Two Types of Types

- So far, we have mainly been dealing with objects, like DrawingGizmo, Window, Triangle, etc. that are:
  1. Specified via a class definition
  2. Instantiated using: `new ConstructorName()`
  3. Written by a programmer (you or someone else) using Java
  4. Known as reference types (because the variable identifier has a value that is a reference address to an object in memory)

- The language has some other things as well:
  1. Built into Java without definition as a class
  2. No constructors used for instantiation
  3. Known as primitive types (because the variable identifier stores the basic value we want directly)

Using Primitive Variables

- Declaring an `int` (or any other primitive type), uses the same syntax as before:
  ```java
  public class Program {
    public static void main(String[] args) {
      int num1;  // Simple, given values.
      num1 = 10;
      int num2 = -77;  // Value computed, based on math expression.
      int num3 = 8 / 4;  
    }
  }
  ```

- When we want to assign a value to the variable identifier, we do not instantiate using a call to a constructor method.
- Instead, we can simply assign a given or computed value.

### Primitive Types in Java

#### Integer types
- `byte` (-128...127)
- `short` (2 bytes) (-32,768...32,767)
- `int` (4 bytes) (-2,147,483,648...2,147,483,647)
- `long` (8 bytes) (-9,223,372,036,854,775,808...9,223,372,036,854,775,807)

#### Real number types
- `float` (4 bytes) 7 decimal digits of accuracy
- `double` (8 bytes) 15 decimal digits of accuracy

#### Other types (to be covered later)
- `boolean` (1 byte) `true/false`
- `char` (2 bytes) single character of text
Using Primitive Variables

- Once you create a primitive variable, it can be used anywhere that values of the same type can be used.
- For example, we can use an `int` variable wherever an `int` itself can be used, such as in arithmetic, or as input to a method that requires an `int`.

```java
public class Program {
    public static void main(String[] args) {
        int size = 1112;
        int halfSize = size / 2;
        Window bigWin = new Window();
        bigWin.setSize(size);
        Window smallWin = new Window();
        smallWin.setSize(halfSize);
    }
}
```

The Java `int` and Its Relatives

**Syntax**

- 1 or more consecutive decimal digits (0-9)
- optional: preceded by + or -
- optional: can separate long numbers using underscore `_`
- examples: 215, -17, +0, 7_234_562

**Unary operator**

- unary negation (e.g., -33)

**Binary infix operators**

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` real division
- `%` remainder

**Postfix operators**

- `++` increment by 1
- `--` decrement by 1

Integer division leads to integer results, with no decimal place. These results are simply truncated, not rounded.

<table>
<thead>
<tr>
<th>Integer divide results</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 / 1 is 10</td>
</tr>
<tr>
<td>10 / 2 is 5</td>
</tr>
<tr>
<td>10 / 3 is 3</td>
</tr>
<tr>
<td>10 / 4 is 2</td>
</tr>
</tbody>
</table>

The Java `float` or `double`

**Syntax**

- a number with a single decimal point
- optional: preceded by + or -
- optional: can separate long numbers using underscore `_`
- optional: can use scientific (E) notation
- examples: 21.5, -1.7, 123_456.789, 5.23E7

**Unary operator**

- unary negation (e.g., -3.3)

**Binary operators**

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` real division
- `%` remainder

Dividing doubles does produce decimal places.

<table>
<thead>
<tr>
<th>Double divide results</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0 / 1.0 is 10.0</td>
</tr>
<tr>
<td>10.0 / 2.0 is 5.0</td>
</tr>
<tr>
<td>10.0 / 3.0 is 3.333..</td>
</tr>
<tr>
<td>10.0 / 4.0 is 2.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Increase/decrease the value by 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code> increment by 1</td>
</tr>
<tr>
<td><code>--</code> decrement by 1</td>
</tr>
</tbody>
</table>
The Java float or double

Syntax
- a number with a single decimal point
- optional: preceded by + or -
- optional: can separate long numbers using underscore _
- optional: can use scientific (E) notation
- examples: 21.5 -1.7 123_456.789 5.23E7

