Brief and Incomplete History

- Work started by Sun in 1991
- Initial goal: programming for consumer electronics devices, smart appliances, etc.
  - e.g. TV set-top boxes, video-on-demand
  - too early for this: no market
- In 1994 focus switched to the Internet
  - Java applets as the “next wave” after older technologies such as FTP and HTML
  - First big break: agreement with Netscape to support Java in the Netscape browser (1995)

Brief and Incomplete History (cont)

- Currently: Java as a general-purpose programming language
  - “Forget the annoying applets, think of it as a better C++”
  - Many advantages: platform-independent, “clean”, many existing libraries (speeds up development), widely used
- IBM is pushing Java-based technology: J2EE (Java2 Enterprise Edition)
- Google is pushing Java-based technology: Android

The “Big Three” Books

- **The Java Programming Language, 4th Edition,** Ken Arnold, James Gosling, David Holmes
- **[JLS] The Java Language Specification, Java SE 8 Edition,** James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, Alex Buckley
- **[JVMS] The Java Virtual Machine Specification, Java SE 7 Edition,** Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley

Quick Overview

- Classes, methods, fields
- Objects and reference variables
- Constructors
- Compilation model and execution model
- Inheritance
- Method invocation
- Abstract classes and interfaces

Classes

- A **class** is a blueprint for creating objects

```
class Rectangle {
        public double height, width;
        public double area() {
                return height * width;
    }
}
```

- **Class members**: methods and fields

Objects

- The central concept of “object-oriented” programming
- In Java, they are **instances** of classes, created through ***new***
  - e.g. when expression ***new Rectangle()*** is evaluated, a new object is created
  - “instance” = “object”
  - “class X is instantiated” = “an instance of X is created”

References to Objects

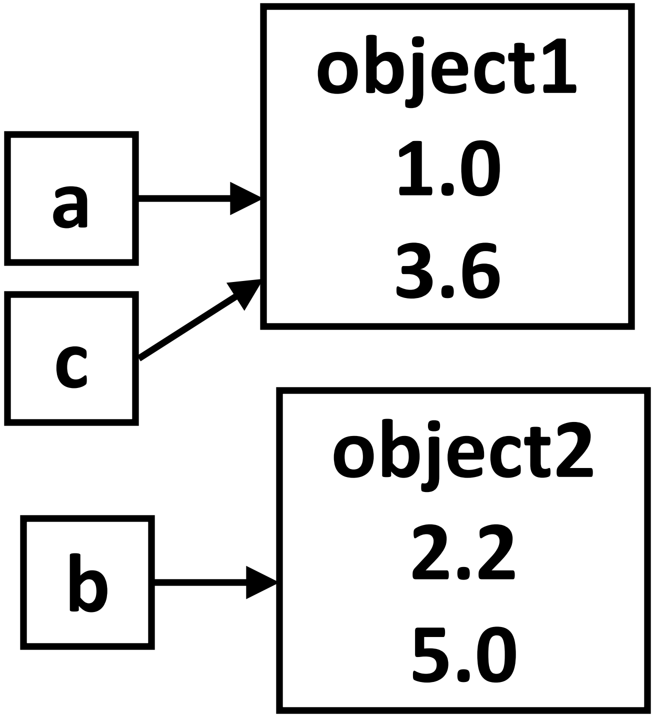
- Objects are manipulated indirectly through object references ("clean pointers")

- ***Rectangle x = new Rectangle()***
  - ***x*** is a variable of type "reference to Rectangle objects"
  - During the evaluation of ***new Rectangle():***
    - A new instance of Rectangle is created
    - A reference to this instance is "returned"
  - ***x*** is assigned this reference value
    - e.g. the value is may be the <u>address</u> of the first byte of the object's memory layout

Class Members: Methods and Fields

- Two separate kinds: *instance members* and *static members*
  - Instance members: each instance of the class has a separate copy

```
Rectangle a, b, c;
a = new Rectangle();
b = new Rectangle();
a.height = 1.0; a.width = 3.6;
b.height = 2.2; b.width = 5.0;
c = a;
```



Instance Methods

- An instance method operates on objects
  - “method m is invoked on the object”

double area() { return height*width;}

in reality, this is syntactic sugar over

```
double area(Rectangle this) {
        return this.height * this.width; }
```

