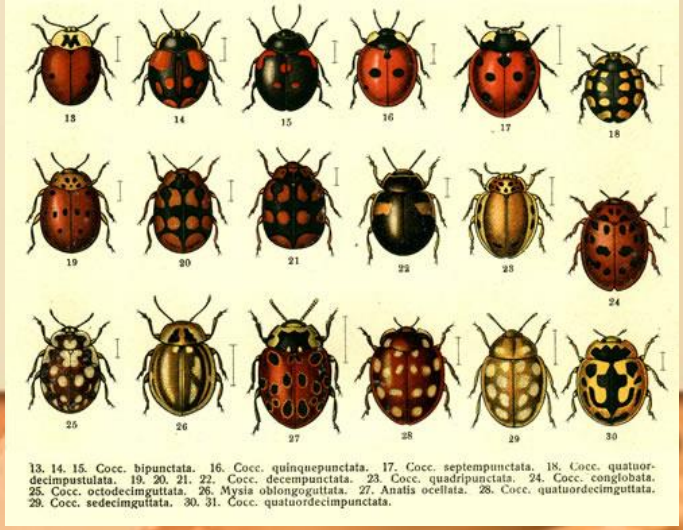
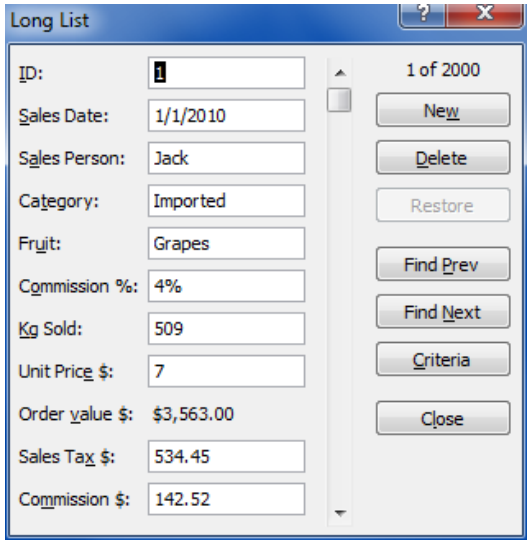
Lecture 1-2

Data Types



13. 14. 15. Cocc. bipunctata. 16. Cocc. quinquepunctata. 17. Cocc. septempunctata. 18. Cocc. quatuordecimpustulata. 19. 20. 21. 22. Cocc. decempunctata. 23. Cocc. quadripunctata. 24. Cocc. conglobata. 25. Cocc. octodecimguttata. 26. Mysia oblongoguttata. 27. Anatis ocellata. 28. Cocc. quatuordecimguttata. 29. Cocc. sedecimguttata. 30. 31. Cocc. quatuordecimpunctata.

Data




```
salesDate = "1/1/2010";
salesPerson = "Jack";
...
commission = 0.04;
kgSold = 509;
unitPrice = 7;
...
orderSent = true;
...
```

```
...
10100011001110011100110011001100
10101010010101010101001001010100
11101010010101001010101001010100
11110010100101010101001010010100
11101010010100101001010010100010
00000001001001000010100101010010
10100011001110011100110011001100
11101010010101001010101001010100
11110010100101010101001010010100
11101010010100101001010010100010
00000001001001000010100101010010
10100011001110011100110011001100
10101010010101010101001001010100
11110010100101010101001010010100
...
```

<u>User view</u>

An image on the screen

<u>Programmer view</u>

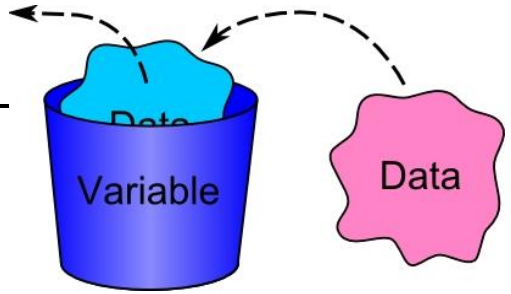
Data is manipulated using variables, statements, ...

