A First Look At Java
Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a Java language system
Example

- Colored points on the screen
- What data goes into making one?
  - Coordinates
  - Color
- What should a point be able to do?
  - Move itself
  - Report its position
My x: 5
My y: 20
My color: black
Things I can do:
move
report x
report y

My x: 17
My y: 25
My color: light grey
Things I can do:
move
report x
report y

My x: 20
My y: 10
My color: dark grey
Things I can do:
move
report x
report y
Java Terminology

- Each point is an **object**
- Each includes three **fields**
- Each has three **methods**
- Each is an **instance** of the same **class**

Thought bubble:
- My x: 10
- My y: 50
- My color: 'black'
- Things I can do: move
- report x
- report y
Object-Oriented Style

- Solve problems using objects: little bundles of data that know how to do things to themselves
- Not the computer knows how to move the point, but rather the point knows how to move itself
- Object-oriented languages make this way of thinking and programming easier
public class Point {
    private int x, y;
    private Color myColor;

    public int currentX() {
        return x;
    }

    public int currentY() {
        return y;
    }

    public void move(int newX, int newY) {
        x = newX;
        y = newY;
    }
}
Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a Java language system
Primitive Types We Will Use

- **int**: $-2^{31}..2^{31}-1$, written the usual way
- **char**: $0..2^{16}-1$, written 'a', '
', etc., using the Unicode character set
- **double**: IEEE 64-bit standard, written in decimal (1.2) or scientific (1.2e-5, 1e3)
- **boolean**: `true` and `false`
- **Oddities**: `void` and `null`
Primitive Types We Won’t Use

- **byte**: \(-2^7..2^7-1\)
- **short**: \(-2^{15}..2^{15}-1\)
- **long**: \(-2^{63}..2^{63}-1\), written with trailing **L**
- **float**: IEEE 32-bit standard, written with trailing **F** \((1.2e-5, 1e3)\)
Constructed Types

**constructed types are all reference types:** they are references to objects

- Any class name, like `Point`
- Any interface name (Chapter 15)
- Any array type, like `Point[]` or `int[]` (Chapter 14)
Strings

- Predefined but not primitive: a class `String`
- A string of characters enclosed in double-quotes works like a string constant
- But it is actually an instance of the `String` class, and object containing the given string of characters
A String Object

"Hello there"

My data: Hello there
My length: 11
Things I can do:
- report my length
- report my ith char
- make an uppercase version of myself
- etc.

Chapter Thirteen
Modern Programming Languages, 2nd ed.
Numeric Operators

- **int**: +, −, *, /, %, unary −

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+2*3</td>
<td>7</td>
</tr>
<tr>
<td>15/7</td>
<td>2</td>
</tr>
<tr>
<td>15%7</td>
<td>1</td>
</tr>
<tr>
<td>-(5*5)</td>
<td>-25</td>
</tr>
</tbody>
</table>

- **double**: +, −, *, /, unary −

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.0*2.0</td>
<td>26.0</td>
</tr>
<tr>
<td>15.0/7.0</td>
<td>2.142857142857143</td>
</tr>
</tbody>
</table>
### Concatenation

The `+` operator has special overloading and coercion behavior for the class `String`.

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;123&quot;+&quot;456&quot;</td>
<td>&quot;123456&quot;</td>
</tr>
<tr>
<td>&quot;The answer is &quot; + 4</td>
<td>&quot;The answer is 4&quot;</td>
</tr>
<tr>
<td>&quot;&quot; + (1.0/3.0)</td>
<td>&quot;0.3333333333333333&quot;</td>
</tr>
<tr>
<td>1+&quot;2&quot;</td>
<td>&quot;12&quot;</td>
</tr>
<tr>
<td>&quot;1&quot;+2+3</td>
<td>&quot;123&quot;</td>
</tr>
<tr>
<td>1+2+&quot;3&quot;</td>
<td>&quot;33&quot;</td>
</tr>
</tbody>
</table>
Comparisons

- The usual comparison operators <, <=, >=, and >, on numeric types
- Equality == and inequality != on any type, including double (unlike ML)

