Introduction to Programming

Week 2

Magnus Madsen

Week 2: Basic Java Types

Monday

- Java Basics
- Strings
- Integers and Floating-point
- Booleans

Thursday

- Comparison operators
- Type conversions
- Overflow and underflow
- Live programming

Prologue

Quote of the week

"And programming computers was so fascinating. You create your own little universe, and then it does what you tell it to do."

— Vint Cerf

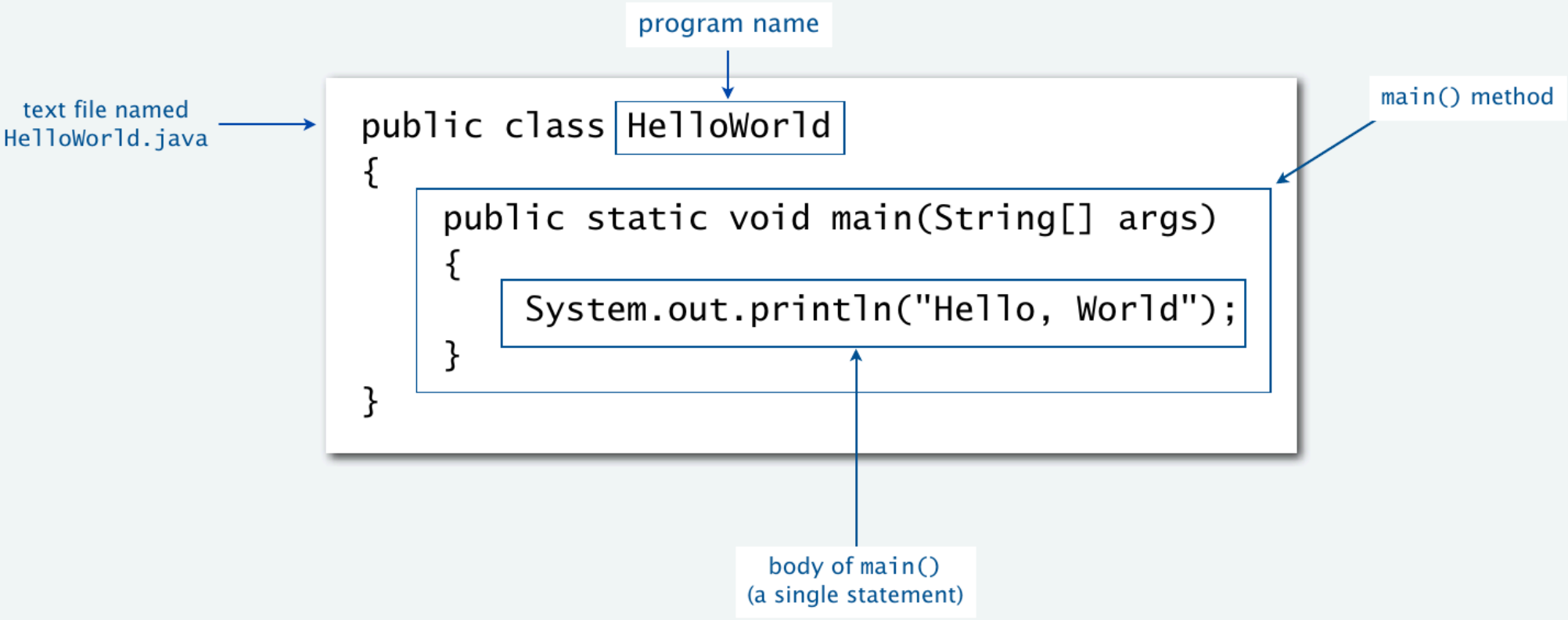
Epigram of the week

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine the wrong numbers, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

— Charles Babbage

Java Basics

Anatomy of your first program



A rich subset of the Java language vocabulary

built-in types
int
long
double
char
String
boolean

operations on numeric types
+
-
*
/
%
++
--

String *operations*
+
""
length()
charAt()
compareTo()
matches()

assignment
=

object oriented
static
class
public
private
new
final
toString()
main()

Math *methods*
Math.sin()
Math.cos()
Math.log()
Math.exp()
Math.pow()
Math.sqrt()
Math.min()
Math.max()
Math.abs()
Math.PI

System *methods*
System.print()
System.println()
System.printf()

punctuation
{
}
(
)
,
;

comparisons
<
<=
>
>=
==
!=

boolean operations
true
false
!
&&
||

flow control
if
else
for
while

arrays
a[]
length
new

type conversion methods
Integer.parseInt()
Double.parseDouble()

our Std methods
StdIn.read*()
StdOut.print*()
StdDraw.*()
StdAudio.*()
StdRandom.*()

Your programs will primarily consist of these plus identifiers (names) that you make up.

Q: What is the job of the Java compiler `javac`?

Compiler error (1/2)

The compiler will check the well-formedness of our programs.

For example, here we forgot the `public` keyword.

```
public class Main {
    static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

```
Error: Main method not found in class Main, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Compiler error (2/2)

Here, we forgot the `void` keyword.

```
public class Main {
    public static main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

```
Main.java:2: error: invalid method declaration; return type required
    public static main(String[] args) {
                  ^
1 error
```

A misformatted program

Formatting is important for *humans* reading a program, but the compiler does not care.

Java considers this program as **exactly the same thing** as the first *Hello World* program.