<u>Computer view</u>

Data is represented as 0's and 1's (bits)

Different representations of the same data

Variables



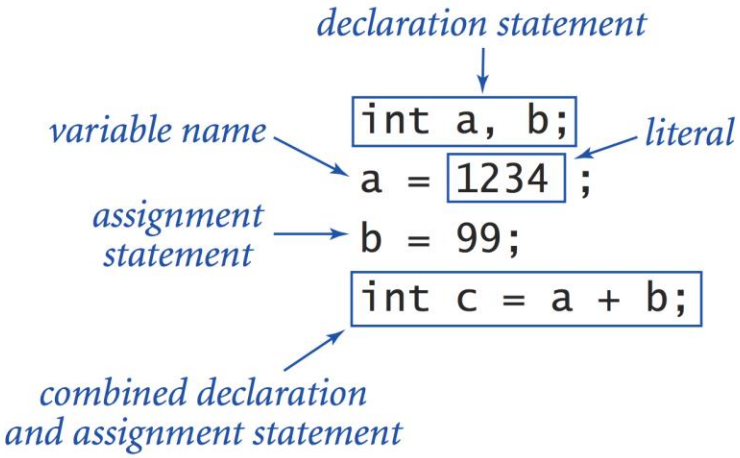
Variable: a container that has a *name*, a *type*, and a *value*

Examples:

```
int x = 17;                    // a variable of type int, now holding the value 17
String city = "Herzliya";      // a variable of type String, now holding the value "Herzliya"
double sum = 5012.35;          //  a variable of type double, now holding the value 5012.35
```

Code examples:

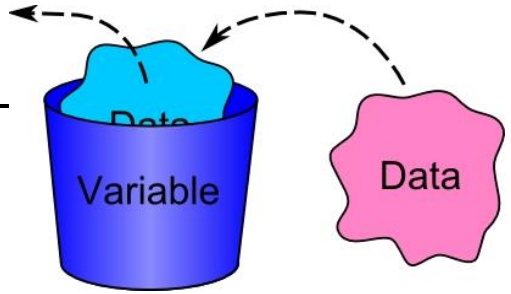
```
...
int a, b;
a = 1234;
b = 99;
int c = a + b;
...
```



Variable declaration: a statement that creates (and initializes) a variable

Assignment: a statement that assigns a value to a variable.

Data types



Variable: an abstract container that has a *name*, a *type*, and a *value*

Examples:

```
int x = 17;                    // a variable of type int, now holding the value 17
String city = "Herzliya";      // a variable of type String, now holding the value "Herzliya"
double sum = 5012.35;          // a variable of type double, now holding the value 5012.35
```

Code examples:

```
…
int a, b;
a = 1234;
b = 99;
int c = a + b;
…
```

Variable declaration: a statement that creates, and optionally initializes, a variable

Assignment: a statement that assigns a value to a variable.

Data types

Data type: Set of possible values, and possible operations

Primitive types

type	set of values	example values	typical operations
`int`	integer numbers	`17` `-5034`	add, subtract, multiply, divide
`double`	floating point numbers	`3.1415` `-171.19`	add, subtract, multiply, divide
`boolean`	truth values	`true` `false`	and, or, not
`char`	characters	`'c' 'K' '?' '5' '+'`	compare
`...`			

- Represent "scalar" / single values
- The Java language features eight primitive data types
- Built-into the language

Object types

`String`	sequences of characters	`"xckd"` `"hello world"`	concatenate
`Date`	dates	`03/11/2018`	equals, greater than, less than
`...`			

- Represent "aggregates" of values
- Java libraries feature many object types
- More object types can be created, as needed

