Is data, value, or object???

Definition

Data is the basis for reasoning or calculation, the quantities used by a computer application.

Definition

Values are the specific quantities used in computer computation.

Definition

Objects are the more abstract quantities used in computer computation, e.g., the color red.

Data Types

A computer and hence a computer program computes on or refers to “things” of interest. These things go by different names data, objects, quantities, or values. We often think of integers as being the primary data values. But at the lowest level all data is just binary bits.

But a program language organizes the bits into varieties called *data types*. With different data types the same bit pattern might represent a different object. A language can provide assistance in computing with these data types giving the illusion or the abstraction of computing with complex objects like real numbers, sound, video, graphs, etc.

Now we look at the data types supported by the Java programming language.

Overview

- Definition: *objects* are everything but primitives
- The eight primitive data type in Java
    - twos complement representation, IEEE754
- The wrapper classes: `java.lang.Boolean`, `Character`, `Float`, etc.
- `java.math.BigInteger`, `java.math.BigDecimal`
- Strings: `java.lang.String`, and `java.lang.StringBuilder`, but do not use `java.lang.StringBuffer`.
- Arrays and its "wrapper" class `java.util.Arrays`
- Lists (abstractly and practically) is a later topic.
- Enums (particular kind of classes)
- Characters: Unicode

Know the Java API (i.e., memorize) for `Integer`, `String`, `StringBuilder`, and `Arrays`.

Java Primitive Data Types

The primitive data types (of which there are eight in Java) are those data types with simple structure and can be represented in the 32 or 64 bits of hardware, for example `int` and `long`. The operations on the primitive data types usually have hardware support. So computing with primitive data types is typically fast and convenient. These data types are used a lot in computing, though as computers get more powerful, programmers get more knowledgeable, and APIs get more expressive, it is more and more common for programs to use more complex data types.

Java Objects

The other families of data types in Java are called *objects.* (This is another use of the overworked word "object." In Java jargon an object is a specific category of data types: namely all those that are not primitive.)

Arrays and strings are objects, not primitive types (but they have special support in the language syntax).
The programmer can even define new Java objects and this is extremely important. This is the subject of a later lecture.

Java Primitive Data Types

- `boolean`
- `char` (16 bit)
- arithmetic
  - integral (twos-complement representation)
    - `byte` (8 bit)
    - `short` (16 bit)
    - `int` (32 bit)
    - `long` (64 bit)
  - floating-point (IEEE 754)
    - `float` (32 bit)
    - `double` (64 bit)

`boolean`

There are only two different boolean values: true and false. The boolean data type is essential for expressions to control conditional statements and loops.

The actual implmentation used by the computer is entriely immaterial to the programmer—a true abstract data type.

However, the progammer must be aware of the implementation of the other data types in order to write correct programs. Because these problems are infrequent, it is easy to forget or ignore the details. This make these bugs all the more insidious.

`char`

The data type `char` represents characters of text like the letter 'A', 'B', etc. The repertoire of characters comes from the important and well-established Unicode standard. We discuss this interesting and Byzantine collection later in much detail.

Arithmetic Data Types

In an ideal world it ought not to matter how the data is represented. However, the usual mathematical laws do not apply to data represented by a computer. For example, none of the following hold for all values:

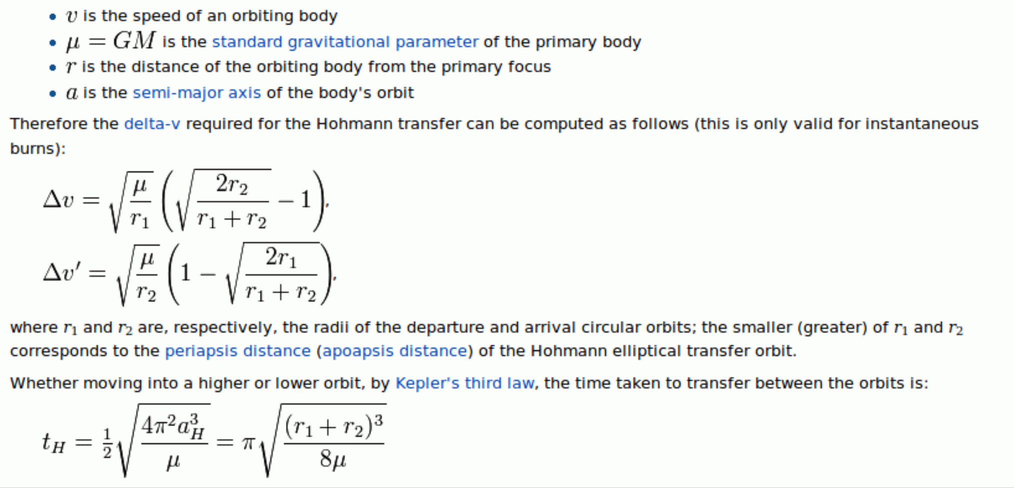
$$|x| \geq 0$$

$$x(y+z) = xy + xz$$

$$x + 1 > x$$

Worse is that fact that the laws often apply and so people educated in mathematics may delude themselves into thinking that they can program a computer. Society is beset by the consequences.

