Chapter 4
Fundamental types

• Remember: Java has two kinds of types
  • primitive types
    • e.g. int, float, etc.
  • reference types
    • all object and array types
• What are the differences?
Primitive types

• Three basic categories:
  • Whole numbers
  • Numbers with fractional parts
  • Truth values (i.e. boolean)
## Whole-number types

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>+/- $9.2 \times 10^{18}$</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>+/- ~2 billion</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32768 - 32767</td>
</tr>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>-128 - 127</td>
</tr>
<tr>
<td>char</td>
<td>2 bytes</td>
<td>Unicode</td>
</tr>
</tbody>
</table>
# Fractional-number types

<table>
<thead>
<tr>
<th>name</th>
<th>size</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>+/- $10^{308}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 sig. digits</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>+/- $10^{38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 sig. digits</td>
</tr>
</tbody>
</table>
## Arithmetic operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = -x; )</td>
<td>unary negation</td>
</tr>
<tr>
<td>( z = +x; )</td>
<td>unary positive</td>
</tr>
<tr>
<td>( z = x % y; )</td>
<td>modulus</td>
</tr>
<tr>
<td>( z = x / y; )</td>
<td>division</td>
</tr>
<tr>
<td>( z = x * y; )</td>
<td>multiplication</td>
</tr>
<tr>
<td>( z = x - y; )</td>
<td>subtraction</td>
</tr>
<tr>
<td>( z = x + y; )</td>
<td>addition</td>
</tr>
</tbody>
</table>
Number basics

• We’ve seen string literals; there are also numeric literals:
  • 1234L (long)
  • 1234.0F (float)
  • 1234.0D (double)

• Why is it important to be able to declare the type of a numeric literal?
Arithmetic operations

- Add, subtract, and multiply work as you’d expect
- Two kinds of division
  - Integer division
  - Floating-point division
- Modulus/remainder
Integer vs. FP division

- Floating-point division works as you’d expect:
  - \( 5.0F / 2.0F == 2.5F \)

- Integer division discards the remainder:
  - \( 5 / 2 == 2 \)

- Which is used for given operands?
  - Integer if *all* operands are integers
  - FP if *any* operand is floating-point
Assignment

• Remember the assignment operator?
  • \( x = 5; \)  // “\( x \) gets the value 5”

• Other assignment operators:
  • \( x++; \)  // “increment \( x \), evaluate to old \( x \)”
  • \( ++x; \)  // “increment \( x \), evaluate to new \( x \)”
  • \( x+=5; \)  // “increment \( x \) by 5”

• Also:  \(*=, -=, /=\)
Operator precedence

• This slide is simple: when in doubt, use parentheses!
Things to lose sleep over

- Overflow
  - Integer types have limited range
- Rounding errors
- Division by zero
  - Unfortunately, still undefined. May crash your program.
Overflow
Rounding errors

- Floating-point arithmetic is not totally precise.
  - Remember, FP types only have a finite number of significant digits
  - Good enough for most applications, maybe not for all
- Rounding errors can be magnified in a sequence of operations
- Financial institutions, etc., use (slow but) precise classes for FP math
Casting

• You can treat an expression as if it has a different type:
  • $(typeName)exp$

• Example: $(int)4.2F$

• Why do this? What are the tradeoffs?
String

• Strings are very useful
• Many of the methods in String will make your life easier
  • Note that none of these modifies the base String -- String is *immutable*.
• You can use the + operator to concatenate Strings
String s1 = "x";
String s2 = s1 + s1;
String s3 = "banana";
String s4 = s3.substring(1, 3);
String s5 = s4.replace('a', 'o');
Wrapper classes

• What’s a primitive type?
• What’s a reference type?
• Remember that Java maintains a divide between primitive types and reference types. Wrapper classes provide a way around this!
Wrapper classes

• One for each primitive type: e.g. `Integer` for `int`, etc.

• Can make an `Integer` from an `int`, and can get the `int` value from an `Integer`.

• All wrapper classes are *immutable*, just like `String`.

• Why might we use these?
Constants: why?

• What does the following line of code mean?
  • \( x = 42 \times y; \)

• What does “42” mean? Why?

• Constants make programs easier to read and maintain.
final locals

- final int NUM_SHELVES = 42;
- x = NUM_SHELVES * y;
- Can only be assigned to once!
Constants

• Why are we interested in constants?
• How can we use named constants in our Java programs?
Constants: Why?

