Chapter 4 – Fundamental Data Types

4.1 Number Types

- Recall: All values are either references to objects or one of 8 primitives
- A numeric computation overflows if the result falls outside the range for the number type

```java
int n = 1000000;
System.out.println(n * n);  // prints -727379968
```

Primitives

- So far: 6 number types
  - byte, short, int, long for whole numbers
  - float, double for reals
- Numbers can have overflow (value too large)
- Reals can also have underflow (rounding errors)

```java
double f = 4.35;
System.out.println(100 * f);  // prints 434.99999999999994
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>The integer type, with range -2,147,483,648 ... 2,147,483,647 (about 2 billion)</td>
<td>4 bytes</td>
</tr>
<tr>
<td>byte</td>
<td>The type describing a single byte, with range -128 ... 127</td>
<td>1 byte</td>
</tr>
<tr>
<td>short</td>
<td>The short integer type, with range -32768 ... 32767</td>
<td>2 bytes</td>
</tr>
<tr>
<td>long</td>
<td>The long integer type, with range -9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807</td>
<td>8 bytes</td>
</tr>
<tr>
<td>double</td>
<td>The double-precision floating-point type, with a range of about ±10^308 and about 15 significant decimal digits</td>
<td>8 bytes</td>
</tr>
<tr>
<td>float</td>
<td>The single-precision floating-point type, with a range of about ±10^38 and about 7 significant decimal digits</td>
<td>4 bytes</td>
</tr>
<tr>
<td>char</td>
<td>The character type, representing code units in the Unicode encoding scheme (see Advanced Topic 4.5)</td>
<td>2 bytes</td>
</tr>
<tr>
<td>boolean</td>
<td>The type with the two truth values false and true (see Chapter 6)</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

Conversion

- 2 types of conversion – increase in precision, decrease in precision

```java
int i = 20;
double d = i / 20;
```

Chapter Goals

- To understand integer and floating-point numbers
- To recognize the limitations of the numeric types
- To become aware of causes for overflow and roundoff errors
- To understand the proper use of constants
- To write arithmetic expressions in Java
- To use the String type to define and manipulate character strings
- To learn how to read program input and produce formatted output
Type casting

- Error is because compiler wants us to know that we are going to possibly lose information
  - 4.0 to 4, not a big deal
  - 0.99 to 0, is a big deal
- To override this (i.e. tell the compiler to convert anyways) we must use the type cast operator

Example

```java
double balance = 12.45;
int dollar = (int) balance; // stores 12
```

Type casting

- User defined function
- Example

```
example(element)
```

- Type of element
- Use predefined function

Example

```
example(element)
```

Rounding

- What if you want to round instead of truncate?
- 2 solutions:
  - Hack: add 0.5 to number then truncate
  - Real solution: Use Math.round()

```
long rounded = Math.round(balance);
```

4.2 Constants

- In math, equations have variables and constants
- Constants are numbers that do not change but have specific meanings are important
- Randomly placed numbers in a program can be confusing (and hard to change later), so we give them symbolic names

Constants

- Use a constant to hold the value
  - Value cannot change once it is set
- Use reserved word final to modify declaration
  - ```final double PI = 3.14159;```  
- Example

```
BankAccount
```

Uses

- Constants often used for important numbers
- Make meaning of code easier to understand
- Ex. Go back to BankAccount
  - Want to add something to withdraw method
public void withdraw(double amount)
{
    double newBalance = balance - amount - 5;
    balance = newBalance;
}

What is the 5 doing?
Why is it there?

Constant Fields
- Often, we want to use the same constant across several methods in a class
- Make a constant field (class constant)

public static final int BANK_FEE = 5;

public void withdraw(double amount)
{
    double newBalance = balance - amount - BANK_FEE;
    balance = newBalance;
}

Constant Definition
- In a method (local constant)
  final <type> <name> = <value>;
  final int HOURS_IN_DAY = 24;

- In a class (class constant)
  <accessSpecifier> static final <type> <name> = <value>;
  public static final int HOURS_IN_DAY = 24;

public class Math{
...
  public static final double E = 2.71828
  public static final double PI = 3.14159
  ...
}
Magic Numbers

- So when should I use a constant?
- In general if the number has significance, give it a symbolic name (make it a constant identifier)
  Ex.
  \[
  \text{time} = \text{duration} \times 60;
  \]
  Is this converting minutes to seconds? Hours to minutes?