One can look up, say, the equations for Hohmann transfer orbit on Wikipedia. But a computer scientist can translate the concept into correctly working computer programs.



- v is the speed of an orbiting body
- $\mu = GM$ is the standard gravitational parameter of the primary body
- r is the distance of the orbiting body from the primary focus
- a is the semi-major axis of the body's orbit

Therefore the delta-v required for the Hohmann transfer can be computed as follows (this is only valid for instantaneous burns):

$$\Delta v = \sqrt{\frac{\mu}{r_1}}\left(\sqrt{\frac{2r_2}{r_1+r_2}}-1\right),$$

$$\Delta v' = \sqrt{\frac{\mu}{r_2}}\left(1-\sqrt{\frac{2r_1}{r_1+r_2}}\right),$$

where r_1 and r_2 are, respectively, the radii of the departure and arrival circular orbits; the smaller (greater) of r_1 and r_2 corresponds to the periapsis distance (apoapsis distance) of the Hohmann elliptical transfer orbit.

Whether moving into a higher or lower orbit, by Kepler's third law, the time taken to transfer between the orbits is:

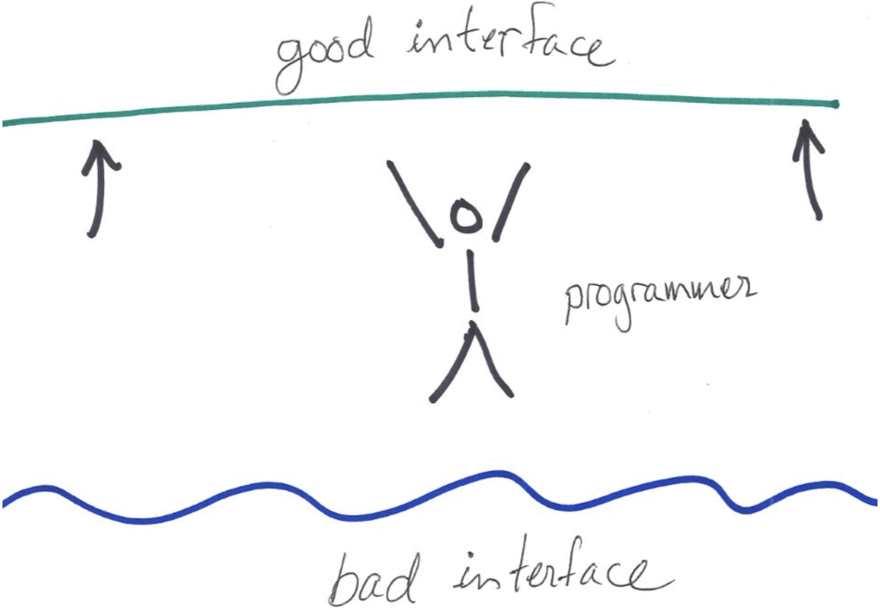
$$t_H = \frac{1}{2}\sqrt{\frac{4\pi^2 a_H^3}{\mu}} = \pi\sqrt{\frac{(r_1+r_2)^3}{8\mu}}$$

Failure to know the science of computing may led to disaster. Take the Ariane 5 disaster and many other failures, both big and small.

See the Ariane 5 explosion ↗ on YouTube.

Doing math is a lot like programming, but we are usually lost in the weeds and fail to the see the goal at all. A computer does math rather poorly, and so the programmer must understand the limitations and avoid them, or a provide better and better infrastructure.

In fact, looking at software development over time, much of programming can be viewed as taking a bad interface and improving it.


good interface
programmer
bad interface

By understanding data, particularly arithmetic data, it is possible to write correct programs in Java. But this is only possible because the Java programming language *defines* its data types. This means that a correctly written program will execute that same regardless of the Java implementation or the hardware on which it is run.

In some languages, notably C and C++, the properties of the data depend on the implementation or the hardware. Programs written in these languages are difficult to port.