```
public
   class
Main { public static    void main(String[]    args) { System.out.println(
"Hello, World"
); } }
```

Program development in Java

is a three-step process, *with feedback*

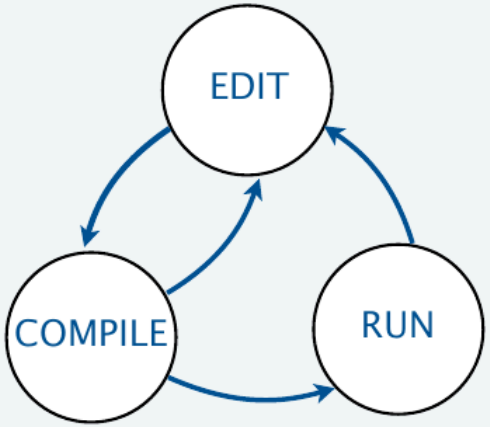
1. EDIT your program
   - Create it by typing on your computer's keyboard.
   - Result: a text file such as `HelloWorld.java`.

2. COMPILE it to create an executable file
   - Use the Java compiler
   - Result: a Java bytecode file such as `HelloWorld.class`
   - Mistake? Go back to 1. to fix and recompile.

not a legal Java program

3. RUN your program
   - Use the Java runtime.
   - Result: your program's output.
   - Mistake? Go back to 1. to fix, recompile, and run.

a legal Java program that does the wrong thing



Q: What is the difference between compile-time and run-time? How can you tell?

Data Types

A **data type** is a set of values and a set of operations on those values.

Built-in data types

Type	Values	Examples	Operations
`boolean`	truth values	`true` `false`	and, or, not
`char`	characters	`'A'` `'@'`	compare
`double`	floating-point numbers	`3.1415` `6.022e23`	add, subtract, multiply, divide
`int`	integers	`17` `12345`	add, subtract, multiply, divide
`String`	sequences of characters	`"Hello World"` `"CS is fun"`	concatenate

Type safety

Type safety prevents you from trying to do **illogical** things with your data.

We cannot:

- multiply booleans `true * 42`
- make an uppercase integer `3.toUpperCase()`
- …

In Java every variable has a type:

```java
int x = 123;
String y = "hello";
```

that controls which operations are permitted.

We can't put the square peg in the round hole.

Identifiers, Declarations, Statements, Expressions

Identifiers

An **identifier** is the **name** of a variable (or other program construct).

Java identifiers have special rules:

Identifier
`abc`
`abc123`
`aBcD`
`tragic`

Not Identifier	Why Not
`ab c`	includes a space
`123abc`	starts with a digit
`aB%#D`	includes special characters
`static`	reserved keyword

A *valid* identifier is not necessarily a good one.

`Ke$ha` is a valid identifier, but it's *unconventional* and annoying to type.

Most of your identifiers should be of the form: `<letters><optional digits>`

Naming conventions

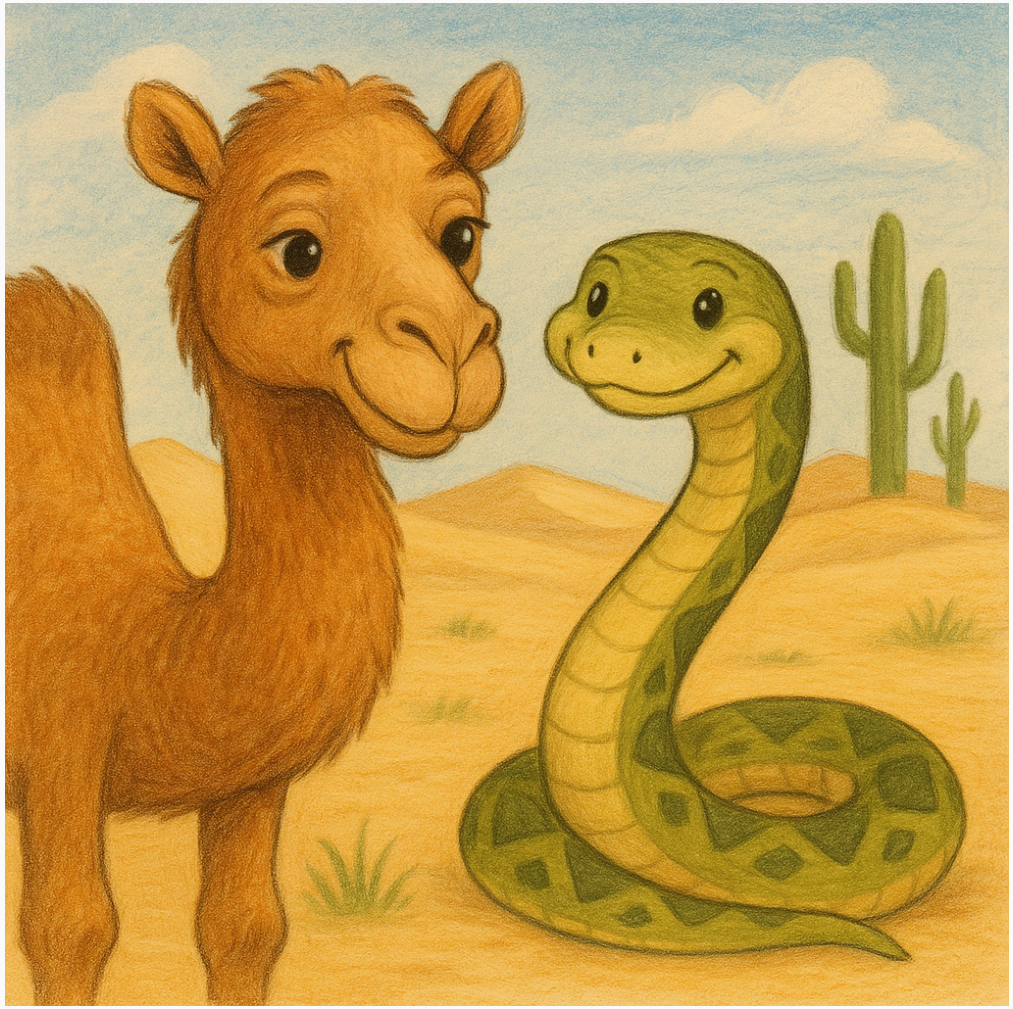
Most languages have **naming conventions**.

Java uses:

- 🐫 `lowerCamelCase` for most variables
  - `int daysInWeek = 7`
- 🐫 `UpperCamelCase` for classes
  - `ComputeTax`
- 🐍 `SCREAMING_SNAKE_CASE` for constants
  - `Math.PI`

Other languages use:

- 🐍 `snake_case`
- 🥙 `kebab-case`



Declarations, statements, expressions

In a **declaration**, we create a *new variable* and assign it a value.

