```
str = "It was the best of time";
System.out.println(str.indexOf('t'));     // prints 1
System.out.println(str.indexOf('t',3));   // prints 7
System.out.println(str.indexOf("best"));  // prints 11
```

Here overloading is used to feature different variants of "the same" `indexOf` function.

Functions

- ✓ Function anatomy
- ✓ Function call and return
- ✓ Overloading
- ➡ Software engineering: Some topics
  - Modularity
  - API
  - Roles

Modularity

Modularity: Dividing a program into several modules

```
public class MyMath {
    public static void main(String args[]) {
        double d = hypot(2,1,5,4);
        System.out.println(d);
    }

    /** Euclidean distance between (x1,x2) and (y1,y1) */
    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }

   // Computes sqrt(x)
   public static double sqrt(double x) {
      double epsilon = 0.01;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
   ...
}
```

Benefits

Reduces:

- code complexity
- redundant code (errors)

Enables:

- code reuse
- unit testing
- maintenance
- parallel development.

Modularity

Modularity: Dividing a program into several modules

```
public class MyMath {
    public static void main(String args[]) {
        double d = hypot(2,1,5,4);
        System.out.println(d);
    }

    /** Euclidean distance between (x1,x2) and (y1,y1) */
    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        double dx = x2 - x1;
        double dy = y2 - y1;
        return sqrt(dx*dx + dy*dy);
    }

   // Computes sqrt(x)
   public static double sqrt(double x) {
      double epsilon = 0.01;
      double g = x;
      while (Math.abs(g * g - x) > epsilon) {
         g = g - (g * g - x) / (2 * g);
      }
      return g;
   }
   ...
}
```

Methods in one class can call public methods in other classes:

```
public class Foo {
   ...
   System.out.println(MyMath.hypot(2,1,5,4));
   ...
   double q = MyMath.sqrt(54267);
   ...
}
```

Method calling syntax

- If the callee is in the same class: *functionName*(*argument list*)
- If the callee is in a different class: *ClassName*.*functionName*(*argument list*)

API

```
public class Foo {
    ...
    String s = str.substring(3,5);
    ...
}
```

Substring API:

substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.

The caller's perspective

When I call a function:

- I view and use the function as a black box abstraction
- I don't care how it is implemented
- I care only ***how to use it***
- All I need is the function's API

For example:
I use a search engine / chatbot to review the "`String class Java Oracle API`"

API: "Application Programming Interface":

- Documents the *function's interface*
- Explains *how to use the function*
- Focuses on the caller's perspective
- Written by the function's developer / architect.

Typical roles in software development

Class skeleton

```
public class MyMath {
    public static void main(String args[]) {
        double d = hypot(2,1,5,4);
        System.out.println(d);
    }

    /** Euclidean distance between (x1,x2) and (y1,y1) */
    public static double hypot(double x1, double y1,
                               double x2, double y2) {
        return 0;
    }

    // Computes sqrt(x)
    public static double sqrt(double x) {
        return 0;
   }
   ...
}
```

Some examples (approx. / initial product)

- Chrome: 100 developers
- Slack: 20
- Whatsapp: 30
- Minecraft: 1
- Linux: 1 architect, thousands developers

System architects

Write class skeletons:

- Method signatures
- API documentation
- Standard tests

Developers

Implement the functions, adds more tests, as needed

Testers

Test everything.