Integral Data Types

Integral Data Types

How should an integer be represented?

Binary numbers

Using binary numbers as in discrete math class to represent integers has two problems:

1. They assume an indefinite number of bits.
2. They do not have negative numbers.

So given a fixed number of bits to represent an integer, what system of bit patters do we use?

Binary numbers

Using binary numbers as in discrete math class to represent integers has two problems:

1. They assume an indefinite number of bits.
2. They do not have negative numbers.

So given a fixed number of bits to represent an integer, what system of bit patters do we use?

Sign-magnitude seems simple enough. Why not?

Binary numbers

Using binary numbers as in discrete math class to represent integers has two problems:

1. They assume an indefinite number of bits.
2. They do not have negative numbers.

So given a fixed number of bits to represent an integer, what system of bit patters do we use?

Sign-magnitude seems simple enough. Why not?

Two bit patterns for zero.

Binary numbers

Using binary numbers as in discrete math class to represent integers has two problems:

1. They assume an indefinite number of bits.
2. They do not have negative numbers.

So given a fixed number of bits to represent an integer, what system of bit patters do we use?

Sign-magnitude seems simple enough. Why not?

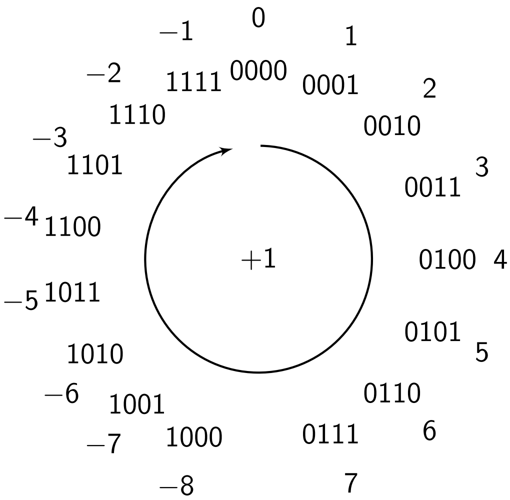
Two bit patterns for zero.

Two's complement was proposed by John von Neuman and is universally used by computer hardware. (I not not sure exactly why.)

Two's complement (8 bits)

$$\begin{array}{ccr}
0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & = & 127 \\
\vdots & & \\
0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 & = & 2 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 & = & 1 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & = & 0 \\
1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & = & \text{-1} \\
1\ 1\ 1\ 1\ 1\ 1\ 1\ 0 & = & \text{-2} \\
\vdots & & \\
1\ 0\ 0\ 0\ 0\ 0\ 0\ 1 & = & \text{-127} \\
1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & = & \text{-128}
\end{array}$$