```
String name = "Alice";
int year = 1987;
```

Declarations, statements, expressions

In a **declaration**, we create a *new variable* and assign it a value.

```
String name = "Alice";
int year = 1987;
```

In a **statement**, we take some *action*, like printing or assigning a value.

```
System.out.println("hello");
String country = "Denmark";
country = "Sweden";
```

Declarations, statements, expressions

In a **declaration**, we create a *new variable* and assign it a value.

```
String name = "Alice";
int year = 1987;
```

In a **statement**, we take some *action*, like printing or assigning a value.

```
System.out.println("hello");
String country = "Denmark";
country = "Sweden";
```

An **expression** evaluates to a *value*.

```
14 + 3                          // value: 17
"I have " + n + " pencils."  // value: "I have 22 pencils" (if n = 22)
```

Variables and assignment

Variables store values that can be used later in the program.

```java
public class CircleArea {
    public static void main(String[] args) {
        double radius = Double.parseDouble(args[0]);
        double pi = 3.14159;
        double area = pi * radius * radius;

        System.out.println("Radius: " + radius);
        System.out.println("Area: " + area);
    }
}
```

```
$ java CircleArea 5.0
```

```
Radius: 5.0
Area: 78.53975
```

Variables with reassignment

Variables can be updated with new values during program execution.

```java
public class Counter {
    public static void main(String[] args) {
        int count = 0;
        System.out.println("Initial count: " + count);

        count = count + 1;
        System.out.println("After increment: " + count);

        count = count * 2;
        System.out.println("After doubling: " + count);

        count = count - 1;
        System.out.println("Final count: " + count);
    }
}
```

```
$ java Counter
```

```
Initial count: 0
After increment: 1
After doubling: 2
Final count: 1
```

Example: swapping two variables

Swapping two variable values requires using a temporary third variable.

```java
public class Swap {
    public static void main(String[] args) {
        String word1 = args[0];
        String word2 = args[1];

        String tmp = word1;
        word1 = word2;
        word2 = tmp;

        System.out.println(word1 + " " + word2);
    }
}
```

```
$ java Swap apple banana
```

```
banana apple
```

Example: running total

We can reassign the value of variables in our program.

```
public class TotalThree {
    public static void main(String[] args) {
        int total = 0;
        total = total + Integer.parseInt(args[0]);
        System.out.println(total);
        total = total + Integer.parseInt(args[1]);
        System.out.println(total);
        total = total + Integer.parseInt(args[2]);
        System.out.println(total);
    }
}
```

```
$ java TotalThree 14 -11 1
```

```
14
3
4
```

Quiz: running total

What happens if we give **too many** arguments? `java TotalThree 123 2 -77 4`

What happens if we give **too few** arguments? `java TotalThree 93 47`

```
public class TotalThree {
    public static void main(String[] args) {
        int total = 0;
        total = total + Integer.parseInt(args[0]);
        System.out.println(total);
        total = total + Integer.parseInt(args[1]);
        System.out.println(total);
        total = total + Integer.parseInt(args[2]);
        System.out.println(total);
    }
}
```

Answer: running total

Extra arguments are **ignored**:

```
$ java TotalThree 11 3 -7 4
```

```
11
14
7
```

Missing arguments cause **errors**:

```
$ java TotalThree 93 47
```

```
93
140
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
Index 2 out of bounds for length 2
  at TotalThree.main(TotalThree.java:8)
```

Q:

What are the identifiers, declarations, statements, and expressions here?

```java
public class CircleArea {
    public static void main(String[] args) {
        double radius = Double.parseDouble(args[0]);
        double pi = 3.14159;
        double area = pi * radius * radius;

        System.out.println("Radius: " + radius);
        System.out.println("Area: " + area);
    }
}
```

Strings

Data type for computing with strings: `String`

`String` data type

values	sequences of characters
typical literals	`"Hello, "` `"1 "` `" * "`
operation	concatenate
operator	+

Examples of `String` operations (concatenation)

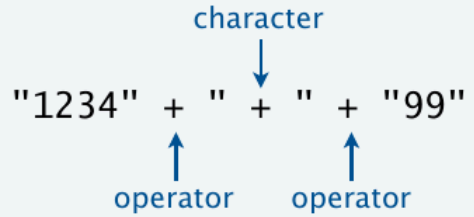
expression	*value*
`"Hi, " + "Bob"`	`"Hi, Bob"`
`"1" + " 2 " + "1"`	`"1 2 1"`
`"1234" + " + " + "99"`	`"1234 + 99"`
`"1234" + "99"`	`"123499"`

Typical use: Input and output.

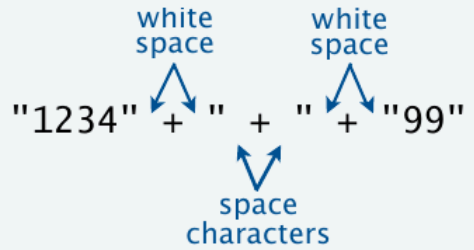
Important note:

Character interpretation depends on context!

Ex 1: plus signs



Ex 2: spaces



String example