- Even if we choose good names for local variables, “magic numbers” can make our programs worse
  - harder to understand
  - harder to maintain

- Consider:
  - products = 42 * perShelf; // vs.
  - products = NUM_SHELVES * perShelf;
Constants: How?

```java
public int totalProducts(int pps) {
    final int NUM_SHELVES = 42;

    return NUM_SHELVES * pps;
}
```
Constants: How?

```java
public int totalProducts(int pps) {
    final int NUM_SHELVES = 42;
    NUM_SHELVES = 36;
    return NUM_SHELVES * pps;
}
```
public int totalProducts(int pps) {
    final int NUM_SHELVES = 42;
    NUM_SHELVES = 36;
    return NUM_SHELVES * pps;
}
Constants: How?

```java
public int totalProducts(int pps) {
    final int NUM_SHELVES = 42;
    NUM_SHELVES = 36;
    return NUM_SHELVES * pps;
}
```

Final variables can only get one value!
A limitation of final locals

• What if you want to use the same constant in multiple methods?
• Is there a good way to do this?
• Is there any way to do this at all?
Well....

- You could just declare a `final` local in each method that is to use the constant.
- That’s sort of clunky.
- Duplicating constant declarations in each method is tedious and error-prone.
- Why are we using constants in the first place?
If some programming task is *tedious and error-prone*, it probably indicates *bad design, bad style, or both!*
A better solution

• **final** data members

• Two kinds:
  • **final** instance variables
  • **static final** variables

• Different applications for each
final instance vars

• These correspond to something that can’t change once the object is created

• “Factory-installed options”

• Examples:
  • “Parents” of a Dog instance
  • “Capacity” of Mug instance
final instance vars

accessSpecifier final type id;

or

accessSpecifier final type id = expression;
class Beer {
    public final Date expiration;
    /* ... */

    public void consume() {
        /* ... */
    }
}
static final fields

• Sometimes, it makes sense for a constant to belong to a class instead of to an instance

• Fields that belong to a class are called static fields or class fields

• Example:
  • CashRegister.NICKEL_VALUE = 0.05;
  • Can you think of another static field we’ve seen in class so far?
Other examples

• Math.PI
• Math.E
• Can you think of any other useful constants that should be static?
static final vars

accessSpecifier static final type id = expression;

public static final int COURSE = 302;
Constants wrap-up

- Use constants instead of “magic numbers” whenever possible
  - `final` local variables when a constant is only needed in *one method*
  - `final` instance fields when a field cannot be changed once an object is created
  - `static final` fields when a constant can be shared between methods in different classes (or between every instance of one class)
Static methods

• Remember that class constants are fields that *belong to a class rather than to an instance*

• We use the `static` reserved word to indicate this

• We can also declare *methods* that belong to a class rather than to an instance.
Static methods belong to a class, not to an instance.
Static methods

• Static methods are those that don’t operate on any particular instance of the class in which they’re declared

• These have no implicit parameter; you can’t refer to this!

• Why might we want to declare such a method?
Examples

• Factory methods: methods that return references to newly-created instances.

• Utility methods: methods that operate on primitive types.
  • e.g. Math.sqrt(), Math.pow(), etc.

• Accessor and mutator methods for non-final static fields.
  • e.g. System.setIn()
public static void main(String[] args) {
    System.out.println("Hello, world!");
}

main is a static method.
(println is an instance method, but System.out is a static field.
Confused yet?)
User input

• Remember `System.out`?
• It’s a static field of class `System`
• and a reference to an instance of class `OutputStream`
• that sends output to the console

• `System` has another field for input from the console
User input

- `System.in` is a reference to an instance of class `InputStream`
- `InputStream` has methods to read one byte or a sequence of bytes at a time.
  - e.g. `read()`, `read(byte[])`, etc.
User input

• However, it isn’t convenient to deal in bytes!

• For example:
  
  • Hello, world!\n
  • 72 101 108 108 111 44 32 119 111
     114 108 100 33 10
The Scanner class interacts with an InputStream and provides an improved interface
Scanner in = new Scanner(System.in);
System.out.print("Enter your name: ");
String name = in.next();
Scanner methods

• `next()` and `nextWord()` return next *token* (i.e., until space)
• `nextInt()` interprets next token as an `int`, returns `int` value
• `nextDouble()`, etc., are similar to `nextInt()`
• `nextLine()` returns the next line