Variables

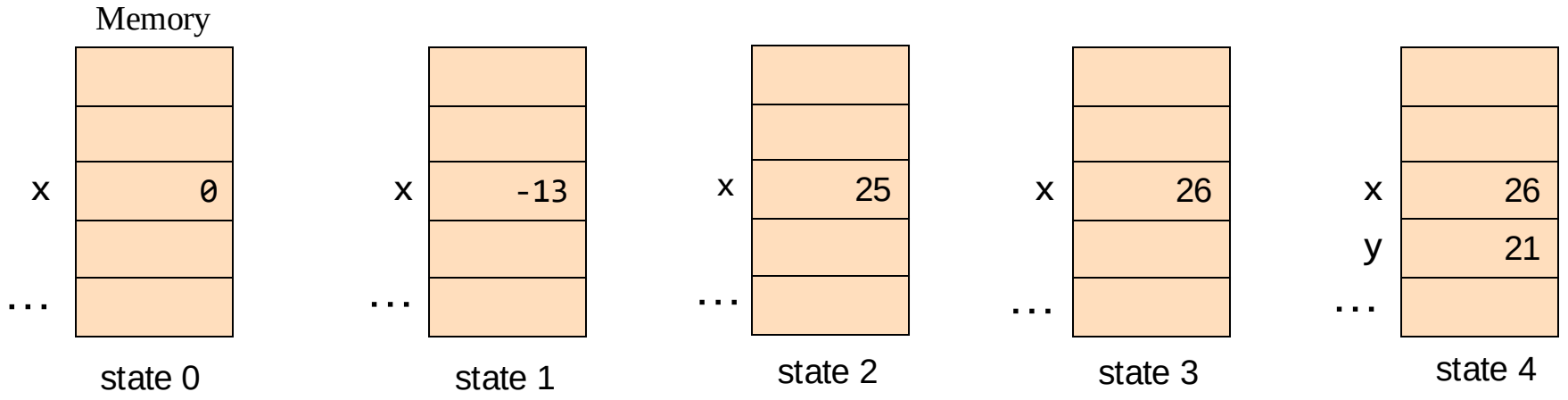
Abstraction

A container that has three properties: *name* (fixed), *type* (fixed), *value* (changing)

```
public class Demo1 {
  public static void main(String[] args) {
    int x;          // state 0
    x = -13;        // state 1
    x = 25;         // state 2
    x = x + 1;      // state 3
    int y = x - 5; // state 4
  }
}
```

Implementation

A physical memory location, assigned to represent the variable



Variables

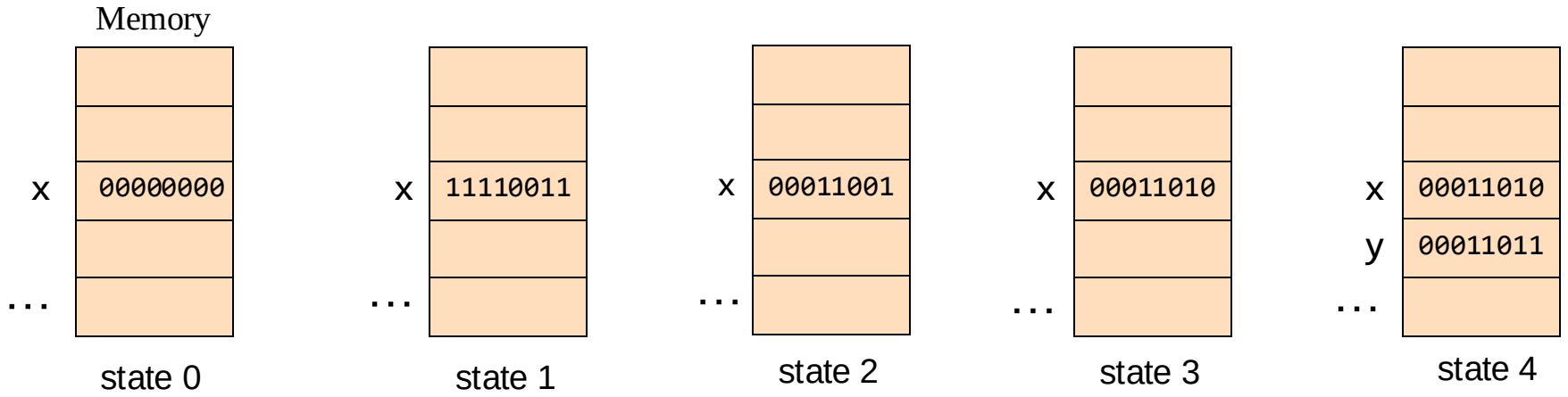
Abstraction

A container that has three properties: *name* (fixed), *type* (fixed), *value* (changing)

```
public class Demo1 {
  public static void main(String[] args) {
    int x;          // state 0
    x = -13;        // state 1
    x = 25;         // state 2
    x = x + 1;      // state 3
    int y = x - 5; // state 4
  }
}
```

Implementation (bits...)

A physical memory location, assigned to represent the variable, contains bits



Variable declaration

Examples:

```
int weight;
```

```
int x1, x2, n;
```

```
int z = x + y;
```

Syntax:

type varName;

type varName1, varName2 … ;

type varName = expression;

- Variables are declared (typed and and named) by the programmer, anywhere in the program, as needed
- Can only hold data of the declared type.