```
public class Greet {
    public static void main(String[] args) {
        String first = args[0];
        String last = args[1];
        System.out.println("Hello " + first + " " + last + "!");
    }
}
```

```
$ java Greet Gustavo Almadovar
```

```
Hello Gustavo Almadovar!
```

String equality

We must compare strings by writing `s.equals(s)`.

- Writing `str1 == str2` is wrong.

Q:

What is the output of this program?

```
public class Quiz {
    public static void main(String[] args) {
        String word = "Java";
        String sep = "+";
        String result = word + sep + word + sep + sep;
        System.out.println(result);
    }
}
```

Q: What is the difference between `print` and `println`?

Integers

Data type for computing with integers: `int`

`int` data type

values	integers between -2^{31} and $2^{31}-1$				
typical literals	1234	99	0	1000000	
operations	add	subtract	multiply	divide	remainder
operator	+	-	*	/	%

Important note:

Only 2^{32} different `int` values.

not *quite* the same as integers

Examples of `int` operations

expression	*value*	*comment*
5 + 3	8	
5 - 3	2	
5 * 3	15	
5 / 3	1	*drop fractional part*
5 % 3	2	*remainder*
1 / 0		*runtime error*

Precedence

expression	*value*	*comment*
3 * 5 - 2	13	** has precedence*
3 + 5 / 2	5	*/ has precedence*
3 - 5 - 2	-4	*left associative*
(3 - 5) - 2	-4	*better style*

Typical usage: Math calculations; specifying programs (stay tuned).

Integer example

```
public class Rectangle {
    public static void main(String[] args) {
        int length = Integer.parseInt(args[0]);
        int width = Integer.parseInt(args[1]);
        int area = length * width;
        System.out.println(area);
    }
}
```

```
$ java Rectangle 5 8
40
```

Division vs. modulus

When we divide, we get a quotient and a remainder.

$$31 \div 10 = 3 \ \frac{1}{10}$$

The `/` operator performs **floor division**. It returns only the quotient.

`31 / 10`	`3`
`2 / 5`	`0`

The `%` operator returns only the remainder.

`31 % 10`	`1`
`2 % 5`	`2`

Q: What is `0` / `0`?

Speed calculator: km/h to m/s

```
public class ConvertSpeed {
    public static void main(String[] args) {
        double input = Double.parseDouble(args[0]);
        double secondsPerHour = 60 * 60;
        double metersPerKilometer = 1000;
        double result = (input * metersPerKilometer) / secondsPerHour;
        System.out.println(result);
    }
}
```

```
$ java ConvertSpeed 100
27.777777777778
```

Floating-point

Data type for computing with floating point numbers: double

double data type

values	real numbers				
typical literals	3.14159	2.0	1.4142135623730951	6.022e23	
operations	add	subtract	multiply	divide	remainder
operator	+	-	*	/	%

6.022×10^{23}

Typical double values are *approximations*

Examples:

no double value for π.

no double value for $\sqrt{2}$

no double value for 1/3.

Examples of double operations

expression	*value*
3.141 + .03	3.171
3.141 - .03	3.111
6.02e23/2	3.01e23
5.0 / 3.0	1.6666666666666667
10.0 % 3.141	0.577
Math.sqrt(2.0)	1.4142135623730951

Special values

expression	*value*
1.0 / 0.0	Infinity
Math.sqrt(-1.0)	NaN

"not a number"

Typical use: Scientific calculations.

Excerpts from Java's Math Library

public class Math	
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
double sin(double theta)	*sine function*
double cos(double theta)	*cosine function*
double tan(double theta)	*tangent function*
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)*

also defined for int, long, and float

inverse functions also available: asin(), acos(), and atan()

Degrees in radians. Use toDegrees() and toRadians() to convert.

You can discard your calculator now (please).

Example of computing with floating point numbers: quadratic equation

From algebra: the roots of $x^2 + bx + c$ are $\dfrac{-b \pm \sqrt{b^2 - 4c}}{2}$

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

      // Parse coefficients from command-line.
      double b = Double.parseDouble(args[0]);
      double c = Double.parseDouble(args[1]);

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

      // Print them out.
      System.out.println(root1);
      System.out.println(root2);
   }
}
```

```
% java Quadratic -3.0 2.0
2.0                                   x^2 - 3x + 2
1.0

% java Quadratic -1.0 -1.0
1.618033988749895                     x^2 - x - 1
-0.6180339887498949

% java Quadratic 1.0 1.0
NaN                                   x^2 + x + 1
NaN

% java Quadratic 1.0 hello
java.lang.NumberFormatException: hello

% java Quadratic 1.0
java.lang.ArrayIndexOutOfBoundsException
```

Need two arguments.
(Fact of life: Not all error messages are crystal clear.)