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;=2</td>
<td>true</td>
</tr>
<tr>
<td>1==2</td>
<td>false</td>
</tr>
<tr>
<td>true!=false</td>
<td>true</td>
</tr>
</tbody>
</table>
Boolean Operators

- `&&` and `||`, short-circuiting, like ML’s `andalso` and `orelse`
- `!`, like ML’s `not`
- `a?b:c`, like ML’s `if a then b else c`

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;=2 &amp;&amp; 2&lt;=3</td>
<td>true</td>
</tr>
<tr>
<td>1&lt;2</td>
<td></td>
</tr>
<tr>
<td>1&lt;2 ? 3 : 4</td>
<td>3</td>
</tr>
</tbody>
</table>
Operators With Side Effects

- An operator has a *side effect* if it changes something in the program environment, like the value of a variable or array element.
- In ML, and in Java so far, we have seen only *pure* operators—no side effects.
- Now: Java operators with side effects.
Assignment

- $a=b$: changes $a$ to make it equal to $b$
- Assignment is an important part of what makes a language *imperative*
Rvalues and Lvalues

■ Why does \( a=1 \) make sense, but not \( 1=a \)?
■ Expressions on the right must have a value: \( a, 1, a+1, f() \) (unless \texttt{void}), etc.
■ Expressions on the left must have memory locations: \( a \) or \( d[2] \), but not \( 1 \) or \( a+1 \)
■ These two attributes of an expression are sometimes called the \textit{rvalue} and the \textit{lvalue}
Rvalues and Lvalues

In most languages, the context decides whether the language will use the rvalue or the lvalue of an expression

A few exceptions:
- Bliss: \texttt{x := .y}
- ML: \texttt{x := !y} (both of type \texttt{'a ref})
More Side Effects

- **Compound assignments**

<table>
<thead>
<tr>
<th>Long Java Expression</th>
<th>Short Java Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = a + b )</td>
<td>( a += b )</td>
</tr>
<tr>
<td>( a = a - b )</td>
<td>( a -= b )</td>
</tr>
<tr>
<td>( a = a \ast b )</td>
<td>( a \ast = b )</td>
</tr>
</tbody>
</table>

- **Increment and decrement**

<table>
<thead>
<tr>
<th>Long Java Expression</th>
<th>Short Java Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = a + 1 )</td>
<td>( a ++ )</td>
</tr>
<tr>
<td>( a = a - 1 )</td>
<td>( a -- )</td>
</tr>
</tbody>
</table>
Values And Side Effects

- Side-effecting expressions have both a value and a side effect
- Value of \( x = y \) is the value of \( y \); side-effect is to change \( x \) to have that value

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a + (x=b) + c )</td>
<td>the sum of ( a ), ( b ) and ( c )</td>
<td>changes the value of ( x ), making it equal to ( b )</td>
</tr>
<tr>
<td>( (a=d) + (b=d) + (c=d) )</td>
<td>three times the value of ( d )</td>
<td>changes the values of ( a ), ( b ) and ( c ), making them all equal to ( d )</td>
</tr>
<tr>
<td>( a=b=c )</td>
<td>the value of ( c )</td>
<td>changes the values of ( a ) and ( b ), making them equal to ( c )</td>
</tr>
</tbody>
</table>
Pre and Post

Values from increment and decrement depend on placement

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>a++</td>
<td>the old value of a</td>
<td>adds one to a</td>
</tr>
<tr>
<td>++a</td>
<td>the new value of a</td>
<td>adds one to a</td>
</tr>
<tr>
<td>a--</td>
<td>the old value of a</td>
<td>subtracts one from a</td>
</tr>
<tr>
<td>--a</td>
<td>the new value of a</td>
<td>subtracts one from a</td>
</tr>
</tbody>
</table>
## Instance Method Calls