Two's complement

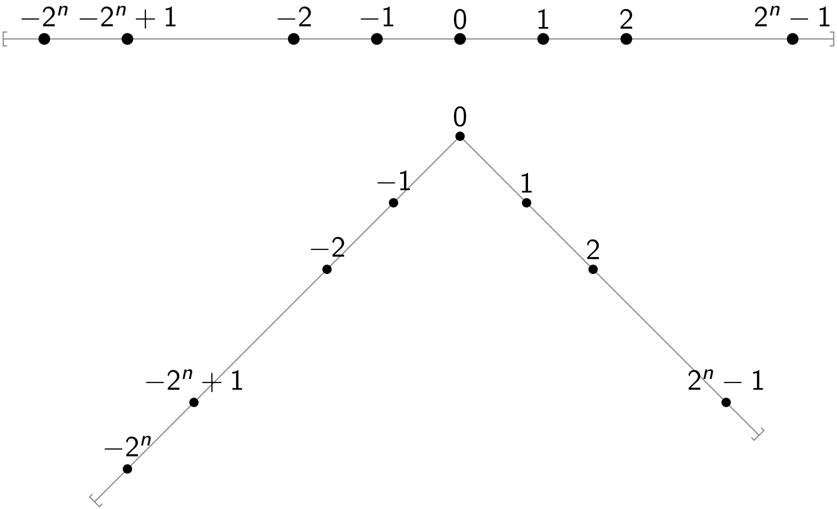


Two's complement

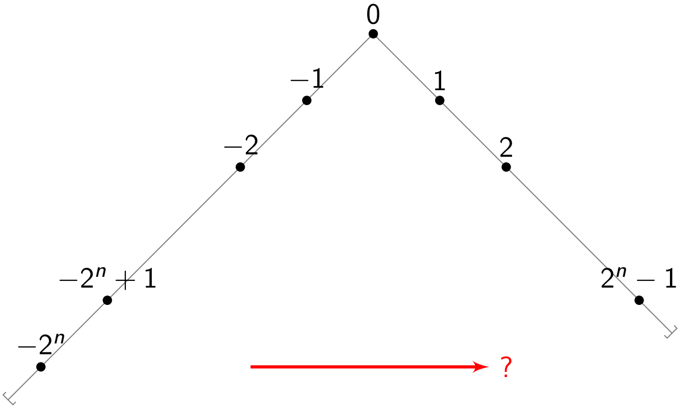
- fixed number of bits, hence bounded
- `int` ± nine (decimal) digits, `long` ± nine (decimal) digits,
- one bit pattern for zero, asymetric
- cyclic (wraps abound)
- signed; no unsigned data type in Java `Integer.toUnsignedString()` ↗, `Integer.toUnsignedLong()` ↗ `Integer.tocompareUnsigned()` ↗
- caution with % (negative numbers); see topic on expresssions

Two's complement

Two's complement



Two's complement

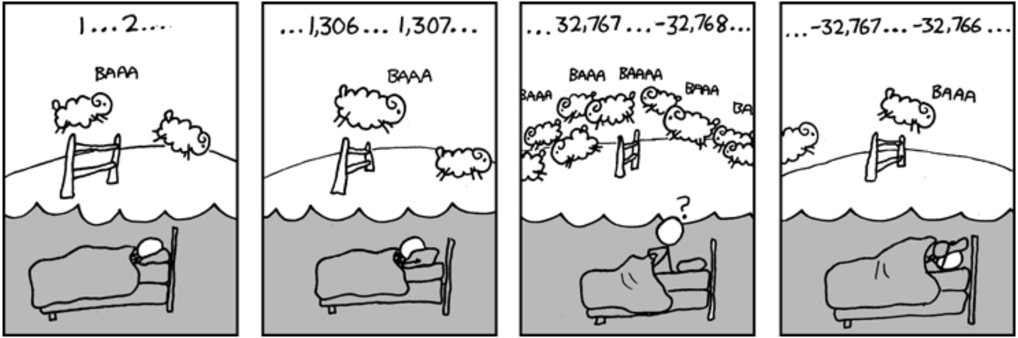


`byte`

The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127. The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
In computing many think of the word `byte` as describing an *unsigned* data value of eight bits. This leads to a source of errors in Java code.

`short`

The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767. As with `byte`, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters. It sometimes slower to manipulate units smaller than the word size of the hardware.



```java
short i = Short.MAX_VALUE;
System.out.println (i);
i = i + 1;
System.out.println (i);
System.out.println (Short.MIN_VALUE};
```

```
32767
-32768
-32768
```

Integral data type silently overflow.

`int`

The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647. For integral values, this data type is generally the default choice unless there is a reason to choose something else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.

approximately, ± a billion, all ± nine (decimal) digit numbers

The Library of Congress holds approximately 119 million items.
Approximately 400,000,000 native speakers of English.
The Guide Star Catalog II (2008) lists 945,592,683 stars.
The music video for South Korean singer Psy's Gangnam Style has been viewed more than 2,147,483,647 times.
Now 2.9 billion active Facebook users (2021) up from 500,000,000 users in 2010.

`long`

The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807. Use this data type when you need a range of values wider than those provided by int.

approximately, ± a quintillion, all ± 18 (decimal) digit numbers
not quite enough to count all the insects on earth, or the possible states of the Rubik's cube

Integer Literals

For `int` and `long` only.

```
0     2     0372     0xDada_Cafe      1996     0x00_FF__00_FF
0l     0777L     0x100000000L     2_147_483_648L     0xC0B0L

0b0110_1101_0000
0b0110_1101_0000L

0xffff_ffff                                /* -1   */
0b1111_1111_1111_1111_1111_1111_1111_1111  /* -1   */
```

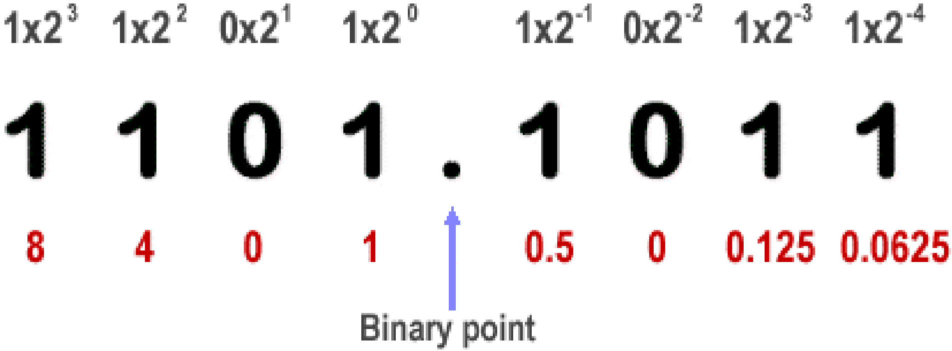
Don't use octal literals!

