Java

The Java language grew out of a research project at Sun about 1991 that was looking at a language to embed in consumer electronics.

Cross-platform portability was important, but security was also important.

Then when the Web came out, the designers decided that they could embed it in a Web browser.

The first Java book came out in late 1995. In the first year, about 180 books were published or announced.

Java has few innovative features. It has borrowed extensively from C, C++, Objective C, Cedar/Mesa, Modula, and Simula.

Java’s place among programming languages: The most obvious distinction between Smalltalk and Java is that Java is statically typed.

• In Smalltalk, any object may be assigned as the value of any variable.

• In Java, only objects of certain classes may be assigned to a variable. These classes are determined by the declaration of the variable.

Like Smalltalk, but unlike C++, Java provides garbage collection.

Java has other features to promote secure, robust behavior. Java “applets” may be run by Web browsers.

No Java program run via a Web browser can open, read, or write files on the user’s computer without the user’s permission.

Java programs are completely portable; their behavior is defined in terms of how they execute on the Java byte-code interpreter.
Java programs are dynamically loaded, on demand.

- This means that programs will be no larger than necessary.
- This also allows security checks or compilation to be performed at the last minute.

**The form of a Java program**

As in other OOLs, Java programs are defined in classes. A class definition consists of—

- class variables,
- class methods,
- instance variables,
- instance methods.

A Java program consists of one or more class definitions.

- Each class is compiled into its own `.class` file of Java virtual-machine object code.
- One of these files must define a method called `main()`, which is where the program begins.
  (Java applets do not contain `main()` methods, but are started via `init()` methods.)
- To invoke a Java program, one compiles the program with
  
  `javac <classname>.java`

  and then types the command
  
  `java <classname>`.

Here is the “Hello, world” program in Java:

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world.");
    }
}
```

The only parameter of `main` is an array of string objects that are the program’s command-line arguments.
Its only statement invokes the `println` method of the `System` class’s `out` object. This specifies that the information is to be shown on the display.

### Variable declarations

Here is a Java program that prints a Fibonacci sequence:

```java
class Fibonacci {
    /** Print out the Fibonacci
     * sequence for values < 50 */
    public static void main(String[] args) {
        int lo = 0, hi = 1;
        System.out.println(lo);
        while (hi < 50) {
            System.out.println(hi);
            hi = lo + hi;  // new hi
            lo = hi - lo;  /* new lo is (sum -
                             old lo), i.e., the old hi */
        }
    }
}
```

In this method, two `int`s are declared. Both are initialized to 1.

Unlike C and C++,

- Java has no “default” types; the type of each variable must be declared.
- For the sake of portability, the precision of each numeric type is defined by the language.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean</code></td>
<td>either <code>true</code> or <code>false</code></td>
</tr>
<tr>
<td><code>char</code></td>
<td>16-bit Unicode 1.1 character</td>
</tr>
<tr>
<td><code>byte</code></td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td><code>short</code></td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td><code>int</code></td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td><code>long</code></td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td><code>float</code></td>
<td>32-bit floating-point (IEEE 754-1985)</td>
</tr>
<tr>
<td><code>double</code></td>
<td>64-bit floating-point (IEEE 754-1985)</td>
</tr>
</tbody>
</table>
Variable declarations do not have to precede executable statements. They can appear anywhere before the first use of the variable.

Named constants are declared very similarly to variables. They are preceded by the words `static final`.

```java
class CircleStuff {
    static final float PI = 3.14159265;
}
```

*Arithmetic expressions*

Arithmetic expressions in Java are quite similar to most other procedural languages.

For example, multiplication and division have higher precedence than addition and subtraction.

Java supports almost all the operators that C does, except —

- the reference and dereference operators `*` and `&`, and `sizeof`,
- the sequencing operator `,`.

Most operators in Java associate from left to right, e.g.,

```
24 / 4 / 2 evaluates to ____ , not to ____.
```

The order of evaluation of an expression is left to right. Thus, expressions like