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.length()</code></td>
<td>the length of the String <code>s</code></td>
</tr>
<tr>
<td><code>s.equals(r)</code></td>
<td><code>true</code> if <code>s</code> and <code>r</code> are equal, <code>false</code> otherwise</td>
</tr>
<tr>
<td><code>r.equals(s)</code></td>
<td>same</td>
</tr>
<tr>
<td><code>r.toUpperCase()</code></td>
<td>A String object that is an uppercase version of the String <code>r</code></td>
</tr>
<tr>
<td><code>r.charAt(3)</code></td>
<td>the char value in position 3 in the String <code>r</code> (that is, the fourth character)</td>
</tr>
<tr>
<td><code>r.toUpperCase().charAt(3)</code></td>
<td>the char value in position 3 in the uppercase version of the String <code>r</code></td>
</tr>
</tbody>
</table>
Class Method Calls

- *Class methods* define things the class itself knows how to do—not objects of the class
- The class just serves as a labeled namespace
- Like ordinary function calls in non-object-oriented languages

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String.valueOf(1==2)</code></td>
<td>&quot;false&quot;</td>
</tr>
<tr>
<td><code>String.valueOf(5*5)</code></td>
<td>&quot;25&quot;</td>
</tr>
<tr>
<td><code>String.valueOf(1.0/3.0)</code></td>
<td>&quot;0.3333333333333333&quot;</td>
</tr>
</tbody>
</table>
Method Call Syntax

Three forms:

- Normal instance method call:
  \[ \text{method-call} ::= \text{reference-expression}.\text{method-name} \]
  \[ (\text{parameter-list}) \]

- Normal class method call
  \[ \text{method-call} ::= \text{class-name}.\text{method-name} \]
  \[ (\text{parameter-list}) \]

- Either kind, from within another method of the same class
  \[ \text{method-call} ::= \text{method-name}(\text{parameter-list}) \]
Object Creation Expressions

- To create a new object that is an instance of a given class
  
  \[
  \text{<creation-expression>} : = \text{new } \text{<class-name>}
  \]

- Parameters are passed to a constructor—like a special instance method of the class

<table>
<thead>
<tr>
<th>Java Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new String()</code></td>
<td>a new <code>String</code> of length zero</td>
</tr>
<tr>
<td><code>new String(s)</code></td>
<td>a new <code>String</code> that contains a copy of <code>String s</code></td>
</tr>
<tr>
<td><code>new String(chars)</code></td>
<td>a new <code>String</code> that contains the <code>char</code> values from the array</td>
</tr>
</tbody>
</table>
No Object Destruction

- Objects are created with `new`
- Objects are never explicitly destroyed or deallocated
- Garbage collection (chapter 14)
General Operator Info

- All left-associative, except for assignments
- 15 precedence levels
  - Some obvious: * higher than +
  - Others less so: < higher than !=
  - Use parentheses to make code readable

- Many coercions
  - `null` to any reference type
  - Any value to `String` for concatenation
  - One reference type to another sometimes (Chapter 15)
Numeric Coercions

Numeric coercions (for our types):

- `char` to `int` before any operator is applied (except string concatenation)
- `int` to `double` for binary ops mixing them

<table>
<thead>
<tr>
<th>Java expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'+'b'</td>
<td>195</td>
</tr>
<tr>
<td>1/3</td>
<td>0</td>
</tr>
<tr>
<td>1/3.0</td>
<td>0.3333333333333333</td>
</tr>
<tr>
<td>1/2+0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1/(2+0.0)</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Boxing and Unboxing Coercions

- Preview: Java supports coercions between
  - most of the primitive types (including \texttt{int}, \texttt{char}, \texttt{double}, and \texttt{boolean}), and
  - corresponding predefined reference types (\texttt{Integer}, \texttt{Character}, \texttt{Double}, and \texttt{Boolean})

- More about these coercions in Chapter 15
Statements

- That’s it for expressions
- Next, statements:
  - Expression statements
  - Compound statements
  - Declaration statements
  - The `if` statement
  - The `while` statement
  - The `return` statement

- Statements are executed for side effects: an important part of imperative languages
**Expression Statements**

\[
\text{<expression-statement>} \ : \ := \ \text{<expression>} \ ;
\]