- There is an implicit formal parameter this which is *a reference to the object on which the method was invoked*

Static Members

- Static field: there is only one copy for the entire class
  - Very similar to a global variable
  - ***public static final int VERSION = 2;***
- Static method: does not operate on objects
  - Very similar to a traditional procedure
  - Of course, ***this*** is not allowed inside
- Static members should be used carefully: with too many, the code starts looking procedural rather than object-oriented

Method Overloading

- A class may have more than one method with the same name
  - the name is "overloaded"
- All overloaded methods must have different signatures
  - Signature: method name + types of formal parameters
- double area() { … } → **area()**
- double area(int num_digits) { … } → **area(int)**

Constructors

- Constructors are used to set up the initial state of new objects

```
public Rectangle(double height,double width) {
        this.height = height; this.width=width; }
```

- ***x = new Rectangle(1.0,2.0)***
  - A new object is created
    - With default field values as defined in JLS
  - The constructor is invoked on this object
  - A reference to the object is assigned to x

Constructors (cont)

- A constructor may call another constructor in the same class

```
class Rectangle {
        Rectangle(double h, double w) { … }
        Rectangle(double h) { this(h,1.0); }
}
```

- Only allowed as the first statement in the constructor body

Constructors (cont)

- If a class contains no constructors, a default constructor is provided automatically
  - It only calls the superclass constructor
- According to the JLS, constructors are not considered class members
  - e.g., this means that they cannot be inherited
- More about constructors and inheritance in a few slides

Compilation Model

- The compiler takes as input source code
  - Typically, class A is stored in file A.java
    - Exception: nested classes
- Compiler output: Java bytecode
  - A.java -> A.class
  - A standardized platform-independent representation of Java code
  - Essentially, a programming language that is understood by the Java Virtual Machine

Rectangle.class (decompiled with javap)

```
class Rectangle extends java.lang.Object {
    public double h;  public double w;
    Rectangle();
    public double area();
}
Rectangle()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 return
double area()
   0 aload_0
   1 getfield #4 <Field double h>
   4 aload_0
   5 getfield #5 <Field double w>
   8 dmul
   9 dreturn
```

Execution Model

- Java bytecode is executed by a Java Virtual Machine (JVM)
  - Several vendors for JVMs
    - e.g. IBM sells a JVM that is performance-tuned for enterprise server applications
- Platform independence: as long as there are JVMs available, the exact same Java bytecode can be executed anywhere

JVM

- There are two ways to execute the bytecode
- Interpretation: the VM just executes each bytecode instruction itself
  - Initial JVMs used this model
- Compilation: the VM uses its own internal compiler to translate bytecode to native code for the platform
  - The native code is executed by the platform
  - Faster than interpretation

Compilation inside a VM

- Just-in-time: the first time some bytecode needs to be executed, it is compiled to native code on the fly
  - Typically done at method level: the first time a method is invoked, the compiler kicks in
  - Problems: compilation has overhead, and the overall running time may actually increase
- Profile-driven compilation
  - Start executing through interpretation, but track “hot spots” (e.g. frequently executed methods), and at some point compile them

Memory Allocation in a VM

- Three general “chunks of memory”
- Heap area for objects
  - e.g. ***new Rectangle()***
- Stack area: a place to store local variables, partial results, arguments and returns for method invocations
  - void m() { Rectangle x; x = new Rectangle(); }
  - Separate stack frame for each method call
- Method area: all code + all static fields

Inheritance

- class B extends A { … }
  - Single inheritance: only one superclass
- Every member of A is inherited by B
  - if a field **f** is defined in A, every object of class B has an **f** field
  - if a method **m** is defined in A, this method can be invoked on an object of class B
- B may declare new members
- Reference variables for A objects also may refer to B objects
  - A a = new B();

Example

```
class Rectangle {
   double h,w;
   Rectangle(double h,double w) { … }
   double area() { … }
}
class RectangleWithHoles {
   int hole_size;
   RectangleWithHoles(double h,double w,int hs)
         { super(h,w); hole_size = hs; }
   void reduceHoles() { hole_size--; }
   double area()
                    { return super.area() – hole_size; }
}
```

Constructors and Inheritance

- Constructors are not inherited
- A constructor in a subclass typically invokes a constructor in the superclass
  - **super(a,b,c);**
- If a superclass constructor is not invoked, and if there is no call to another subclass constructor **this(a,b);** the compiler automatically inserts a call **super();**