The assignment operation

Examples:

```
int x = 5, y = 9;
...
x = -3;
...
y = x;
...
x = y * y - 10 / x + 17;
...
int celsius, f;
f = 120;
celsius = (f - 32) * 5 / 9;
...
```

Syntax:

variableName = *expression*;

Assignment anatomy:

1. The expression on the right hand side is evaluated
2. The resulting value is assigned to the variable on the left hand side, overwriting its current value

(some languages use `let`, to make the syntax more explicit:
`let` *variableName* = *expression*;)

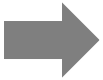
`f`: example of a badly named variable

Notes

- Variable names (and program readability) are critically important
- The assignment operator '=' has nothing to do with algebra's '='
- What happens if the type of the right hand side does not match the variable's type? More about this, later.

Lecture plan

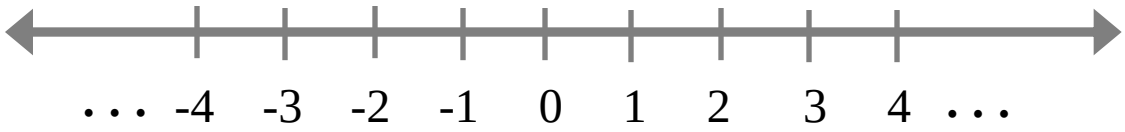
Variables



Representing integers: `int`

- Representing text: `String`
- Representing real numbers: `double`
- Representing logical values: `boolean`
- Casting (“data type conversions”)

Integer numbers



Java data types for integers

`byte:` (8 bits), stores integers from -128 to 127

`short:` (16 bits), stores integers from -32,768 to 32,767

`char:` (16 bits), stores integers from 0 to 65535 (used to represent characters, later)

`int:` (32 bits), stores integers from -2,147,483,648 to 2,147,483,647

`long:` (64 bits), stores integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

As a convention, in this course we'll represent integer numbers using `int`

Typical operations

`a + b` Addition

`a - b` Subtraction

`a * b` Multiplication

`a / b` Integer division

`a % b` Modulo (remainder)

Integer expressions

```
public class Demo2 {

   public static void main(String[] args) {

      System.out.println(5 + 3);  // 8

      System.out.println(5 - 3);  // 2

      System.out.println(5 * 3);  // 15

      System.out.println(5 / 5);  // 1

      System.out.println(5 % 3);  // 2

      System.out.println(1 / 0);  // Run-time error
```

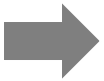
Integer expressions

```
public class Demo2 {

   public static void main(String[] args) {

      System.out.println(5 + 3);  // 8

      System.out.println(5 - 3);  // 2

      System.out.println(5 * 3);  // 15

      System.out.println(5 / 5);  // 1

      System.out.println(5 % 3);  // 2

      System.out.println(1 / 0);  // Run-time error

      System.out.println(3 * 5 - 2);    // 13  (* has precedence)

      System.out.println(3 + 5 / 2);    // 5   (/ has precedence)

      System.out.println(3 - 5 - 2);    // -4  (- associates to the left)

      System.out.println((3 - 5) - 2);  // -4  (better style)

      System.out.println(3 - (5 - 2));  // 0   (parentheses have precedence)

   }

}
```

Lecture plan


Variables


Representing integers: `int`


Representing text: `String`

- Representing real numbers: `double`
- Representing logical values: `boolean`
- Casting (“data type conversions”)

Textual information

Text:

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness. That to secure these rights, Governments are instituted among Men, ...

DNA:

HTML:

Java:

```
String textSegment =
       "We hold these truths …";

String dnaSegment = "ATGTTGCGGCTCGGGGCCAA ...";

String HTMLSegment =
       "<!DOCTYPE html> <html lang=\"en-US\"> ...";
```

Java represents a sequence of characters using a data type called `String`

Strings

Common operation: *concatenation*

```
public class Demo3 {

   public static void main(String[] args) {

      String s1 = "Tel";

      String s2 = "Aviv";

      System.out.println(s1 + s2);       // "TelAviv"

      System.out.println(s2 + s1);       // "AvivTel"

      System.out.println(s1 + " " + s2); // "Tel Aviv"

      System.out.println(s1 + 3);        // "Tel3"

      System.out.println(6 + 5);         // 11

      System.out.println("6" + "5");     // "65"

      System.out.println("6" + 5);       // "65"

      System.out.println(2 + " + " + 3 + " = " + (2 + 3));  // "2 + 3 = 5"

   }

}
```