- Any expression followed by a semicolon
- Value of the expression, if any, is discarded
- Java does not allow the expression to be something without side effects, like \texttt{x==y}

<table>
<thead>
<tr>
<th>Java Statement</th>
<th>Equivalent Command in English</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{speed = 0;}</td>
<td>Store a 0 in \texttt{speed}.</td>
</tr>
<tr>
<td>\texttt{a++;}</td>
<td>Increase the value of \texttt{a} by 1.</td>
</tr>
<tr>
<td>\texttt{inTheRed = cost &gt; balance;}</td>
<td>If \texttt{cost} is greater than \texttt{balance}, set \texttt{inTheRed} to \texttt{true}, otherwise to \texttt{false}.</td>
</tr>
</tbody>
</table>
Compound Statements

\[
<\text{compound-statement}> ::= \{ <\text{statement-list}> \}
\]
\[
<\text{statement-list}> ::= <\text{statement}> <\text{statement-list}> | <\text{empty}>
\]

- Do statements in order
- Also serves as a block for scoping

<table>
<thead>
<tr>
<th>Java Statement</th>
<th>Equivalent Command in English</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>Store a zero in a, then store a 1 in b.</td>
</tr>
<tr>
<td>a = 0;</td>
<td></td>
</tr>
<tr>
<td>b = 1;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>Increment a, then increment b, then increment c.</td>
</tr>
<tr>
<td>a++;</td>
<td></td>
</tr>
<tr>
<td>b++;</td>
<td></td>
</tr>
<tr>
<td>c++;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>{ }</td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>
Declaration Statements

\[
\langle \text{declaration-statement} \rangle ::= \langle \text{declaration} \rangle ;
\]

\[
\langle \text{declaration} \rangle ::= \langle \text{type} \rangle \ \langle \text{variable-name} \rangle \\
| \ \langle \text{type} \rangle \ \langle \text{variable-name} \rangle = \langle \text{expression} \rangle
\]

- Block-scoped definition of a variable

| boolean done = false;         | Define a new variable named \textit{done} of type \texttt{boolean}, and initialize it to \texttt{false}. |
| Point p;                      | Define a new variable named \textit{p} of type \texttt{Point}. (Do not initialize it.) |
| \{                            | Swap the values of the integer variables \textit{a} and \textit{b}. |
|  \hspace{1em} int temp = a;  |                                              |
|  a = b;                       |                                              |
|  b = temp;                    |                                              |
| \}                            |                                              |
The **if** Statement

```
<if-statement> ::= if (<expression>) <statement>
| if (<expression>) <statement> else <statement>
```

*Dangling else resolved in the usual way*

<table>
<thead>
<tr>
<th>Java Statement</th>
<th>Equivalent Command in English</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (i &gt; 0) i--;</code></td>
<td>Decrement i, but only if it is greater than zero.</td>
</tr>
<tr>
<td><code>if (a &lt; b) b -= a; else a -= b;</code></td>
<td>Subtract the smaller of a or b from the larger.</td>
</tr>
<tr>
<td><code>if (reset) {</code></td>
<td>If reset is <strong>true</strong>, zero out a and b and then set reset to <strong>false</strong>.</td>
</tr>
<tr>
<td>a = b = 0;</td>
<td></td>
</tr>
<tr>
<td>reset = false;</td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>
The \textbf{while} Statement

\texttt{<while-statement>} ::= \texttt{while \(<expression>\)} \texttt{<statement>}

