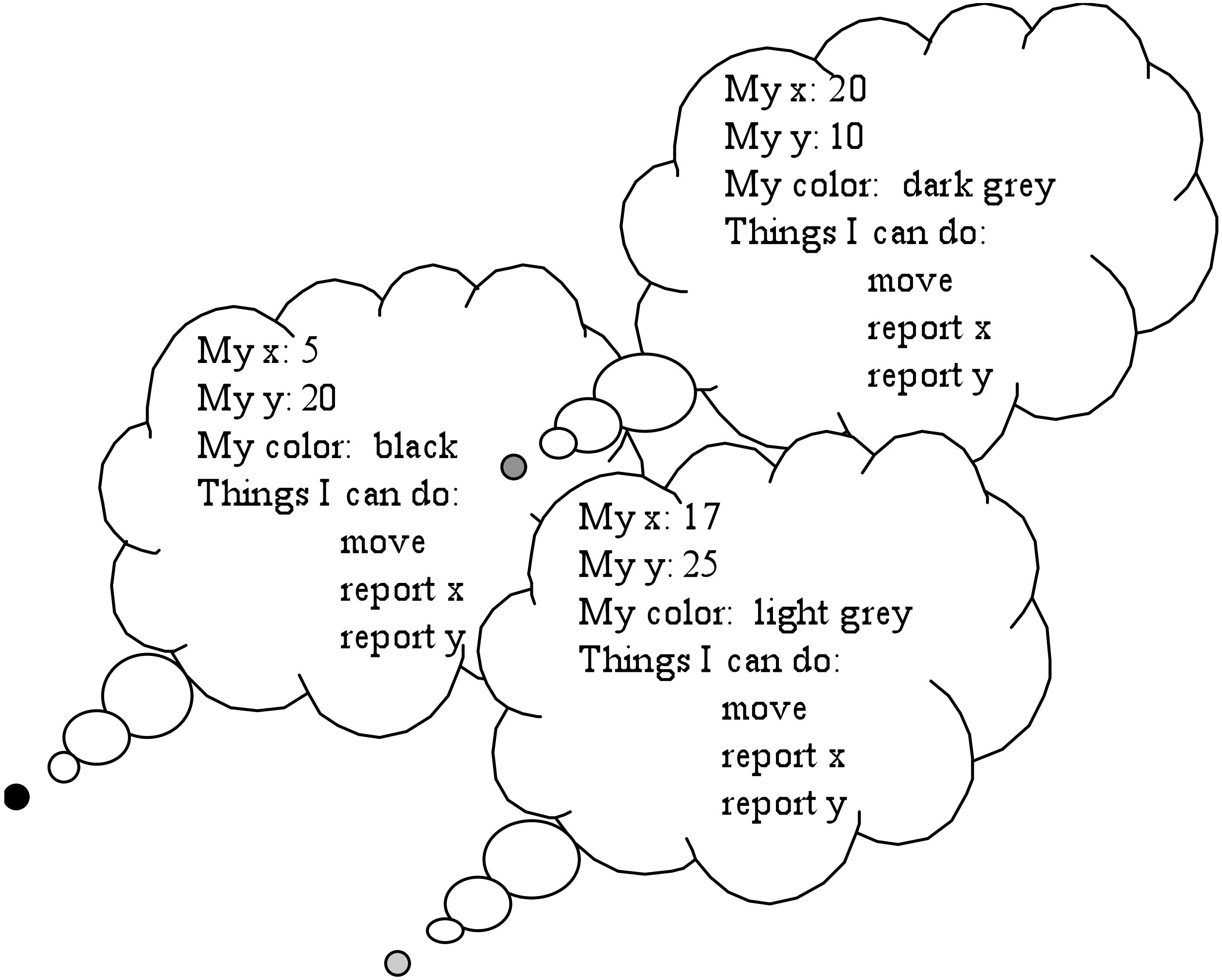
A First Look At Java

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

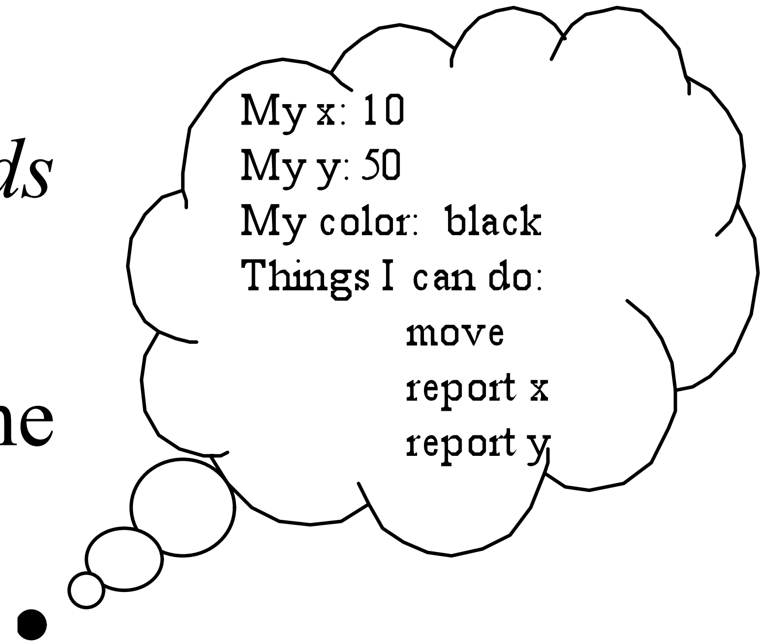
Example

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

Java Terminology

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



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

Java Class Definitions: A Peek

```
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;
  }
}
```

field definitions

method definitions

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'`**, **`'\n'`**, 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"`

Numeric Operators

- **int**: **+**, **-**, *****, **/**, **%**, unary **-**

Java Expression	Value
`1+2*3`	`7`
`15/7`	`2`
`15%7`	`1`
`-(5*5)`	`-25`

- **double**: **+**, **-**, *****, **/**, unary **-**

Java Expression	Value
`13.0*2.0`	`26.0`
`15.0/7.0`	`2.142857142857143`

Concatenation

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

Java Expression	Value
`"123"+"456"`	`"123456"`
`"The answer is " + 4`	`"The answer is 4"`
`"" + (1.0/3.0)`	`"0.3333333333333333"`
`1+"2"`	`"12"`
`"1"+2+3`	`"123"`
`1+2+"3"`	`"33"`

Comparisons

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

Java Expression	Value
`1<=2`	`true`
`1==2`	`false`
`true!=false`	`true`

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`**

Java Expression	Value
`1<=2 && 2<=3`	`true`
`1<2 || 1>2`	`true`
`1<2 ? 3 : 4`	`3`

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 **`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 *rvalue* and the *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: `x := .y`
  - ML: `x := !y` (both of type `'a ref`)

More Side Effects