Increment

- Net result is to increment \text{items} by 1
- Increment is so common, Java uses a shorthand for it
- Increment items by one
  \[
  \text{items}++;\n  \]
- Decrement (subtract) by one
  \[
  \text{items}--;\n  \]

Modulo

- Returns remainder of division
  \[
  23 \% 5 = 3\n  \]
  \[
  4 \% 2 = 2\n  \]
- Mainly used for integers (possible for reals also, but not as useful)

Other Shortcuts

- Arithmetic modifications to a variable can use shorthand operations
  \[
  \text{items} = \text{items} + 5; \rightarrow \text{items} += 5;\n  \text{items} = \text{items} * 2; \rightarrow \text{items} *= 2;\n  \text{items} = \text{items} - 2*5; \rightarrow \text{items} -= 2*5;\n  \text{items} = \text{items} / x; \rightarrow \text{items} /= x;\n  \]

Precedence

\begin{itemize}
  \item \textbf{High}
    \begin{itemize}
      \item Subexpressions \((())\)
      \item Unary operators \(-\) (negative)
      \item Multiplicative operators \(*, /, \%\)
      \item Additive operators \(+, -\)
    \end{itemize}
  \item \textbf{Low}
\end{itemize}
**Integer Division**

- When dividing two integers, only the whole number is returned
- Has its advantages

```java
int totalTime = 503;
int MINUTES_PER_HOUR = 60;
int hours = totalTime / MINUTES_PER_HOUR;
int minutes = totalTime % MINUTES_PER_HOUR;
```

**Math Class**

- Provided automatically (package `java.lang`)
- Power
  ```java
  Math.pow(x, n) \rightarrow x^n
  ```
- Square root
  ```java
  Math.sqrt(x) \rightarrow x^{0.5}
  ```

**Check**

- What is the value of 1729 / 100? Of 1729 % 100?
- Computing Average: Is this correct?
  ```java
double average = s1 + s2 + s3 / 3;
```
- What do we have to worry about if `s1, s2, s3` are integers?

**Avoid Redundant Code**

- Important property for good programming – never have to lines of code doing the exact same thing
- Won’t be heavily enforced, but good to get practice
- Inefficient – making computer do work twice
- Error prone – when there is a bug, you may only fix it one place

**Example – Quadratic Eqn**

```java
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
x2 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

versus