When a String value is concatenated with a value of *any* other type:

- Java casts the other value as a String
- The type of the resulting expression is a String

Strings

```
// Illustrates "growing" a string
public class Demo4 {
   public static void main(String[] args) {
      String s = "1";          // "1"
      s = s + " 2 " + s;       // "1 2 1"
      s = s + " 3 " + s;       // "1 2 1 3 1 2 1"
      s = s + " 4 " + s;       // etc.
      System.out.println(s);
   }
}
```

```
% java Demo4
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

Command line arguments

```
public class Demo5 {
   public static void main(String[] args) {
      // Performs simple arithmetic operations on two command-line arguments
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      System.out.println(a + " + " + b + " = " + (a + b));
      System.out.println(a + " * " + b + " = " + (a * b));
      System.out.println(a + " / " + b + " = " + (a / b));
      System.out.println(a + " % " + b + " = " + (a % b));
   }
}
```

```
% javac Demo5.java

% java Demo5 5 3
5 + 3 = 8
5 * 3 = 15
5 / 3 = 1
5 % 3 = 2

% java Demo5 12 4
12 + 4 = 16
12 * 4 = 48
12 / 4 = 3
12 % 4 = 0
```

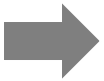
- Inputs can be entered into a program as *command-line arguments*
- Convention: The command-line arguments are named `args[0]`, `args[1]`, `args[2]`, ...
- The entered values (say, `12, 4`) are treated as Strings (`"12","4"`).
- `Integer.parseInt(`*String*`)`: A function that returns a string of digits as an `int`, part of a Java class library named `Integer`.

Lecture plan


Variables

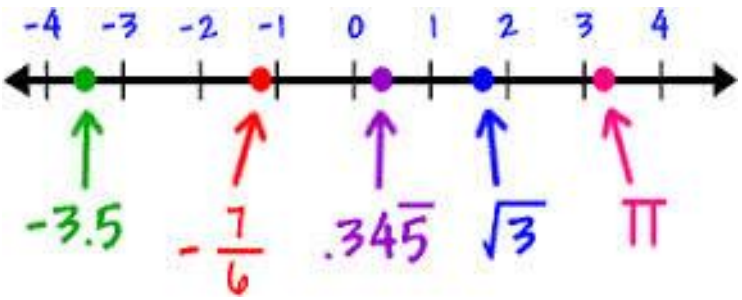

Representing integers: `int`


Representing text: `String`


Representing real numbers: `double`

- Representing logical values: `boolean`
- Casting (“data type conversions”)

Real Numbers



Java data types for approximating real numbers

`float:` Uses 32 bits,
represents values up to 7 digits after the decimal point

`double:` Uses 64 bits,
represents values up to 15 digits after the decimal point

As a convention, in this course we'll represent real numbers using `double`.

The `double` data type

```
3.141 + 0.03     // 3.171
3.141 - .03      // 3.111
6.02e23 / 2      // 3.01E23
5.0 / 2.0        // 2.5
5 / 2            // 2
5.0 / 2          // 2.5
5 / 2.0          // 2.5
1 / 3            // 0
1.0 / 3.0        // 0.3333333333333333
1.0 / 0.0        // Infinity
1 / 0            // Runtime error
Math.sqrt(2.0)   // 1.4142135623730951
Math.sqrt(-1.0)  // NaN (Not a Number)
```

Scientific notation

Shorthand notation for `3.01` × 10^{23}

Type of *x op y* (*op* being `+`, `-`, `*`, `/`):

If either *x* or *y* is `double`, the resulting type is `double`

Observations

- Computers represent real numbers using finite *numerals* like 3.141592653589793
- Therefore, most real numbers are not represented accurately
- For example, 1/3 is represented as 0.333333333333333
- Can lead to exotic bugs (discussed later).

The double data type