Unary operator
- unary negation (e.g., -3.3)

Infix operators
- addition
- subtraction
- multiplication / real division
- % remainder

Postfix operators
- ++ increment by 1
- -- decrement by 1

Basic Java Division
- Evaluate the following expressions:
  1. int x = 5 / 2;
  2. int x = 5 / 8;
  3. double x = 5.0 / 2.0;
  4. double x = 5.0 / 8.0;

Precedence for Mathematical Operators
- Consider an expression like: 7 * 3 + 2
- Is this: (7 * 3) + 2 = 23?
- Or maybe: 7 * (3 + 2) = 35?
- We can decide using following evaluation rules:
  1. Anything grouped in parentheses goes first.
  2. Within a group, use following precedence (top down order), doing the various operations in sequence:

<table>
<thead>
<tr>
<th>Precedence Table for Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ --</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>* / %</td>
</tr>
<tr>
<td>+ -</td>
</tr>
</tbody>
</table>

3. With equal precedence, go left-to-right

An Example
\[ 1 + 2 * 3 - 4 - 5 / 6 + 9 \% 2 \]

First precedence: * / %
\[ 1 + [2 * 3] - 4 - [5 / 6] + [9 \% 2] \]
\[ 1 + 6 - 4 - 0 + 1 \]

Remember: int division has no decimal precision!

Next precedence: left-to-right
\[ [1 + 6] - 4 - 0 + 1 \]
\[ [7 - 4] - 0 + 1 \]
\[ [3 - 0] + 1 \]
\[ 3 + 1 \]
\[ 4 \]
Mixing Types in Java

- We can **combine** types in an expression like: `10 * 3.6`
- Computation coerces “narrow” types into “wider” ones

```
byte  short  int  long  float  double
```

- The result will be of the **widest** type contained in the basic binary expression we are calculating
- Examples:
  - `7 / 2 == 3 (int)`
  - `7.0 / 2.0 == 3.5 (double)`
  - `7 / 2.0 == 3.5 (double)`
  - `7.0 / 2 == 3.5 (double)`

Safe & Unsafe Casting

- When we turn a narrow type into a wider one, this is considered “safe” in Java
  - Safety here comes from precision
  - For example, since floating-point numbers have decimal places and integers don’t, we don’t lose any information by turning an `int` into a `double` during execution of our code
- Going the other way can lead to problems, however
  - Possible loss of precision
  - Information “thrown away”
- Compiler **won’t allow** this sort of thing:
  ```
  double num1 = 6.0;
  int num2 = num1;  // error!
  ```

Explicit Casting

- Sometimes, we want to **force** a loss of precision, e.g.:
- Rounding down numbers in calculations
- Rounding up to ensure we have enough room for something
- To do so, we can **cast** one expression type to another one, using the following syntax:

```
(primitiveType) mathExpression
```

- An example would be:
  ```
  double dub = 76 / 8.0;  // == 9.5
  int num = (int) dub * 2; // == 18
  ```

Precedence of Casting

- In our example:
  ```
  double dub = 76 / 8.0;  // == 9.5
  int num = (int) dub * 2; // == 18
  ```
  - We get 18 because casting has **higher precedence** than any of the arithmetic operations
  - Thus, 9.5 is turned into an integer, 9, **before** multiplication
- Must be careful to **control casting** properly, using parentheses as needed to get the right results:
  ```
  int num = (int)(4.8 * 2);  // == 9
  int num = (int) 4.8 * 2;    // == 8
  int num = (int) 2 * 4.8;    // error!
  ```
This Week & Next

- **Meetings this week:**
  - Monday, Wednesday: in the CS Lab (16 Wing)
  - Tuesday, Friday: regular classroom

- **Program 01:** available now
  - Due: Monday 10 February, by 11:59 PM

- **Reading assignment:** Chapters 1–2
  - Thursday 06 February, by 12:00 PM

- **Office Hours:** Wing 212
  - Monday/Wednesday/Friday: 11:00 AM–12:00 PM
  - Tuesday: 3:15 PM–4:15 PM