Example: Celsius to Fahrenheit

We can write a program to to convert Celsius to Fahrenheit. Recall:

$$F = C \times \frac{9}{5} + 32$$

```
public class CelsiusToFahrenheit {
    public static void main(String[] args) {
        double celsius = Double.parseDouble(args[0]);
        double fahrenheit = celsius * (9 / 5) + 32;
        System.out.println(fahrenheit);
    }
}
```

Example: Celsius to Fahrenheit

Standard human body temperature: **37°C** = **98.6°F**

```
$ java CelsiusToFahrenheit 37
69.0
```

Example: Celsius to Fahrenheit

Standard human body temperature: **37°C = 98.6°F**

```
$ java CelsiusToFahrenheit 37
69.0
```

What went wrong?

Example: Celsius to Fahrenheit

Q: Can you spot the bug? (discuss)

$$F = C \times \frac{9}{5} + 32$$

```java
public class CelsiusToFahrenheit {
    public static void main(String[] args) {
        double celsius = Double.parseDouble(args[0]);
        double fahrenheit = celsius * (9 / 5) + 32;
        System.out.println(fahrenheit);
    }
}
```

Example: Celsius to Fahrenheit

Solution: We need to use the `double` data type within our formula.

```
public class CelsiusToFahrenheit {
    public static void main(String[] args) {
        double celsius = Double.parseDouble(args[0]);
        double fahrenheit = celsius * (9.0 / 5.0) + 32.0;
        System.out.println(fahrenheit);
    }
}
```

```
$ java CelsiusToFahrenheit 37
98.60000000000001
```

Purity

A **pure** mathematical function:

- always succeeds
- always returns the same result
- does not affect anything else

Pure	Impure
`Math.pow`	`Math.random`
`String.length`	`System.out.println`

Q: What is `0.0 / 0.0`?

Division by zero

- **Integer** division by zero causes an **exception**.
  - Exceptions are Java's way of saying "something went wrong".
- **Float** division by zero results in a **special value**.
  - These values behave strangely. Try to avoid them.

Integers		**Floats**	
Input	Result	Input	Result
`1 / 0`	`ArithmeticException`	`1.0 / 0.0`	`Infinity`
`0 / 0`	`ArithmeticException`	`0.0 / 0.0`	`NaN`
`-1 / 0`	`ArithmeticException`	`-1.0 / 0.0`	`-Infinity`

Q: What are some mathematical functions – other than division – that are only partially defined?

Q: Which has the larger value, a `double` or an `int`?

Booleans

Data type for computing with true and false: `boolean`

`boolean` data type

values	true false
literals	`true` `false`
operations	and or not
operator	`&&` `||` `!`

Truth-table definitions

a	!a
true	false
false	true

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

Recall first lecture

Q. a XOR b?

A. `(!a && b) || (a && !b)`

Proof

a	b	!a && b	a && !b	(!a && b) || (a && !b)
false	false	false	false	false
false	true	true	false	true
true	false	false	true	true
true	true	false	false	false

Typical usage: Control logic and flow of a program (stay tuned).

Boolean example

We can manipulate Booleans with the operators: not (`!`), or (`||`), and and (`&&`).

```
public class Booleans {
    public static void main(String[] args) {
        boolean a = true;
        boolean b = !a;
        boolean c = a || b;
        boolean d = a && b;
        System.out.println("a: " + a);
        System.out.println("b: " + b);
        System.out.println("c: " + c);
        System.out.println("d: " + d);
    }
}
```

```
$ java Booleans
```

```
a: true
b: false
c: true
d: false
```

Complex Boolean expression

Evaluating this expression with `a = true`, `b = false`, `c = true`:

```
(a || b) && ~(a || c) || (b && a) || (c && b)
(true || false) && ~(true || true) || (false && true) || (true && false)
```

Step-by-step evaluation:

```
(true || false) && ~(true || true) || (false && true) || (true && false)
     true       && ~(true || true) || (false && true) || (true && false)
     true       &&      ~true      || (false && true) || (true && false)
     true       &&      false      || (false && true) || (true && false)
              false                || (false && true) || (true && false)
              false                ||      false      || (true && false)
              false                ||      false      ||     false
                                 false
```

Computing with Data Types

Comparison operators

Fundamental operations that are defined for each primitive type allow us to *compare* values.

- Operands: two expressions of the same type.
- Result: a value of type `boolean`.

operator	*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`

Examples

non-negative discriminant?	`( b*b - 4.0*a*c ) >= 0.0`
beginning of a century?	`( year % 100 ) == 0`
legal month?	`( month >= 1 ) && ( month <= 12 )`

Typical double values are *approximations* so beware of == comparisons

Example of computing with booleans: leap year test

Q. Is a given year a leap year?

A. Yes if either (i) divisible by 400 or (ii) divisible by 4 but not 100.

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

      // divisible by 4 but not 100
      isLeapYear = (year % 4 == 0) && (year % 100 != 0);

      // or divisible by 400
      isLeapYear = isLeapYear || (year % 400 == 0);

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

```
% java LeapYear 2016
true

% java LeapYear 1993
false

% java LeapYear 1900
false

% java LeapYear 2000
true
```

Q: What is the difference between = and ==?

Q: Find the imposter(s):

==, >=, <=, =<, !=.

Type Conversions

Type checking

Types of variables involved in data-type operations always must match the definitions.

The Java compiler is your *friend*: it checks for type errors in your code.

```
public class BadCode
{
   public static void main(String[] args)
   {
      String s = "123" * 2;
   }
}
```

```
% javac BadCode.java
BadCode.java:5: operator * cannot be applied to java.lang.String,int
        String s = "123" * 2;
                         ^
1 error
```

When appropriate, we often *convert* a value from one type to another to make types match.

Type conversion with built-in types

We use several different methods to convert between types.