```
// Illustrates expressions that use double values.
public class Demo6 {
   public static void main(String[] args) {
    System.out.println(3.141 + .03);     // 3.171
    System.out.println(3.141 - .03);     // 3.111
    System.out.println(6.02e23 / 2);     // 3.01E23
    System.out.println(5.0 / 2.0);       // 2.5
    System.out.println(5 / 2);           // 2
    System.out.println(5.0 / 2);         // 2.5
    System.out.println(5 / 2.0);         // 2.5
    System.out.println(1 / 3);           // 0
    System.out.println(1.0 / 3.0);       // 0.3333333333333333
    System.out.println(1.0 / 0.0);       // Infinity
    System.out.println(1 / 0);           // Runtime error
    System.out.println(Math.sqrt(2.0));  // 1.4142135623730951
    System.out.println(Math.sqrt(-1.0)); // NaN
   }
}
```

(Same examples in executable form)

Example: Solve the quadratic equation $x^2 + bx + c = 0$

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

Inputs: `b` and `c` (assumption: the coefficient of x^2 is 1)

$x^2 - 3x + 2$

```
% java Quad1 -3.0 2.0
2.0
1.0
```

the two solutions

command-line arguments

$x^2 - x - 1$

```
% java Quad1 -1.0 -1.0
1.618033988749895
-0.6180339887498949
```

$x^2 + x + 1$

```
% java Quad1 1.0 1.0
NaN
NaN
```

not a number

invalid argument

```
% java Quad1 1.0 hello
java.lang.NumberFormatException: hello
```

missing argument

```
% java Quad1 1.0
java.lang.ArrayIndexOutOfBoundsException
```

Error messages can be cryptic;

But, they actually make sense, and one gets used to them.

Example: Solve the quadratic equation $x^2 + bx + c = 0$

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

Inputs: `b` and `c` (assumption: the coefficient of x^2 is 1)

```
public class Quad1 {
   // Computes the roots of the equation x^2 + bx + c = 0
   public static void main(String[] args) {
      // Gets and parses b and c from the command-line
      double b = Double.parseDouble(args[0]);
      double c = Double.parseDouble(args[1]);

      // Calculates the roots
      double discriminant = b * b - 4.0 * c;
      double d = Math.sqrt(discriminant);
      double root1 = (-b + d) / 2.0;
      double root2 = (-b - d) / 2.0;

      // Prints the roots
      System.out.println(root1);
      System.out.println(root2);
    }
}
```

Calling a function from a Java library named `Double`

Calling a function from a Java library named `Math`

Calling a function from a Java library named `System`

In any programming language, a lot of action is done by functions that are called from libraries. The libraries are classes that extend the basic language.

Java's Math library

Offers common math functions:

`double abs(double a)`	*absolute value of a*
`double max(double a, double b)`	*maximum of a and b*
`double min(double a, double b)`	*minimum of a and b*

Note 1: `abs()`, `max()`, *and* `min()` *are defined also for* `int`, `long`, *and* `float`.

`double sin(double theta)`	*sine function*
`double cos(double theta)`	*cosine function*
`double tan(double theta)`	*tangent function*

Note 2: Angles are expressed in radians. Use `toDegrees()` *and* `toRadians()` *to convert.*
Note 3: Use `asin()`, `acos()`, *and* `atan()` *for inverse functions.*

`double exp(double a)`	*exponential* (e^a)
`double log(double a)`	*natural log* ($\log_e a$, *or* $\ln a$)
`double pow(double a, double b)`	*raise a to the bth power* (a^b)
`long round(double a)`	*round to the nearest integer*
`double random()`	*random number in* [0, 1)
`double sqrt(double a)`	*square root of a*
`double E`	*value of e (constant)*
`double PI`	*value of π (constant)*

Math.random()

A function that returns a pseudo-random `double` value, distributed uniformly in the interval [0,1)

```
public class Demo7 {
    public static void main(String[] args) {
        // prints a random value in [0,1)
        double x = Math.random();
        System.out.println(x);

        // prints a random value in [0,2)
        System.out.println(2.0 * Math.random());

        // prints a random value in [-1,+1)
        System.out.println(2.0 * Math.random() - 1.0);
    }
}
```