```
++x + x
```

have well defined results in Java. However, some operators associate from right to left.

Assignment, for example, produces a value, the same as the value assigned. Assignment associates from right to left. Why?
Other predicates

=, ==, and equals: In Java, = means assignment, while == means a test for equality. As in C or C++, a common error is to write = when you mean to test for equality.

In Java, == compares

- values, if the objects being compared are primitive types, or
- references, if the objects being compared are class types.

Thus, it does not test objects for equality. However, several classes define an equals method to test objects for equality.

For example, suppose we define

```java
String s1 = new String("Hello, world!");
String s2 = new String("Hello, world!");
```

Then s1 == s2 returns

while s1.equals(s2) returns

If we then define

```java
String s3 = s1;
```

then s1 == s3 returns

and s1.equals(s3) returns

One caveat is that if we define

```java
String s4 = "Hello, world!";
String s5 = "Hello, world!";
```

then s4 == s5 and s4.equals(s5) are both true.

This is because Java treats all anonymous Strings with the same contents as one anonymous String that has many references.

&& and | | vs. & and | |: The double operators && and | | may not evaluate both of their operands.
The left-hand side operand is evaluated. If the truth or falsity of the expression can be determined from the lh. operand, then the rh. operand is not evaluated.

However, & and | always evaluate both of their operands.

Ordinarily, it is more efficient to use && and || on Boolean expressions. Use the others only when you are depending on side-effects of evaluating the right-hand expression.

The operators | and & can also be used on integral expressions; in this case, they perform bitwise and and or.

**Augmented assignment:** The form \langle operator \rangle = \ldots\text{ means that an operation is performed on the lhs. and rhs. and then assigned to the lhs.}\ That is,

\[
\text{result } *= 2
\]

means the same as

\[
\text{result } = \text{result } * 2
\]

This is a useful shorthand, but sometimes obscures the fact that an assignment is being performed.

The augmented assignment may be quicker when using arrays, because the index expression needs to be calculated only once. So, \text{a[i*5] } *= 2\ may be faster than \text{a[i*5] } = \text{a[i*5]} \ast 2.\n
**Switch statement:** Java has a switch statement like C or C++:

```java
public class Demo {
    public static int fibonacci(int n) {
        switch (n) {
            case 0: return 0;
            case 1: return 1;
            default: return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }
}
```
Note that if the first two cases did not return, we’d have to insert break statements to prevent control from falling through to the next case.

### Methods

Java methods look a lot like C++ member functions. They are also similar to C functions.

Here is a simple method to convert Celsius temperatures to Farenheit:

```java
class Demonstrate {
    public static float farenheit (float c) {
        return c * 9/5 + 32;
    }
}
```

It can be called by the `main` method, below.

```java
public static void main (String args[]) {
    System.out.println(farenheit(30));
    System.out.println(farenheit(-40));
}
```

The parameter types and return type of a method must be specified.

Notice that the `farenheit` method returns a value of class `float`. It has one parameter, `c`.

If the method had had two parameters, e.g., `method2(float a, float b)`, it would have been necessary to specify a data type for each.

Java passes arguments *by value*. This means that a copy of the argument is made when the method is called. (For objects of class types, a copy of the *reference* to the object is made.)

The `static` keyword indicates that the `farenheit` method is a class method.

The `public` keyword indicates that the method can be called by any method defined in any class.

A return statement is not required in methods with a return type void, which do not return values:
public static void printCelsius (float f) {
    System.out.println((f - 32) * 5/9);
}

A method can be called from a class that does not define it by prefixing the name of the class to the method name.

class TempConvert {
    public static void printCelsius (float f) {
        System.out.println((f - 32) * 5/9);
    }
}

...  
class Demonstrate {
    public static void main (String args[]) {
        TempConvert.printCelsius(59);
    }
}

If multiple classes define methods of the same name, the method name is said to be overloaded.

In this case, the method that will be invoked depends on the class of the target instance.

However, unlike C++, Java does not permit the programmer to overload operators.

Except for the obvious overloading of arithmetic operators for different numeric types, there is only one operator that is overloaded in Java: +.

    • For numbers, + means addition.
    • For strings, + means concatenation.

Why do you suppose that Java doesn’t allow overloading operators?

In addition to overloading method names in different classes, it is also possible to overload method names in the same class, provided that each has a unique pattern of parameter data types.

For example, you could have different fahrenheit methods for integer and floating-point temperatures in the Demonstrate class.
**Variable scope and extent**

Suppose a method of class `Dictionary` takes two parameters, `k` and `v` (for key and value).