For converting to and from `String`, we use a *function*:

Function	From	To	Example	Result
`Integer.parseInt`	`String`	`int`	`Integer.parseInt("123")`	`123`
`Double.parseDouble`	`String`	`double`	`Double.parseDouble("123.45")`	`123.45`
`Integer.toString`	`int`	`String`	`Integer.toString(123)`	`"123"`
`Double.toString`	`double`	`String`	`Double.toString(123.45)`	`"123.45"`

Type conversion with built-in types

For converting between numeric types, we use a *cast*:

Cast	From type	To type	Example	Result
`(int)`	`double`	`int`	`(int) 123.45`	`123`
`(double)`	`int`	`double`	`(double) 123`	`123.0`

Doubles become *truncated*: `(int) 999.99 = 999`

Type conversion with built-in types

Java automatically converts types in some cases.

- In mixed-type arithmetic, numbers get cast to the more precise type.
- When strings concatenation is involved, numbers get converted to `String`.

Left type	Right type	Result type	Example	Result
`int`	`double`	`double`	`11 * 0.25`	`2.75`
`double`	`int`	`double`	`0.01 + 999`	`999.01`
`String`	`int`	`String`	`"x: " + 99`	`"x: 99"`
`double`	`String`	`String`	`3.14 + "159"`	`"3.14159"`

Pop quiz on type conversion

Q. Give the type and value of each of the following expressions.

a. `( 7 / 2 ) * 2.0`

b. `( 7 / 2.0 ) * 2`

c. `"2" + 2`

d. `2.0 + "2"`

Example: guessing game

```java
public class Guess {
    public static void main(String[] args) {
        int answer = (int) (Math.random() * 10);
        int guess = Integer.parseInt(args[0]);
        boolean correct = answer == guess;
        System.out.println("       Your answer was: " + correct);
        System.out.println("The correct answer was: " + answer);
    }
}
```

```
$ java Guess 2
```

```
       Your answer was: false
The correct answer was: 4
```

Overflow and Underflow

Overflow

Java `int`s are not mathematical integers ($\mathbb{Z}$). They have a **maximum** and **minimum**.

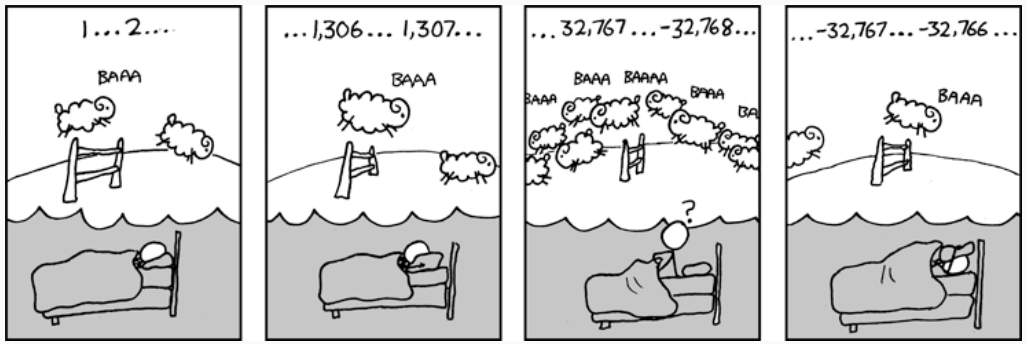
$$\underbrace{-2147483648}_{\text{min int}}, \ldots, -1, 0, 1, \ldots, \underbrace{2147483647}_{\text{max int}}$$

(Roughly 2 billion)

If we try to add more than the maximum value, the result **overflows** and "wraps around" to the negative value.

```
> 2147483644 + 123
```

```
-2147483529
```



Underflow

We get **underflow** by going in the opposite direction.

```
> ((5000 - 1000000000) - 1000000000) - 1000000000
```

```
1294972296
```

Loss of precision

Java's floating-point numbers have **limited precision**.

When you write `0.1`, Java actually sees something like:

```
0.1000000000000000555...
```

The precision loss means that some mathematical equalities don't hold:

```
0.1 + 0.2 == 0.3
```

```
false
```

Lesson: Be careful when comparing floating-point numbers!

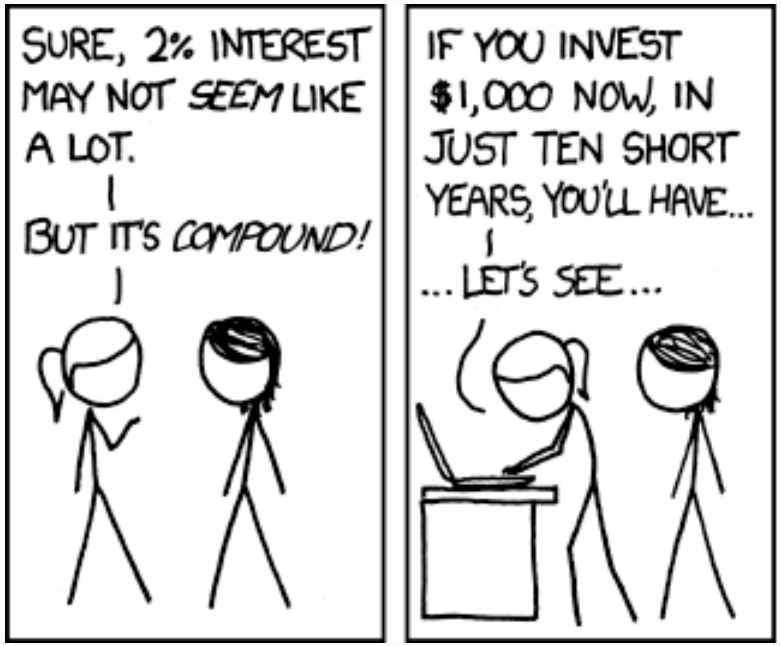
Example: compound interest

We can calculate compound interest with the standard formula.

