Data Types

The objects about which computer programs compute is data. We often think of integers as being the primary data on which computers operate. 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 are the supported by the Java programming language.
Overview

- Definition: *objects* are everything but primitives
- The eight primitive data type in Java
  - twos complement notation, 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`
- Objects and its “wrapper” class `java.util.Objects`
- Enums (particular kind of classes)
- Characters: Unicode
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.
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 extremely important. This is the subject of a later lecture.
Java Primitive Data Types

- boolean
- char
- arithmetic
  - integral (twos-complement notation)
    - byte
    - short
    - int
    - long
  - floating-point (IEEE)
    - float
    - double
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.
An educated person can look up, say, the equations for Hohmann transfer orbit on Wikipedia. But only 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.
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.
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 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.
Integral Data Types
Integral Data Types

How should an integer be represented?
Binary numbers

Binary numbers as in discrete math class have two problems:

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

Sign-magnitude seems simple enough. Why not?
Binary numbers

Binary numbers as in discrete math class have two problems:

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

Sign-magnitude seems simple enough. Why not?

Two bit patterns for zero.
Two’s complement (8 bits)

\[
\begin{align*}
01111111 &= 127 \\
00000010 &= 2 \\
00000001 &= 1 \\
00000000 &= 0 \\
11111111 &= -1 \\
11111110 &= -2 \\
10000001 &= -127 \\
10000000 &= -128
\end{align*}
\]
Two’s complement

\[ -2^n \quad -2^n + 1 \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad 2^n - 1 \]
Two's complement

\[-2^n - 2^n + 1 - 2 - 1 0 1 2 2^n - 1\]
Two’s complement

$$-2^n \quad -2^n + 1 \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad 2^n - 1$$
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 (inclusive). 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.
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 (inclusive). 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.
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 \(2,147,483,647\) (inclusive). 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, \(\pm\) a billion, all \(\pm\) nine digit (decimal) numbers

The Library of Congress holds approximately 119 million items. Approximately 402,000,000 native speakers of English. Approximately 500,000,000 active Facebook users as of March 2010. The Star Catalog lists 998,402,801 distinct astronomical objects. The music video for South Korean singer Psy’s Gangnam Style has been viewed more than 2,147,483,647 times.
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 \(9,223,372,036,854,775,807\) (inclusive). Use this data type when you need a range of values wider than those provided by int.

approximately, ± a quintillion, all ± 18 digit (decimal) numbers not quite enough to count all the insects on earth, or the possible states of the Rubik’s cube
Floating-Point Types
1x2^3 1x2^2 0x2^1 1x2^0 1x2^{-1} 0x2^{-2} 1x2^{-3} 1x2^{-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?
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.

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.”
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. 

\[ s \cdot m \cdot 2^e \]  

\( s \) is 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 \( \pm 0 \)
- infinity 2\( b + 1 \) 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 2\( b + 1 \neq 0 \) NaN
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.
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` 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.

2. Can you measure velocity 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.

Consider the following program.

```java
public final class Monetary {

    private final static 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 final static 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 · · ·
Binary powers (positive and negative) of two:

<table>
<thead>
<tr>
<th>3</th>
<th>8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.0</td>
</tr>
<tr>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>-1</td>
<td>0.5</td>
</tr>
<tr>
<td>-2</td>
<td>0.25</td>
</tr>
<tr>
<td>-3</td>
<td>0.125</td>
</tr>
<tr>
<td>-4</td>
<td>0.0625</td>
</tr>
<tr>
<td>-5</td>
<td>0.03125</td>
</tr>
<tr>
<td>-6</td>
<td>0.015625</td>
</tr>
<tr>
<td>-7</td>
<td>0.0078125</td>
</tr>
</tbody>
</table>
An illustration of a simple algorithm to convert a tenth to a binary fraction.

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

\[0.1 \text{ (decimal)} = 0.00011001100110011\ldots \text{ (binary)}\]

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.
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
- mantissa (23 bits) = 10011001100110011001101
Hexadecimal Floating-Point Literals

[Do literals belong under the topic of expressions or here under data?]
More usefully, the maximum value of \((2 - 2^{-52}) \cdot 2^{1023}\) can be written as \(0x1.fffffffffffffffпп1023\) and the minimum value of \(2^{-1074}\) can be written as \(0x1.0пп-1074\) or \(0x0.00000000000001пп-1022,\)
Wrapper Classes

Corresponding to each primitive data type there is a Java class, known as its *wrapper class*. Instances of the 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 contest of generics which can only be used with objects (non-primitive data). Java boxes and unboxes automatically. Also, this 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.
In many cases it is more efficient to use the class `java.lang.StringBuilder` than `java.lang.String`. `java.lang.String` is an 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
A mutable sequence of characters.

```java
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 later.
► Has numerous operations that change the string: append, insert, delete, replace, reverse, setCharAt, deleteCharAt
► BTW used to implement the ’+’ operator

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

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

```java
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())
```

Arrays

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

```java
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

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;
Conditional expression:

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

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

```java
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;
}
```
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 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);
    }
}