Floating-Point Types

Fractional Numbers

For any base (2, 10, 16, etc) the positional representation of integers can easily extended to fractions including negative exponents.

Fractional Binary Numbers



1×2^3	1×2^2	0×2^1	1×2^0		1×2^{-1}	0×2^{-2}	1×2^{-3}	1×2^{-4}
1	1	0	1	.	1	0	1	1
8	4	0	1		0.5	0	0.125	0.0625

Binary point

8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 13.6875 (Base 10)

Why float the point?

A compromised between efficiency (the limited number of bits makes the operations easily implementable in hardware) and accuracy (not all real number can be represented).

This solution is not perfect and in some contexts one might represent the constructable reals with computer programs.

Scientific Notation

An practical way of presenting real numbers has been developed over a long period of time by scientists: scientific notation.
In scientific notation all numbers are written in the form of

$$a \times 10^b$$

(a times ten raised to the power of b), where the exponent b is an integer, and the coefficient a is a real number, called the significand or mantissa. The exponent shifts (floats) the decimal point.

This notation avoids lots of zeros which tend to make large and small numbers hard to read. A variant of this notation is often used by programming languages for the input and output of real numbers. Because of the typographical difficulty of superscripts these numbers are written with the letter 'e' or 'E' for "exponent."

$$a\mathrm{E}b$$

Floating Point

Floating point uses a fixed, total number of bits in the representation of real numbers.
More bits in the exponent expand the range of numbers represented and the more bits in the mantissa extend the precision of the numbers represented.

IEEE754

bits	exponent	mantissa	bias	range	places
32	8	23	127	$1.5 \cdot 10^{-45} \ldots 3.4 \cdot 10^{38}$	7–8
64	11	52	1023	$5.0 \cdot 10^{-324} \ldots 1.7 \cdot 10^{308}$	15–16

$$1 = 1 \cdot 2^0 = 1 \cdot 2^{127-127} = 0x3F800000 = 00111111110 \cdots 0$$

$$1 = 1 \cdot 2^0 = 1 \cdot 2^{1023-1023} = 0x3FF0000000000000 = 00111111111110 \cdots 0$$

IEEE754

A sign bit, followed by w exponent bits that describe the exponent offset by a bias $b =$, and $p - 1$ bits that describe the mantissa.

$$s \cdot m \cdot 2^e$$

where s is the sign, 0 for positive or 1 for negative. For words of length 32 bits m is a positive integer less that 2^{24}, and e is between -127 and 128, inclusive.

zero 0 0 ± 0
infinity $2b + 1$ 0 $\pm\infty$
denormalized $0 \neq 0$ $\pm 0.f \times 2^{-b+1}$
normalized $1 \leq e \leq 2b$ $\pm 1.f \times 2^{e-b}$
not a number $2b + 1 \neq 0$ NaN

IEEE754

Execute `TestFloat.java` ↗ to show bit patterns for

`Float.NAN`
`Float.POSITIVE_INFINITY`
`Float.MAX_VALUE`
`Float.MIN_VALUE`
`0.1`

and so on.

`float`

The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead.

`double`

The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in section 4.2.3 of the Java Language Specification. For real numbers, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

All the primitive, numeric data in Java are limited (obviously you can only represent 2^{32} or 2^{64} numbers). This is a lot of numbers, but we have the expectation that one does not run out of numbers and this causes trouble in some contexts.

Even worse, the mathematics of computer numbers, is not the same as “real” mathematical numbers and this causes a great deal of trouble.

To ameliorate the these problems the Java libraries provide arbitrary-precision signed decimal numbers `java.lang.BigDecimal` and arbitrary-precision integers `java.lang.BigInteger`. They are not primitive data types, but they have many of the same operations as the primitive data types.

Consider the difference between precise and imprecise quantities.

1. Can you measure temperature exactly? No.
2. Can you measure velocity exactly? No.
3. Can you measure position exactly? No.

But, can you measure money exactly? Yes.

Block, *Effective Java*, Item 48: Avoid float and double, if exact answers are required.