\begin{itemize}
  \item Evaluate expression; if false do nothing
  \item Otherwise execute statement, then repeat
  \item Iteration is another hallmark of imperative languages
  \item (Note that this iteration would not make sense without side effects, since the value of the expression must change)
  \item Java also has \texttt{do} and \texttt{for} loops
\end{itemize}
<table>
<thead>
<tr>
<th>Java Statement</th>
<th>Equivalent Command in English</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while (a&lt;100) a+=5;</code></td>
<td>As long as <code>a</code> is less than 100, keep adding 5 to <code>a</code>.</td>
</tr>
</tbody>
</table>
| `while (a!=b)`  
  `if (a < b) b -= a;`  
  `else a -= b;` | Subtract the smaller of `a` or `b` from the larger, over and over until they are equal. (This is Euclid's algorithm for finding the GCD of two positive integers.) |
| `while (time>0) {`  
  `simulate();`  
  `time--;`  
  `}` | As long as `time` is greater than zero, call the `simulate` method of the current class and then decrement `time`. |
| `while (true) work();` | Call the `work` method of the current class over and over, forever. |
The `return` Statement

\[
\langle \text{return-statement} \rangle ::= \text{return} \ \langle \text{expression} \rangle ; \\
| \ \text{return} ;
\]

- Methods that return a value must execute a return statement of the first form
- Methods that do not return a value (methods with return type `void`) may execute a return statement of the second form
Outline

■ 13.2 Thinking about objects
■ 13.3 Simple expressions and statements
■ 13.4 Class definitions
■ 13.5 About references and pointers
■ 13.6 Getting started with a Java language system
Class Definitions

- We have enough expressions and statements
- Now we will use them to make a definition of a class
- Example: `ConsCell`, a class for building linked lists of integers like ML’s `int list` type
/**
 * A ConsCell is an element in a linked list of ints.
 */

public class ConsCell {
    private int head; // the first item in the list
    private ConsCell tail; // rest of the list, or null

    /**
     * Construct a new ConsCell given its head and tail.
     * @param h the int contents of this cell
     * @param t the next ConsCell in the list, or null
     */
    public ConsCell(int h, ConsCell t) {
        head = h;
        tail = t;
    }
}

Note comment forms, public and private, field definitions.
Note constructor definition: access specifier, class name, parameter list, compound statement
/**
 * Accessor for the head of this ConsCell.
 * @return the int contents of this cell
 */
public int getHead() {
    return head;
}

/**
 * Accessor for the tail of this ConsCell.
 * @return the next ConsCell in the list, or null
 */
public ConsCell getTail() {
    return tail;
}

Note method definitions: access specifier, return type, method name, parameter list, compound statement
```java
/**
 * Mutator for the tail of this ConsCell.
 * @param t the new tail for this cell
 */
public void setTail(ConsCell t) {
    tail = t;
}
```

Note: this *mutator* gives a way to ask a *ConsCell* to change its own tail link. (Not like anything we did with lists in ML!) This method is useful for some of the exercises at the end of the chapter.
Using **ConsCell**

```plaintext
val a = [];  ConsCell a = null;
val b = 2::a; ConsCell b = new ConsCell(2,a);
val c = 1::b; ConsCell c = new ConsCell(1,b);
```

- Like consing up a list in ML
- But a Java list should be object-oriented: where ML applies :: to a list, our Java list should be able to cons onto itself
- And where ML applies `length` to a list, Java lists should compute their own length
- So we can’t use `null` for the empty list
/**
 * An IntList is a list of ints.
 */
public class IntList {
    private ConsCell start; // list head, or null

    /**
     * Construct a new IntList given its first ConsCell.
     * @param s the first ConsCell in the list, or null
     */
    public IntList(ConsCell s) {
        start = s;
    }

    An IntList contains a reference to a list of ConsCell objects, which will be null if the list is empty.
/**
 * Cons the given element h onto us and return the resulting IntList.
 * @param h the head int for the new list
 * @return the IntList with head h, and us as tail
 */

public IntList cons (int h) {
    return new IntList(new ConsCell(h,start));
}

An IntList knows how to cons things onto itself. It does not change, but it returns a new IntList with the new element at the front.
/**
 * Get our length.
 * @return our int length
 */

public int length() {
    int len = 0;
    ConsCell cell = start;
    while (cell != null) { // while not at end of list
        len++;
        cell = cell.getTail();
    }

    return len;
}

An IntList knows how to compute its length
**Using IntList**

ML:

```ml
val a = nil;
val b = 2::a;
val c = 1::b;
val x = (length a) + (length b) + (length c);
```

Java:

```java
IntList a = new IntList(null);
IntList b = a.cons(2);
IntList c = b.cons(1);
int x = a.length() + b.length() + c.length();
```
Outline

■ 13.2 Thinking about objects
■ 13.3 Simple expressions and statements
■ 13.4 Class definitions
■ 13.5 About references and pointers
■ 13.6 Getting started with a Java language system
What Is A Reference?