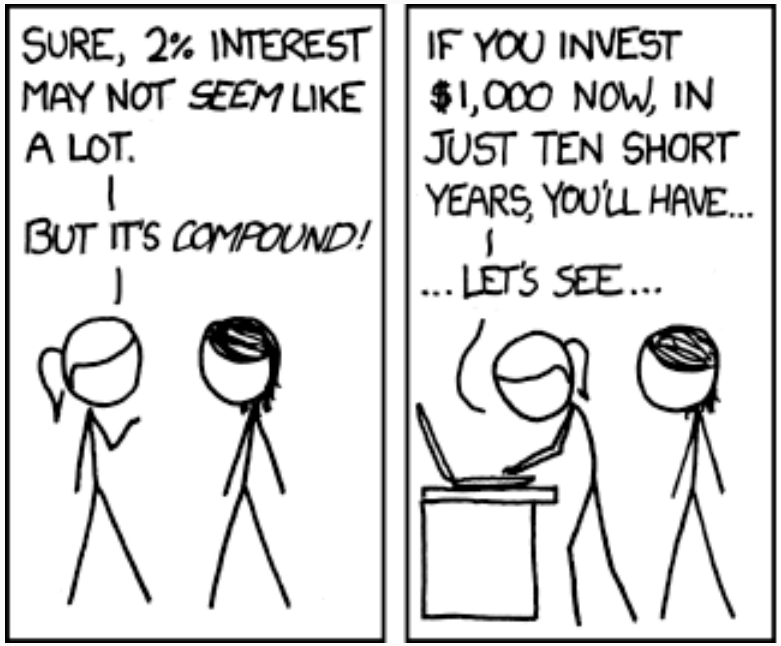
$$T = P \times (1 + r)^t$$

```
public class CompoundInterest {
    public static void main(String[] args) {
        double principal = Double.parseDouble(args[0]);
        double rate = Double.parseDouble(args[1]);
        double time = Double.parseDouble(args[2]);
        double total = Math.pow(principal * (1 + rate), time);
        System.out.println(total);
    }
}
```

Example: compound interest



Example: compound interest




```
$ java CompoundInterest 1000 0.02 10
1.2189944199947571E30
```

$12,000,000,000,000,000,000,000,000,000,000

What went wrong?

Example: compound interest

Q: What is wrong with this program? (discuss)

$$T = P \times (1 + r)^t$$

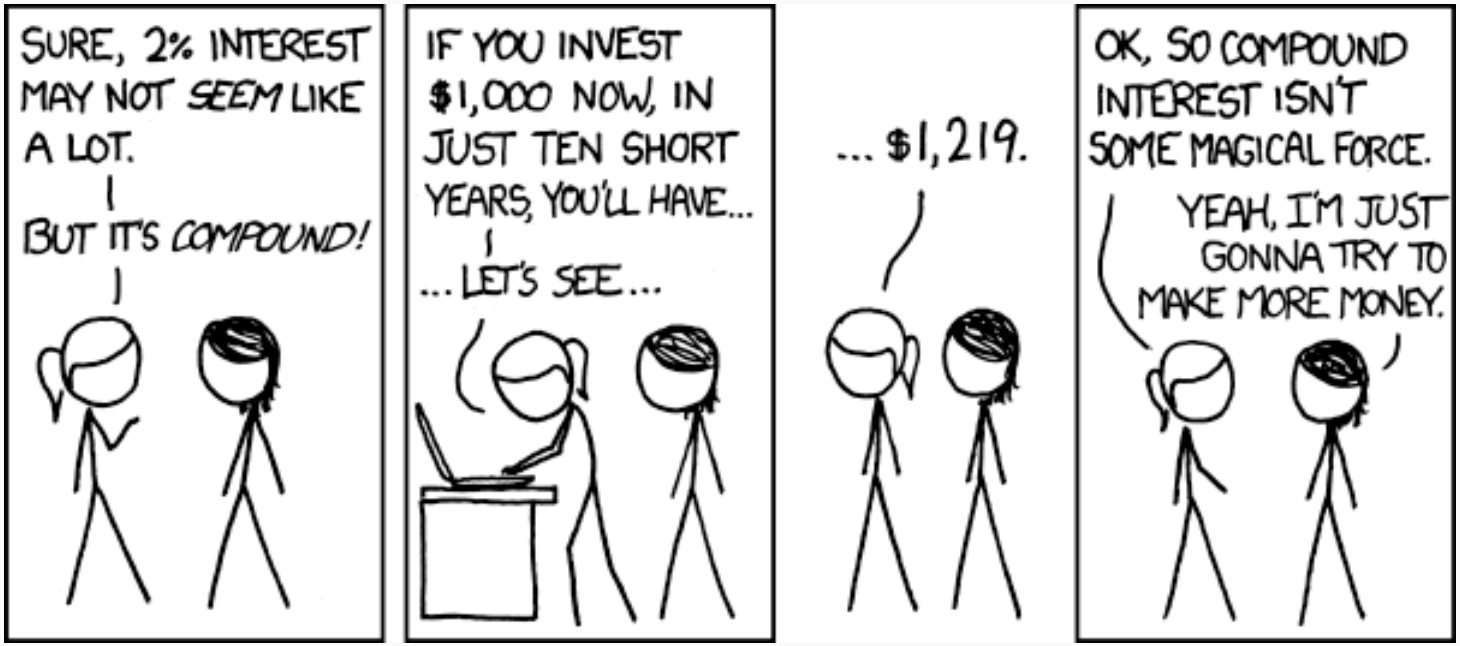
```
public class CompoundInterest {
    public static void main(String[] args) {
        double principal = Double.parseDouble(args[0]);
        double rate = Double.parseDouble(args[1]);
        double time = Double.parseDouble(args[2]);
        double total = Math.pow(principal * (1 + rate), time);
        System.out.println(total);
    }
}
```