Inheritance of Fields

- If a subclass declares a field with the same name, the field in the superclass is hidden (but it is still there)
- Access is still possible through “super”

```
class A { int f; … }
class B extends A {
  int f;
  void m() {
        this.f = 5;
        super.f = 6;
}
```

Inheritance of Methods

- If a subclass declares a method with the same name but a different signature, we have overloading
  - Either method can be invoked on an instance of the subclass
- If a subclass declares a method with the same signature, we have overriding
  - Typically, only the new method applies to instances of the subclass
    - If we want to use the overridden superclass method, we need "super"

“final”

- If a class is declared final, no subclasses are allowed
- If a method is declared final, it cannot be overridden in subclasses
- If a field is declared final, its value cannot be changed after initialization
- In all three cases, the compiler enforces these restrictions

Method Invocation – Compile Time

- What happens when we have a method invocation of the form **x.m(a,b)?**
- Two very different things are done
  - At compile time, by the Java compiler (javac)
  - At run time, by the Java Virtual Machine
- At compile time, a target method is associated with the invocation expression
  - "compile-time target", "static target"
  - The static target is based on the declared type of x

Method Invocation – Compile Time

class A { void m(int p, int q) {...} ... }

class B extends A { void m(int r, int s) {...} ... }

A x;

x = new B();

x.m(1,2);

- Since x has declared type A, the compile-time target is method m in class A
- javac encodes this in the bytecode (foo.class)
  - **virtualinvoke x,<A: void m(int,int)>**

Method Invocation – Run Time

- The Java virtual machine loads the bytecode and starts executing it
- When it tries to execute instruction **virtualinvoke x,<A: void m(int,int)>**
  - Looks at the class Z of the object that x refers to at that particular moment
  - Searches Z for a method with signature **m(int,int)** and return type **void**
  - If Z doesn't have it, goes to Z's superclass, and so on upwards, until a match is found

Method Invocation – Run Time

- The run-time (dynamic) target: “lowest” method that matches the signature and the return type of the static target
  - “Lowest” with respect to the inheritance chain from Z to java.lang.Object
- Once the JVM determines the run-time target method, it invokes it on the object that is pointed-to by x
- “virtual dispatch”, “method lookup”

Terminology

- Invocation x.m(a,b) sends a message m to the object referred to by x
  - This object is the receiver object, and its class is the receiver class
  - The method that contains x.m(a,b) belongs to the sender object
- Dynamic binding (virtual dispatch): mapping the message to a method
- Polymorphic call: more than one possible run-time target

Calls through "super"

- Basic rule: at run time, start from the class of the receiver object and look upwards for a match
- The only exception: calls through "super"
  - At compile time, the static target of the call is defined as the appropriate method in the superclass
  - At run time, the method is invoked directly – there is no virtual dispatch
  - Different bytecode instruction: **invokespecial** instead of **invokevirtual**

Calls through "super" - example

class A { void n() {…} … }

class B extends A { void m() {…} … }

class C extends B {

void n() {…}

void example() {

this.n(); → static C.n; dynamic C.n

this.m(); → static B.m, dynamic B.m

super.n(); → static A.n, dynamic A.n

super.m(); → static B.m, dynamic B.m

} … }

C c = new C(); c.example()

Abstract Classes

- Certain methods do not have bodies
  - abstract void m(int x);
- Of course, we cannot create instances of abstract classes
- An abstract method can be the compile-time target of a method call
  - But not the run-time target, obviously
- Sometimes non-abstract classes are referred to as “concrete classes”

Interfaces

- Very similar to abstract classes in which <u>all</u> methods are abstract
- A class can have only one superclass, but can implement many interfaces
  - **class Y extends X implements A, B { … }**
- A reference variable can be of interface type, and can refer to any instance of a class that implements the interface
- An interface method can be the compile-time target of a method call

Example

```
interface X { void m(); }
interface Y { void n(); }
abstract class A implements X {
        void m() { … }
        abstract void m2();
}
class B extends A implements Y {
        void m2() { … }
        void n() {…}
}
X x = new B(); x.m();
Y y = new B(); y.n();
A a = new B(); a.m2();
```

compile-time targets