> *The float and double types are designed primarily for scientific and engineering calculations. They perform binary floating-point arithmetic, which was carefully designed to furnish accurate approximations quickly over a broad range of magnitudes. They do not, however, provide exact results and should not be used where exact results are required. The float and double types are particularly ill-suited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a float or double exactly.*

Read "What every computer scientist should know about floating point arithmetic" by D. Goldberg, *ACM Computing Surveys*, volume 23, number 1, 1991, pages 5–48.

Consider the following program.

```
public final class Monetary {

   private static final String FMT =
      "Bought %d items @ $%.2f; funds remaining $%.2f%n";

   public static void main (final String[] args) {
      final double price = 0.10;
      double funds = 2.00;
      int items = 0;
      while (funds >= price) {
         funds -= price;
         items++;
      }
      System.out.format (FMT, items, price, funds);
   }
}
```

The output is a surprising:

```
Bought 19 items @ $0.10; funds remaining $0.10
```

because the number 0.1 cannot be represented exactly in binary.

The problem can be avoided by using `BigDecimal` which does not represent numbers in binary digits.

(Of course, everything is binary in a computer; it is just that `BigDecimal` uses some bit patterns to represent decimal digits.)

```
import java.math.BigDecimal;

public final class Monetary2 {

   private static final String FMT =
      "Bought %d items @ $%s; funds remaining $%s%n";

   public static void main (final String[] args) {
      final BigDecimal price = new BigDecimal (".10");
      BigDecimal funds = new BigDecimal ("2.00");
      int items = 0;
      while (funds.compareTo(price) >= 0) {
         funds = funds.subtract(price);
         items++;
      }
      System.out.format (FMT, items, price, funds);
   }
}
```

```
Bought 20 items @ $0.10; funds remaining $0.00
```

Binary Fractions

How is .1 represented on a computer?

Fractions are the sum of negative powers of two: $0.1 = 1/16 + 1/32 + 1/256 \cdots$.
Binary powers (positive and negative) of two:

```
 3 8.0
 2 4.0
 1 2.0
 0 1.0
-1 0.5
-2 0.25
-3 0.125
-4 0.0625
-5 0.03125
-6 0.015625
-7 0.0078125
```

An illustration of a simple algorithm to convert a tenth to a binary fraction.

$$
\begin{array}{ll}
0.1 & 0. \\
0.1 \times 2 = 0.2 < 1 & 0.0 \\
0.2 \times 2 = 0.4 < 1 & 0.00 \\
0.4 \times 2 = 0.8 < 1 & 0.000 \\
0.8 \times 2 = 1.6 \geq 1 & 0.0001 \\
0.6 \times 2 = 1.2 \geq 1 & 0.00011 \\
0.2 \times 2 = 0.4 < 1 & 0.000110 \\
0.4 \times 2 = 0.8 < 1 & 0.0001100 \\
0.8 \times 2 = 1.6 \geq 1 & 0.00011001 \\
0.6 \times 2 = 1.2 \geq 1 & 0.000110011 \\
0.2 \times 2 = 0.4 < 1 & 0.0001100110
\end{array}
$$

0.1 (decimal) $= 0.00011001100110011\ldots$ (binary)

It may come as a surprise that terminating decimal fractions can have non-terminating expansions in binary. (But not the other way around.) Meaning that some (finite) decimal fractions cannot be represented precisely in a finite number of bits.

It seems “unfair” that all binary fractions can be converted to decimal fractions, but not vice versa. A decimal fraction is a fraction with a power of 10 (10^1, 10^2, etc.) in the denominator.

Exact decimal fractions for each power of two:

16 8 4 2 1 0.5 0.25 0.125 0.0625 0.03125

0.10 in IEEE754

In Java, floating-point numbers can be read in and out (or, equivalently, converted to a string and back) without loss by avoiding base 10.

```
 System.out.format ("f = %f %a %s %08x  %s%n", d, d,
   Float.toString(d), Float.floatToIntBits(d), Float.toHexString(d));
```

```
f = 0.100000 0x1.99999ap-4 0.1 3dcccccd  0x1.99999ap-4
```

```
sign bit            = 0
exponent (8  bits) = 01111011, 123; exp = -4
mantissa (23 bits) = 10011001100110011001101
```

Hexadecimal Floating-Point Literals

Hexadecimal floating-point literals originated in C99 and were later included in a revision of the IEEE 754 floating-point standard.
sign, significand, and exponent fields defining a finite floating-point value; sign '0x' significand 'p' exponent. This syntax allows the literal

`0x1.8p1`

to be to used represent the value 3; $1.8_{16} \times 2^1 = 1.5_{10} \times 2 = 3$. More usefully, the maximum value of can be written as `0x1.fffffffffffffp1023` and the minimum value of 2^{-1074} can be written as `0x1.0P-1074` or `0x0.0000000000001P-1022`, which maps easily to the various fields of the floating-point representation and is much more perspicacious than the raw-bit encoding.
In addition, "printf" facility including the %a format for hexadecimal floating-point.