Example: compound interest

Solution: We need to pay attention to precedence:

```
public class CompoundInterest {
    public static void main(String[] args) {
        double principal = Double.parseDouble(args[0]);
        double rate = Double.parseDouble(args[1]);
        double time = Double.parseDouble(args[2]);
        double total = principal * Math.pow((1 + rate), time);
        System.out.println(total);
    }
}
```

Example: compound interest



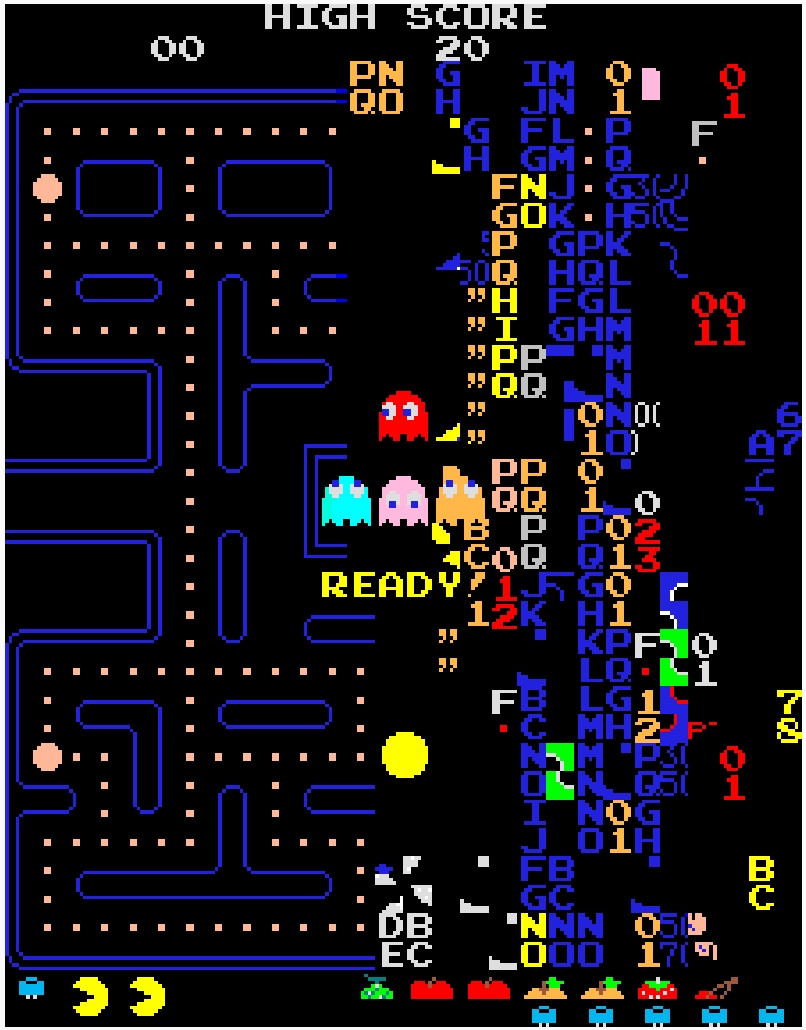

```
$ java CompoundInterest 1000 0.02 10
1218.9944199947574
```

Integer overflow in Pac-Man

Pac-Man's level counter is stored in an 8-bit integer, whose maximum value is 255.

When a player beats level 255, the game adds 1 to the counter, which wraps around to 0.

Chaos ensues.



Epilogue

Compiler error of the week (1/2)

What is the problem here?

```
public class Main {
    public static void main(String[] args) {
        byte b = 300;
    }
}
```

Compiler error of the week (1/2)

What is the problem here?

```
public class Main {
    public static void main(String[] args) {
        byte b = 300;
    }
}
```

```
Main.java:3: error: incompatible types: possible lossy conversion from int to byte
        byte b = 300;
                 ^
1 error
```

Compiler error of the week (2/2)

What is the problem here?

```
public class Main {
    public static void main(String[] args) {
        int d = true / false;
    }
}
```

Compiler error of the week (2/2)

What is the problem here?

```
public class Main {
    public static void main(String[] args) {
        int d = true / false;
    }
}
```

```
Main.java:3: error: bad operand types for binary operator '/'
        int d = true / false;
                     ^
  first type:  boolean
  second type: boolean
1 error
```

Q: Find the imposter(s): `triple`, `bool`, `char`, `byte`, `int`, `bit`, `float`, `String`, `decimal`, `short`, `long`.

Live programming

- Understanding the difference between `print` and `println`.
- Creating and updating local variables.
- Comparing values with `<`, `<=` and so forth.
- Using variables that have different types.
- Converting between different types (with functions and casts).
- What is the meaning of `+`, `+`, and `" + "`.
- Example: Converting days into seconds.
- Example: Converting m/s into miles/h.

Sources for images and slides

- https://introcs.cs.princeton.edu/java/lectures/
- https://xkcd.com/571/
- https://xkcd.com/947/
- https://pacman.fandom.com/wiki/Map_256_Glitch