- Compound assignments

Long Java Expression	Short Java Expression
`a=a+b`	**`a+=b`**
`a=a-b`	**`a-=b`**
`a=a*b`	**`a*=b`**

- Increment and decrement

Long Java Expression	Short Java Expression
`a=a+1`	**`a++`**
`a=a-1`	**`a--`**

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

Java Expression	Value	Side Effect
`a+(x=b)+c`	the sum of **`a`**, **`b`** and **`c`**	changes the value of **`x`**, making it equal to **`b`**
`(a=d)+(b=d)+(c=d)`	three times the value of **`d`**	changes the values of **`a`**, **`b`** and **`c`**, making them all equal to **`d`**
`a=b=c`	the value of **`c`**	changes the values of **`a`** and **`b`**, making them equal to **`c`**

Pre and Post

- Values from increment and decrement depend on placement

Java Expression	Value	Side Effect
`a++`	the old value of **`a`**	adds one to **`a`**
`++a`	the new value of **`a`**	adds one to **`a`**
`a--`	the old value of **`a`**	subtracts one from **`a`**
`--a`	the new value of **`a`**	subtracts one from **`a`**

Instance Method Calls

Java Expression	Value
`s.length()`	the length of the **`String s`**
`s.equals(r)`	**`true`** if **`s`** and **`r`** are equal, **`false`** otherwise
`r.equals(s)`	same
`r.toUpperCase()`	A **`String`** object that is an uppercase version of the **`String r`**
`r.charAt(3)`	the **`char`** value in position 3 in the **`String r`** (that is, the fourth character)
`r.toUpperCase().charAt(3)`	the **`char`** value in position 3 in the uppercase version of the **`String r`**

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

Java Expression	Value
`String.valueOf(1==2)`	`"false"`
`String.valueOf(5*5)`	`"25"`
`String.valueOf(1.0/3.0)`	`"0.3333333333333333"`