Wrapper Classes

boolean	Boolean
char	Character
byte	Bytes
short	Short
int	Integer
long	Long
float	Float
double	Double
	Void

Corresponding to each primitive data type there is a Java class, known as its *wrapper class*.

Using wrapper classes, all data (including primitive data) can be considered objects. Thus a pleasing simplicity is achieved.

Wrapper Classes

Instances of the wrapper class act as data just as the primitive data types to. So, an instance of `java.lang.Integer` is a lot like `int`. These wrapper class allows data of each primitive type to be used as an object. This redundancy is especially significant in the connection to generics which can only be used with objects (non-primitive data).

```
final ArrayList<Integer> list = new ArrayList<> (); // legal
final var list = new ArrayList<Integer> ();         // legal
final ArrayList<int> list = new ArrayList<> ();     // NOT le

final Integer[] a = new Integer [5];   // legal
final int[]     b = new int [5];       // legal
a[0] = b[0];   // legal; auto boxing
b[0] = a[0];   // legal; auto unboxing
```

Wrapper Classes

Leaving aside generics (a future topic), wrapper class have another purpose. These class holds a few static methods that make the use of the primitive data type more convenient, for examples, methods to convert between strings and the data type.

```
public static void main (final String[] a) {
  final int n = Integer.parseInt (a[0]);          /* primitive */
  final double x = Double.parseDouble (a[1]);     /* primitive */
  final boolean b = Boolean.parseBoolean (a[2]);/* primitive */
  final BigInteger big = new BigInteger (a[3]); /* object */
}
```

ParseInt in `Integer` class

```
parseInt("42")                 // returns 42
parseInt("0", 10)              // returns 0
parseInt("473", 10)            // returns 473
parseInt("+42", 10)            // returns 42
parseInt("-0", 10)             // returns 0
parseInt("-FF", 16)            // returns -255
parseInt("1100110", 2)         // returns 102
parseInt("2147483647", 10)     // returns 2147483647
parseInt("-2147483648", 10)// returns -2147483648
parseInt("2147483648", 10)  // throws NumberFormatException
parseInt("99", 8)              // throws NumberFormatException
parseInt("Kona", 10)           // throws NumberFormatException
parseInt("Kona", 27);          // returns 411787
```

Wrapper Classes

There is an important implementation distinction which primitive types which are said to be “unboxed” and data of the wrapper classes which are said to be “boxed.” We discuss this later.

The programmer chooses one or the other, but Java converts back and forth implicitly making them appear the same to the programmer.

```java
public static void main (final String[] args) {
   final long defaultValue = 234L;

   // Note auto boxing and un-boxing
   final long g = Long.getLong ("sysPropKey", defaultValue);
   final Long h = 10L * defaultValue;
   final long i = 20L * h;
}
```

String Objects

String literals have special syntax:

`"  . . .  "`

This raises the questions what can be a part of a string and how can a double quote be included in a string.

Do not confuse character escape sequences with Unicode escapes or the format specifiers used in format strings in conjunction with `printf`.

Character Escape

Escape Sequence	Description
`\t`	Insert a tab in the text at this point.
`\b`	Insert a backspace in the text at this point.
`\n`	Insert a newline in the text at this point.
`\r`	Insert a carriage return in the text at this point.
`\f`	Insert a formfeed in the text at this point.
`\'`	Insert a single quote character in the text at this point.
`\"`	Insert a double quote character in the text at this point.
`\\`	Insert a backslash character in the text at this point.

`java.lang.String`

Text Blocks

[As brought up in the section of lexems/literals/string.]
An example of text blocks can be found on-line in this program: `Block.java`

`java.lang.String`

In many cases it is more efficient to use the class `java.lang.StringBuilder` than `java.lang.String`.

`java.lang.String` is a immutable class and `java.lang.StringBuilder` is a mutable class. Immutable classes will cause fewer programming errors, mutable are more efficient to use in some circumstances. This topic will be discussed later.

`java.lang.String`

An immutable sequence of characters.