- A reference is a value that uniquely identifies a particular object
  
  ```java
  public IntList(ConsCell s) {
      start = s;
  }
  ```

- What gets passed to the `IntList` constructor is not an object—it is a reference to an object

- What gets stored in `start` is not a copy of an object—it is a reference to an object, and no copy of the object is made
Pointers

- If you have been using a language like C or C++, there is an easy way to think about references: a reference is a pointer.
- That is, a reference is the address of the object in memory.
- Java language systems can implement references this way.
But I Thought…

- It is sometimes said that Java is like C++ without pointers
- True from a certain point of view
- C and C++ expose the address nature of pointers (e.g. in pointer arithmetic)
- Java programs can’t tell how references are implemented: they are just values that uniquely identify a particular object
C++ Comparison

- A C++ variable can hold an object or a pointer to an object. There are two selectors:
  - `a->x` selects method or field `x` when `a` is a pointer to an object
  - `a.x` selects `x` when `a` is an object

- A Java variable cannot hold an object, only a reference to an object. Only one selector:
  - `a.x` selects `x` when `a` is a reference to an object
### Comparison

<table>
<thead>
<tr>
<th>C++</th>
<th>Equivalent Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntList* <code>p;</code></td>
<td>IntList <code>p;</code></td>
</tr>
<tr>
<td><code>p = new IntList(0);</code></td>
<td><code>p = new IntList(null);</code></td>
</tr>
<tr>
<td><code>p-&gt;length();</code></td>
<td><code>p.length();</code></td>
</tr>
<tr>
<td><code>p = q;</code></td>
<td><code>p = q;</code></td>
</tr>
<tr>
<td>IntList <code>p(0);</code></td>
<td>No equivalent.</td>
</tr>
<tr>
<td><code>p.length();</code></td>
<td></td>
</tr>
<tr>
<td><code>p = q;</code></td>
<td></td>
</tr>
</tbody>
</table>
Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a Java language system
Text Output

- A predefined object: System.out
- Two methods: `print(x)` to print x, and `println(x)` to print x and start a new line
- Overloaded for all parameter types

```java
System.out.println("Hello there");
System.out.print(1.2);
```
Printing An **IntList**

```java
/**
 * Print ourself to System.out.
 */

public void print() {
    System.out.print("[");
    ConsCell a = start;
    while (a != null) {
        System.out.print(a.getHead());
        a = a.getTail();
        if (a != null) System.out.print(",");
    }
    System.out.println("]");
}
```

Added to the **IntList** class definition, this method gives an **IntList** the ability to print itself out.
The **main** Method

- A class can have a **main** method like this:
  ```java
  public static void main(String[] args) {
    ...
  }
  ```

- This will be used as the starting point when the class is run as an application

- Keyword **static** makes this a class method; use sparingly!
A Driver Class

public class Driver {
    public static void main(String[] args) {
        IntList a = new IntList(null);
        IntList b = a.cons(2);
        IntList c = b.cons(1);
        int x = a.length() + b.length() + c.length();
        a.print();
        b.print();
        c.print();
        System.out.println(x);
    }
}

Compiling The Program

- Three classes to compile, in three files:
  - ConsCell.java, IntList.java, and Driver.java
- (File name = class name plus .java—watch capitalization!)
- Compile with the command javac
  - They can be done one at a time
  - Or, javac Driver.java gets them all
Running The Program

- Compiler produces `.class` files
- Use the Java launcher (`java` command) to run the `main` method in a `.class` file

```
C:\demo> java Driver
[]
[2]
[1,2]
3
```