```java
double root = Math.sqrt(b * b - 4 * a * c);
x1 = (-b + root) / (2 * a);
x1 = (-b - root) / (2 * a);
```
4.5 Calling Static Methods

- So far we have been discussing classes as templates and objects as the entities.
- All the methods and instance fields for the `BankAccount` class belonged to an individual object.
- Each had its own `balance`.
- A method call was on the object itself.
- `harrysChecking.getBalance()` different than `myChecking.getBalance()`.

Static Methods

- Has everything we’ve used been an object?
  - `System`
  - `Math`.
- Both of these are classes, yet we made calls on them to either fields (e.g., `System.out`) or methods (`Math.sqrt()`) of `Math`.
- We never created an instance `Math` of the class or `System` class.

Other Math Class Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.sqrt(x)</td>
<td>Square root of x (double)</td>
</tr>
<tr>
<td>Math.pow(x, y)</td>
<td>$x^y$ (x &gt; 0; or x = 1 and y &gt; 0, or x &lt; 0 and y an integer)</td>
</tr>
<tr>
<td>Math.sin(x)</td>
<td>Sine of x (in radians)</td>
</tr>
<tr>
<td>Math.cos(x)</td>
<td>Cosine of x</td>
</tr>
<tr>
<td>Math.tan(x)</td>
<td>Tangent of x</td>
</tr>
<tr>
<td>Math.asin(x)</td>
<td>Arc sine (x in [-1, 1])</td>
</tr>
<tr>
<td>Math.acos(x)</td>
<td>Arc cosine (x in [-1, 1])</td>
</tr>
<tr>
<td>Math.atan(x)</td>
<td>Arc tangent (x in (-inf, inf))</td>
</tr>
<tr>
<td>Math.abs(x)</td>
<td>Are targets are targets of x</td>
</tr>
<tr>
<td>Math.toRadians(x)</td>
<td>Convert a degree to radians (i.e., return x * Math.PI / 180)</td>
</tr>
<tr>
<td>Math.toDegrees(x)</td>
<td>Convert a radian to degrees (i.e., return x * 180 / Math.PI)</td>
</tr>
<tr>
<td>Math.rint(x)</td>
<td>Round x to the nearest long</td>
</tr>
<tr>
<td>Math.floor(x)</td>
<td>Round x to the nearest long</td>
</tr>
<tr>
<td>Math.ceil(x)</td>
<td>Ceiling of x (as a double)</td>
</tr>
<tr>
<td>Math.min(x, y)</td>
<td>Larger of x and y</td>
</tr>
<tr>
<td>Math.max(x, y)</td>
<td>Absolute value (as a double)</td>
</tr>
</tbody>
</table>

Static Methods

- Static methods are defined in the classes like normal methods.
- They do not operate on object, and don’t have any internal data to deal with.
- Static means the method/variable “belongs” to the class.
- Non-static (instance) means the method/variable “belongs” to the object.

Syntax

```java
ClassName.methodName(parameters)
```

Example:

```java
Math.sqrt(4)
```

Purpose:

To invoke a static method (a method that does not operate on an object) and supply its parameters.
4.6 Strings (Revisited)
- Along with numbers, the most commonly used data type in programming
- Unlike numbers, Strings are objects
- Already learned about `length()` method
  - Empty String (length is 0) → **"**

Number to String
- Converting from number to String is easy
  ```java
  int number = 15;
  String strnum = "" + number;
  ```
- How about the reverse?
  ```java
  int result = Integer.parseInt(someString);
  double result = Double.parseDouble(someString);
  ```
  Similar for all other number types

Parsing
- Only works if entire string is a number
  - "19" // Works for parseInt() & parseDouble()
  - "15.23" // Works for parseDouble() only
  - "123a" // Error for both

substring
- `String subString()` method
  ```java
  String greeting = "Hello, World!";
  String sub = greeting.substring(0, 5);
  // sub is "Hello"
  ```
  Note that the first parameter is the first letter you want (zero based indexing) and the second parameter is the first letter you DON’T want

4.7 Reading Input
- All examples so far have been pretty basic
  - Do not vary, we code in all values when testing
- Most useful programs interact with a user of the terminal (computer) to get data input and perform operations on that data

Output is `System.out`
- You would think Input should then be `System.in`
  - Exists, but not very powerful
  - Takes in only one byte of information at a time, when all characters on the keyboard are 2 bytes!
Scanner

- Can be associated with files, but for now we will learn about keyboard input
- A Scanner object must be created, associated with System.in as follows:

  `Scanner in  = new Scanner(System.in);`

Scanner

- To read numbers from user
  `nextInt(), nextDouble()`
- When called, these methods wait for the user to type input and then hit the enter key. The number inputted is returned.
- Usually, the program will prompt the user for information – give instructions via System.out

Get String

- To get a String, use
  `nextLine()`
- Allows user to type as much as they want. When they hit enter, all characters typed are returned as a String object
- To read just one word (not entire line)
  `next()`

Example

```
Scanner stdin = new Scanner(System.in);
System.out.print("Enter name: ");
String name = stdin.nextLine();
System.out.print("Enter age: ");
int age = stdin.nextInt();
```