`int`	`length()`	string length
`char`	`charAt(int i)`	ith character
`String`	`substring(int i,int j)`	ith - j-1 characters
`boolean`	`equals(Object o)`	test for same string
`int`	`compareTo(String s)`	lexicographic ordering

`java.lang.StringBuilder`

A mutable sequence of characters.

`int`	`length()`	string length
`char`	`charAt(int i)`	ith character
`void`	`setCharAt(int i)`	set ith character
`String`	`substring(int i,int j)`	ith - j-1 characters
`boolean`	`equals(Object o)`	*pointer equality*

`StringBuilder` does not implement comparable.

`http://www.javafaq.nu/java-article641.html`

Know the Java API for `String` and `StringBuilder`! (`StringBuffer` class is obsolete.)

java.lang.StringBuilder

- A mutable sequence of characters.
- This class extends the `Object` class. The implication of class inheritance will be discussed alter.
- Has numerous operations that *change* the string: append, insert, delete, replace, reverse, setCharAt, deleteCharAt
- BTW used to implement the '+' operator.

```
new StringBuilder().append ("abc")
  .append("xyz").toString()
```

- Pitfalls. equals() same as == and does not implement comparator

```
sb1.toString().equals (sb2.toString())
sb1.toString().compareTo (sb2.toString())
```

Equality

While on the subject of equality . . .

- for primitive data use ==
- for objects use (the instance method) `equals()`

Unfortunately for some objects `equals()` has not been implemented in the Java API. in the way one expects. So, we must:

- for arrays use `Arrays.equals()`
- for `java.lang.StringBuilder` use

```
sb1.toString().equals (sb2.toString())
```

An array constains a fixed number of spots for values. The values in an array are called elements.

- Creating an array of any type
- Assigning elements
- Accessing elements (zero indedexing)
- Size of an array
- `IndexOutOfBoundsException`
- Iterating (for, for each)
- Array values
- Array wrapper class
- Printing arrays `Arrays.toString()`
- Sorting arrays `Arrays.sort()`
- Subroutines may take arrays as parameters and return arrays as results

Arrays

- Create, Triangle, Bool, Copy, Sort. Section 1.4, page 89.
- java.util.Arrays ↗

Arrays

```
final double[] a = new double[27];
for (int i=0; i<a.length; i++) {
   a[i] = Math.random();
}

for (int i=0; i<a.length; i++) {
   System.out.printf ("%6.2f%n", a[i])
}
// Or, even easier ...
System.out.println (Arrays.toString(a));

double max = Double.NEGATIVE_INFINITY;
for (int i=0; i<a.length; i++) {
   if (a[i]>max) max=a[i];
}
// Or, even easier ...
for (double d: a) if (d>max) max=d;
```

Arrays

Something everybody should know:

```java
final int[] a = {4,2,8,4,7};
System.out.println (Arrays.toString(a));
```

Produces the output:

```
[4, 2, 8, 4, 7]
```

Arrays

```
final int[] cost= new int [n];
Arrays.fill (cost, Integer.MAX_VALUE);
```

Arrays

```
float[] a = {45.1, 32.9, 74.3};
int n = 5;
float[] b = Arrays.copyOf (a, n);
assert b.length==5;

float[] a = {45.1, 32.9, 74.3, 4.8};
int from = 1;// index, inclusive
int to   = 3;// index, exclusive
float[] b = Arrays.copyOfRange (a, from, to);
assert b.length==to-from;
```

Arrays

The “for each” loop is often used with arrays:

```
double sum = 0.0;
final double[] ar = {2.3, 6.93, 0.011};
for (int i=0; i<a.length; i++) {
   sum += a[i];
}

for (double d: ar) {
   sum += d;
}
```

[This does not really belong here.] Conditional expression:

```
int max;
if (a>b) {
   max = a;
} else {
   max = b;
}
max = (a>b) ? a : b;
```

Enum

Build your own data type with a small, fixed number of values.

```
enum Color {RED, WHITE, BLUE}
```

An example program using enums: `Block.java`↗

Enum

```
public final class Card {
   public enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
   public enum Rank {
      DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
      NINE, TEN, JACK, QUEEN, KING, ACE
   }

   final Rank rank;
   final Suit suit;
}
```

Enum

```java
enum Direction {
   RD(+2,-1), RU(+2,+1), LU(-2,+1), LD(-2,-1),
   UL(-1,+2), UR(+1,+2), DL(-1,-2), DR(+1,-2);
   private int dx,dy;
   Direction (int dx, int dy) {
      this.dx=dx; this.dy=dy;
   }
   public final Square move (Square p) {
      return new Square (p.x+dx,p.y+dy);
   }
 }
```