```
% javac Demo7.java
% java Demo7
0.43628266723130604
1.86291944429295026
-0.62346234623466346
% java Demo7
0.63453463443333444
0.88666695959595995
0.773688444443443434
```

Each program run produces (with high likelihood...) different results.

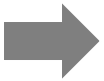
Lecture plan


Variables


Representing whole, signed numbers: `int`

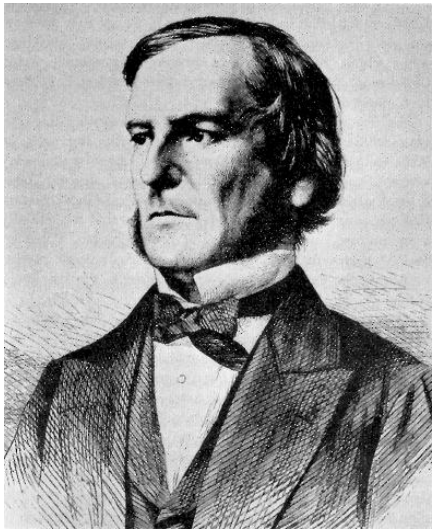

Representing text: `String`


Representing real numbers: `double`


Representing logical values: `boolean`

- Casting (“data type conversions”)

Booleans



George Boole
(1815-1864)

Described a 2-valued calculus now called "Boolean logic"

Booleans

Boolean logic is used for:

- Representing and evaluating logical expressions. Example: `(speed > 120)`
- Controlling program flow. Example:

```
if (speed > 120) {
     giveTicket(licenseNumber)
 }
```

The `boolean` data type

Two values only:

`true`

`false`

The boolean data type

Examples

```
...
String city = "";
int age = 0;

// Gets city and age data from the user (code omitted)
// Checks city
if (city == "Tel Aviv") System.out.println("you are paying too much rent");

// Checks age
if (age < 18) System.out.println("you are too young for this");

// Same effect (makes sense if we have to age-check many times):
boolean tooYoung = age < 18;
if (tooYoung) System.out.println("you are too young for this");

...

// Checks the parity of x
if ((x % 2) == 0)  System.out.println(x + " is even");
else               System.out.println(x + " is odd");
...
```

`if` and `if/else` statements: Will be discussed in the next lecture.

Boolean operators

Examples

```
(b * b - 4.0 * a * c) >= 0    // Nonnegative?
(year % 100) == 0             // Beginning of a century?
x != 1                        // Not equal 1?
```

Boolean expressions: *x op y* where:

x and *y* are compatible Java expressions and *op* is ==, !=, <, <=, >, >=

op	*meaning*	true	false
==	*equal*	2 == 2	2 == 3
!=	*not equal*	3 != 2	2 != 2
<	*less than*	2 < 13	2 < 2
<=	*less than or equal*	2 <= 2	3 <= 2
>	*greater than*	13 > 2	2 > 13
>=	*greater than or equal*	3 >= 2	2 >= 3

Boolean connectors (Not, And, Or)

`!` (not)

a	!a
false	true
true	false

Truth table: gives the output of a Boolean function (in this case, Not, And, Or) for every possible input

`&&` (and)

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

`||` (or)

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

In other words

`!a` is true iff `a` is false

`(a && b)` is true iff both `a` and `b` are true

`(a || b)` is true iff either `a` or `b` or both are true

Example: leap year

Task: determine if a given year is a leap year (שנה מעוברת)

Rule: yes if the year is (i) divisible by 400 or
(ii) divisible by 4 but not by 100

```
public class LeapYear {
   public static void main(String[] args) {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;

      // Checks if the year is divisible by 400
      isLeapYear = ((year % 400) == 0);

      // Then checks if the year is divisible by 4 but not by 100
      isLeapYear = isLeapYear || (((year % 4) == 0) && ((year % 100) != 0)));

      System.out.println(isLeapYear);
   }
}
```

```
% java LeapYear 2020
true

% java LeapYear 1900
false

% java LeapYear 2028
true
```

Example: leap year

Task: determine if a given year is a leap year (שנה מעוברת)

Rule: yes if the year is (i) divisible by 400 or
(ii) divisible by 4 but not by 100

```
public class LeapYear {
   public static void main(String[] args) {
      int year = Integer.parseInt(args[0]);
      boolean isLeapYear;

      // Checks if the year is divisible by 400
      isLeapYear = ((year % 400) == 0);

      // Then checks if the year is divisible by 4 but not by 100
      isLeapYear = isLeapYear || (((year % 4) == 0) && ((year % 100) != 0));

      System.out.println(isLeapYear);
   }
}
```

```
// Same as:
leapYear = ((year % 400) == 0) || ((year % 4) == 0) && ((year % 100) != 0));
```

Lecture plan

- ✓ Variables
- ✓ Representing whole, signed numbers: `int`
- ✓ Representing text: `String`
- ✓ Representing real numbers: `double`
- ✓ Representing logical values: `boolean`
- ➡ Casting (also known as “data type conversions”)

Casting

What happens when we create an expression made of different data types?

- In some cases Java handles the resulting type *implicitly* (example: `5.0 / 2`)
- In other cases the programmer must handle the resulting type *explicitly*

Implicit casting (examples)