```java
public insert(key k, value v) { ... }
```

When `insert` is executed, values for any existing variables named `k` or `v` are protected from modification by the `insert` method.

The accessibility of a variable depends on what *block* it was declared in.

A block is a group of statements surrounded by braces.

- The body of a method is a block.
- The object of an `if-else` statement is often a block.

Variables that are declared inside blocks are said to be *local variables*.

A parameter is treated as though it were a local variable declared inside a method body.

The scope of a local variable is from the point of declaration to the end of the block.

Local variables are thus said to have *local scope*.

The storage for parameters and local variables is returned as soon as the corresponding method is finished executing.

Thus, parameters and local variables are said to have *dynamic extent*.

**Class and instance variables**

Besides local variables, Java programs have class variables and instance variables.

Class variables are essentially the same as class variables in Smalltalk (and static members in C++). They are accessible at any time from any method of the class.
Class variables are defined exactly like local variables, except—

- They are defined inside the body of the class definition, but not inside the body of any method declaration.
- They are prefixed with the keyword `static`.

For example, in our `TempConvert` class, we are likely to use the values 32 and $5/9$ in several places.

```java
class TempConvert {
    public static float FCSlope = (float) 5/9;
    public static int FCOffset = 32;
    ...
}
```

In some classes, we might want to change the value of class variables during execution of a program. In this class, the values should remain constant; hence they should be declared as

Note that such variables must be initialized.

Declaration of a parameter or local variable of the same name as a class variable will hide the class variable. The local variable is said to `shadow` the class variable.

**Declaration vs. definition**

A *declaration* is a program element that provides a compiler with essential information or useful advice.

When you introduce a local variable or parameter, you *declare* it, because you tell the compiler its type.

However, memory is not allocated for the variable until the method is run.

A *definition* causes a compiler to set aside memory at compile time. When you introduce a class variable, you both declare and define it.

Defining a variable always implies declaring it, but not vice versa.
Creating and using objects

Just as in Smalltalk or other OOLs, a class definition is a “cookie cutter” out of which instances can be created.

For example, we might declare a `Point` class like this:

```java
class Point {
    public double x, y;
}
```

The variables $x$ and $y$ are instance variables.

We can create new points by code like this:

```java
Point lowerLeft = new Point();
Point upperRight = new Point();
```

We can assign to the instance variables:

```java
lowerLeft.x = 0.0;
lowerLeft.y = 0.0;

upperRight.x = 640.0;
upperRight.y = 1024.0;
```

When new points are created, they are created on the heap. This means that they do not go away when the method that created them returns.

Instead, class instances are garbage-collected only when the last reference to them is removed.

Just as our classes can have instance variables, they can also have instance methods:

```java
class Point {
    public double x, y;

    public double distance (Point that) {
        double xdiff, ydiff;
        xdiff = x - that.x;
        ydiff = y - that.y;
        return Math.sqrt(xdiff*xdiff + ydiff*ydiff);
    }
}
```
An instance method is invoked by naming the receiver, followed by the name of the method:

```java
double d = lowerLeft.distance(upperRight);
```

**Primitive and reference types**

In Java, types are divided into two categories:

- *primitive* types—boolean, character, and the numeric types.
- *reference* types—class instances, which you can “take apart” into their components.

A primitive-type variable contains the value of the variable, while a reference-type variable contains the address of an object.

When an argument of a reference type is passed, it is the address of the object that is copied.

Therefore, although the called routine cannot affect the value of the variable in the calling method, it can change the instance variables in the instance!

**Constructors**

Like other variables, instance variables can be initialized.

In the point class, we might want to arrange for the \(x\) and \(y\) coordinates to have initial values of 0:

```java
class Point {
    public double x = 0, y = 0;
}
```
In this case, it turns out that it is not necessary to specify initial values, since if we do not specify an initial value, a variable is initialized to its standard default. Arithmetic variables have a standard default of 0.

Java also allows the programmer to write a constructor method, which is automatically called whenever a new class instance is created.

Constructors—

- have the same name as the class, and
- never return values (so a return type is not specified).

A constructor may have zero or more parameters.

Suppose that we wanted to include a constructor in our `Point` class.