Method Call Syntax

- Three forms:
  - Normal instance method call:

    <method-call> `::=` *<reference-expression>*`.`*<method-name>*
    `(`*<parameter-list>*`)`
  - Normal class method call

    <method-call> `::=` *<class-name>*`.`*<method-name>*
    `(`*<parameter-list>*`)`
  - Either kind, from within another method of the same class

    <method-call> `::=` *<method-name>*`(`*<parameter-list>*`)`

Object Creation Expressions

- To create a new object that is an instance of a given class

  <creation-expression> `::=` **`new`** *<class-name>* **`(`***<parameter-list>***`)`**

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

Java Expression	Value
`new String()`	a new **`String`** of length zero
`new String(s)`	a new **`String`** that contains a copy of **`String s`**
`new String(chars)`	a new **`String`** that contains the **`char`** values from the array

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

Java expression	value
`'a'+'b'`	`195`
`1/3`	`0`
`1/3.0`	`0.3333333333333333`
`1/2+0.0`	`0.0`
`1/(2+0.0)`	`0.5`

Boxing and Unboxing Coercions

- Preview: Java supports coercions between
  - most of the primitive types (including **`int`**, **`char`**, **`double`**, and **`boolean`**), and
  - corresponding predefined reference types (**`Integer`**, **`Character`**, **`Double`**, and **`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

<*expression-statement*> `::=` <*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 **`x==y`**

Java Statement	Equivalent Command in English
`speed = 0;`	Store a 0 in **`speed`**.
`a++;`	Increase the value of **`a`** by 1.
`inTheRed = cost > balance;`	If **`cost`** is greater than **`balance`**, set **`inTheRed`** to **`true`**, otherwise to **`false`**.

Compound Statements

<compound-statement> **::=** **{** *<statement-list>* **}**

< statement-list> **::=** *<statement>* *<statement-list>* | *<empty>*

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

Java Statement	Equivalent Command in English

`{`
`  a = 0;`
`  b = 1;`
`}`
	Store a zero in **`a`**, then store a 1 in **`b`**.

`{`
`  a++;`
`  b++;`
`  c++;`
`}`
	Increment **`a`**, then increment **`b`**, then increment **`c`**.

`{ }`	Do nothing.

Declaration Statements

<declaration-statement> **::=** *<declaration>* **;**
<declaration> **::=** *<type>* *<variable-name>*
| *<type>* *<variable-name>* **=** *<expression>*

- Block-scoped definition of a variable

`boolean done = false;`	Define a new variable named **`done`** of type **`boolean`**, and initialize it to **`false`**.
`Point p;`	Define a new variable named **`p`** of type **`Point`**. (Do not initialize it.)
`{ int temp = a; a = b; b = temp; }`	Swap the values of the integer variables **`a`** and **`b`**.

The **`if`** Statement

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

- Dangling else resolved in the usual way

Java Statement	Equivalent Command in English
`if (i > 0) i--;`	Decrement **`i`**, but only if it is greater than zero.
`if (a < b) b -= a;` `else a -= b;`	Subtract the smaller of **`a`** or **`b`** from the larger.

`if (reset) {`
`  a = b = 0;`
`  reset = false;`
`}`
	If **`reset`** is **`true`**, zero out **`a`** and **`b`** and then set **`reset`** to **`false`**.

The `while` Statement

<while-statement> `::=` **`while (`***<expression>***`)`** *<statement>*

- Evaluate expression; if false do nothing
- Otherwise execute statement, then repeat
- Iteration is another hallmark of imperative languages
- (Note that this iteration would not make sense without side effects, since the value of the expression must change)
- Java also has **`do`** and **`for`** loops

Java Statement	Equivalent Command in English
`while (a<100) a+=5;`	As long as `a` is less than 100, keep adding 5 to `a`.

`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

```
<return-statement> ::= return <expression>;
                     | 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

```java
/**
 * 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

```java
/**
 * 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`**

```
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

```java
/**
 * 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:

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

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

```
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

C++	Equivalent Java

`IntList* p;`
`p = new IntList(0);`
`p->length();`
`p = q;`
	`IntList p;`
	`p = new IntList(null);`
	`p.length();`
	`p = q;`

`IntList p(0);`
`p.length();`
`p = q;`
	No equivalent.

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

```
System.out.println("Hello there");
System.out.print(1.2);
```

Printing An **IntList**

```
/**
 * 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:

```
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
```