```
"1234" + 99                 // "123499" (String)

11 * 0.3                    // 3.3 (double)
```

Explicit casting (examples)

```
Integer.parseInt("123")     // 123 (int)

(int) 2.71828               // 2 (int)

(int) 11 * 0.3              // 3.3 (double)

(11 * (int) 0.3)            // 0 (int)

(int) (11 * 0.3)            // 3 (int)
```

Casting

What happens when we create an expression made of different data types?

- In some cases Java handles the resulting type *implicitly* (example: `5.0 / 2`)
- In other cases the programmer must handle the resulting type *explicitly*

```
public class Demo8 {
   public static void main(String[] args) {

      System.out.println("1234" + 99);              // "123499" (String)

      System.out.println(11 * 0.3);                 // 3.3 (double)

      System.out.println(Integer.parseInt("123"));  // 123 (int)

      System.out.println((int) 2.71828);            // 2 (int)

      System.out.println((int) 11 * 0.3);           // 3.3 (double)

      System.out.println((11 * (int) 0.3));         // 0 (int)

      System.out.println((int) (11 * 0.3));         // 3 (int)
   }
}
```

(Same examples in executable form)

Casting

Example: generate a random integer in the range [0,N), where N is a command-line argument

Basic idea: Use `Math.random()` to create a random `double` value in [0,1),
then manipulate this value to create an `int` in the range [0,N).

```
public class RandomInt {
   // Generates a random integer from 0,...,N-1
   public static void main(String[] args) {
      // Gets N from the user
      int N = Integer.parseInt(args[0]);
      double r = Math.random();
      int n = (int) (r * N);

      System.out.println("random integer is " + n);
   }
}
```

Explicit casting: `args[0]`, which is a `String`, is cast as `int`, using a function call.

Explicit casting: Converts the value on the right to an `int` (ignoring the value after the decimal point)

Implicit casting: `double * int` gives `double`

Implicit casting: `String + int` gives `String`

```
% java RandomInt 10
random integer is 6

% java RandomInt 10
random integer is 2

% java RandomInt 35
random integer is 32

% java RandomInt 10000
random integer is 2184
```

Summary

Java data types (discussed in this lecture)

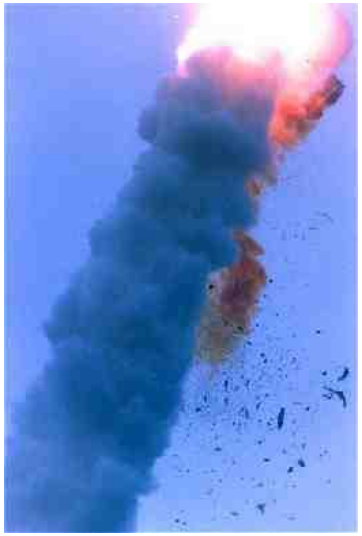
- `int`, `double` for mathematical calculations
- `String` for text processing
- `boolean` for logical expression and controlling program flow
- `...`

Java is a "strongly-typed" language

- Variables must be declared before they are used, and proper use is checked by the compiler
- When needed, type conversion is done either
  - *implicitly* (by the compiler), or
  - *explicitly* (by the programmer)

Watch out:

Programmers must be alert to potential exotic bugs resulting from implicit casting



Arianne 5 rocket crash (1996):

Cause: type cast error.

Overflow and rounding off errors

```
public class Demo9 {
   public static void main(String[] args) {
      System.out.println(100000 * 100000 * 100000);
      System.out.println(1.03 - 0.42);
      System.out.println(1.00 - 9 * .10);
      System.out.println((0.7 + 0.1) == (0.9 - 0.1));
   }
}
```

Expected results

```
% java Demo9
1000000000000000
0.61
0.1
true
```

Actual results

```
% java Demo9
-1530494976
0.6100000000000001
0.09999999999999998
false
```

What to do?

- **Be aware**
- Study the application's requirements
- If necessary, write code that handles rounding-off errors
- Test extreme values
- If exact values are important, there are solutions. But, these solutions may make the program less efficient.

Explanation: Computers represent numbers using a fixed number of bits (`0`'s and `1`'s). Results:

- Overflow / underflow
- Many real values are not represented accurately.