```java
class Point {
    public double x, y;
    public Point(double x0, double y0) {
        x = x0; y = y0;
    }
}
```

Now we can shorten the code above that defined our two points:

```java
Point lowerLeft = new Point(0.0, 0.0);
Point upperRight = new Point(640.0, 1024.0);
```

Of course, since we want both instance variables in `lowerLeft` to be initialized to the standard defaults, the effect is the same as if we had used a zero-parameter constructor:

```java
Point lowerLeft = new Point(); // Error!
```

However, in this case, Java does not let us use the zero-parameter constructor. This is because:

- we have defined a constructor with parameters, but
- we have not defined a zero-parameter constructor.

This is because the lack of arguments suggests that we have forgotten to include arguments for the two-parameter constructor.
We can make the above code work by defining our own zero-parameter constructor:

```java
public Point() {
    x = y = 0;
}
```

or simply

```java
public Point() {}
```

**Accessor methods**

Just like in Smalltalk, getter and setter methods are often defined for instance variables in Java.

Getters and setters for the `Point` class would look like this:

```java
public
}

public
}
```

Setters are especially valuable when you need to coordinate instance-variable changes with display changes.

They also promote data abstraction, because the underlying representation of the `Point` class can be changed, for example, to polar coordinates, by simply changing the `getX`, `setX()`, `getY`, and `setY()` methods.

So far, we have seen only public variables and methods. However, if we really want `Point` to be a data abstraction, we would not want to allow other classes to access its instance variables.

We can arrange this by declaring its instance variables to be private:

```java
class Point {
    private double x, y;
    ...
}
```
Inheritance

The keyword `extends` is used to signify inheritance in Java.

For example, suppose we want to subclass `Point` to create a class `Circle`. Then we might write—

```java
public class Circle extends Point {
    protected double radius;
    public Circle() {
        super(0, 0); // Call the
        // superclass constructor
        setRadius(0);
    }
}
```

So far, we have seen three new features of Java:

- The keyword `extends` is used to signify subclassing.
- The `radius` instance variable is declared not as `public` or `private`, but as `protected`.
  This means that it is accessible to the `Circle` class and any of its subclasses, or other classes in its `package`, but not to other classes in the system.

How does this compare with instance variables in Smalltalk?

In declaring variables and methods to be `public`, `protected`, or `private`, it is good to follow the need-to-know principle: Only those classes that need to be able to access a variable or method should be able to.

This prevents other classes from coming to depend upon them, and allows the programmer to change their implementation later, if needed.

- In the process of creating a `Point` object, the superclass constructor is called. This is denoted by the call `super(...)`.

Continuing with the class definition …
// Three-parameter constructor
public Circle(double r, double x, double y) {
    super(x, y);
    setRadius(r);
}

public void setRadius(double r) {
    radius = r;
}

public double getRadius() {
    return radius;
}

// Calculate area of circle
public double area() {
    return Constants.pi * radius * radius;
}

// Return a representation of circle
public String printString() {
    return "Center = " + "(" + x + "," + y + "); Radius = " + radius;
}

Circle is a subclass of Point. Any subclass of Circle is also a subclass of Point.

Because Circle inherits directly from Point, it is called a direct subclass of Point. Point is called a direct superclass of Circle.

In the last lecture, our definition of Point did not specify that it was extending a class. Since it did not, Object was its direct superclass.

A class can extend only one class; that is, it can have only one direct superclass. Because of this, Java is said to have single inheritance.

The “opposite” of single inheritance is multiple inheritance, as in C++.

Since Java has single inheritance, all classes form a tree with Object at the root.
For safety’s sake, Java always makes sure that a constructor calls its superclass’s constructor. (Why did I say, “For safety’s sake”?)

Thus, if the first statement in a constructor is not a call to super, then Java implicitly inserts the call super().

As in Smalltalk, a subclass can override some of its superclasses’ methods.

To decide which method to use, Java follows the same rule as Smalltalk: If an invoked method is not defined in a class, then the direct superclass is checked, then its direct superclass, etc.

That is, the compiler searches from the object’s class up the tree toward Object until it finds a definition of the method.

Unlike in Smalltalk, the compiler can determine whether the call will succeed. Why?

**Abstract classes**

In Smalltalk, our abstract classes were identified by having some of their methods implemented as subclassResponsibility.

In Java, it is possible to declare a class to be abstract, and then any attempt to create an instance of it will be flagged by the compiler.

Consider an abstract class Shape, with subclasses of Circle and Rectangle:

```java
public abstract class Shape {
    public abstract double area();
    public abstract double circumf();
}
```

Notice that not only the class, but also its methods are declared to be abstract.

A method declared as abstract is required to be implemented in every subclass of the class declaring the abstract method.
Note that an abstract method has no body; a semicolon immediately follows the signature.

If a class extends Shape but does not implement both area and circumf, the compiler will complain.

Here is the code for the Circle class:

```java
public class Circle extends Shape {
    protected double radius;
    public Circle() {radius = 1.0;}
    public Circle(double r) {radius = r;}
    public double area() {
        return Constants.• * radius * radius;
    }
    public double circumf() {
        return 2 * Constants.• * radius;
    }
    public double getRadius()
        { return radius;}
}
```

What would happen here if we didn’t have the getRadius() access method?

Here is the code for Rectangle:

```java
public class Rectangle extends Shape {
    protected double w, h;
    public Rectangle() {}
    public Rectangle(double wd, double ht) {
        w = wd; h = ht;}
    public double area() {return w*h;}
    public double circumf() {return 2*(w+h);}
    public double getWidth() {return w;}
    public double getHeight() {return h;}
}
```

If a class is not supposed to be subclassed, you can declare it as _____ . Then any attempt to subclass it produces a compilation error.
We have seen how a constructor can call a constructor of a superclass. It is also possible for a constructor to call a constructor of the same class.

For example, suppose our `Rectangle` class above included a one-parameter constructor for specifying a zero-height rectangle (a horizontal line). It could be implemented simply by calling the two-parameter constructor:

```java
public Rectangle(double wd) {
    this(wd, 0.0);
}
```

In this call, we need to use the `this` keyword. What would happen if we tried to use `Rectangle(wd, 0.0)`?

In general, `this` in Java is like `self` in Smalltalk; Smalltalk messages that would be sent to `self` are sent to `this` in Java.

For example, suppose we redefined the `area` method of `Circle` in terms of `circumf`:

```java
public double area() {
    return this.circumf() * radius / 2;
}
```

Since the receiver of the message is the same instance as the sender, we send `this.circumf()`.

Instead of explicitly naming `this`, however, Java allows us to omit it. That is, if the receiver is omitted, then `this` is understood:

```java
public double area() {
    return circumf() * radius / 2;
}
```

**Summary**
• Java is a statically typed language designed for security. Classes are dynamically loaded. Programs are portable and can be run from inside network browsers.

• Every Java program must contain a class definition that defines a method named `main`.

• For the sake of portability, all Java implementations have to use variables of the same precision.

• The rules of precedence in arithmetic operations are similar to those in most other procedural languages.

• Control flow is similar to C and C++, except for the fact that expressions are guaranteed to be evaluated left-to-right.

• Like Smalltalk, Java has class and instance variables, and class and instance methods. However, in Java, the parameters and return values are typed.

• Java’s scope rules are similar to C or C++. Its overloading of methods is similar to Smalltalk’s, except that multiple methods with the same name but different parameter types are allowed.

• Java’s parameter passing is similar to Smalltalk’s.

• Constructor methods provide a straightforward way of initializing new class instances in Java.

• Java is a single-inheritance language.

• Java provides linguistic mechanisms for declaring abstract superclasses, and enforcing the requirement that abstract methods be implemented in subclasses.