An Introduction to Java

A pre-tutorial

Prof. Oscar Nierstrasz

Software Composition Group
Institut für Informatik (IAM)
Universität Bern

oscar@iam.unibe.ch
http://iamwww.unibe.ch/~scg

Contents

1. Object-Oriented Programming and Java

Overview

- ❑ Dimensions of Object-Oriented Languages
- ❑ Objects and Dynamic Binding
- ❑ Inheritance and Subtyping
- ❑ Generics and Type Casting

Text:

- ❑ David Flanagan, *Java in a Nutshell*, O’Reilly, 1996

On-line resources:

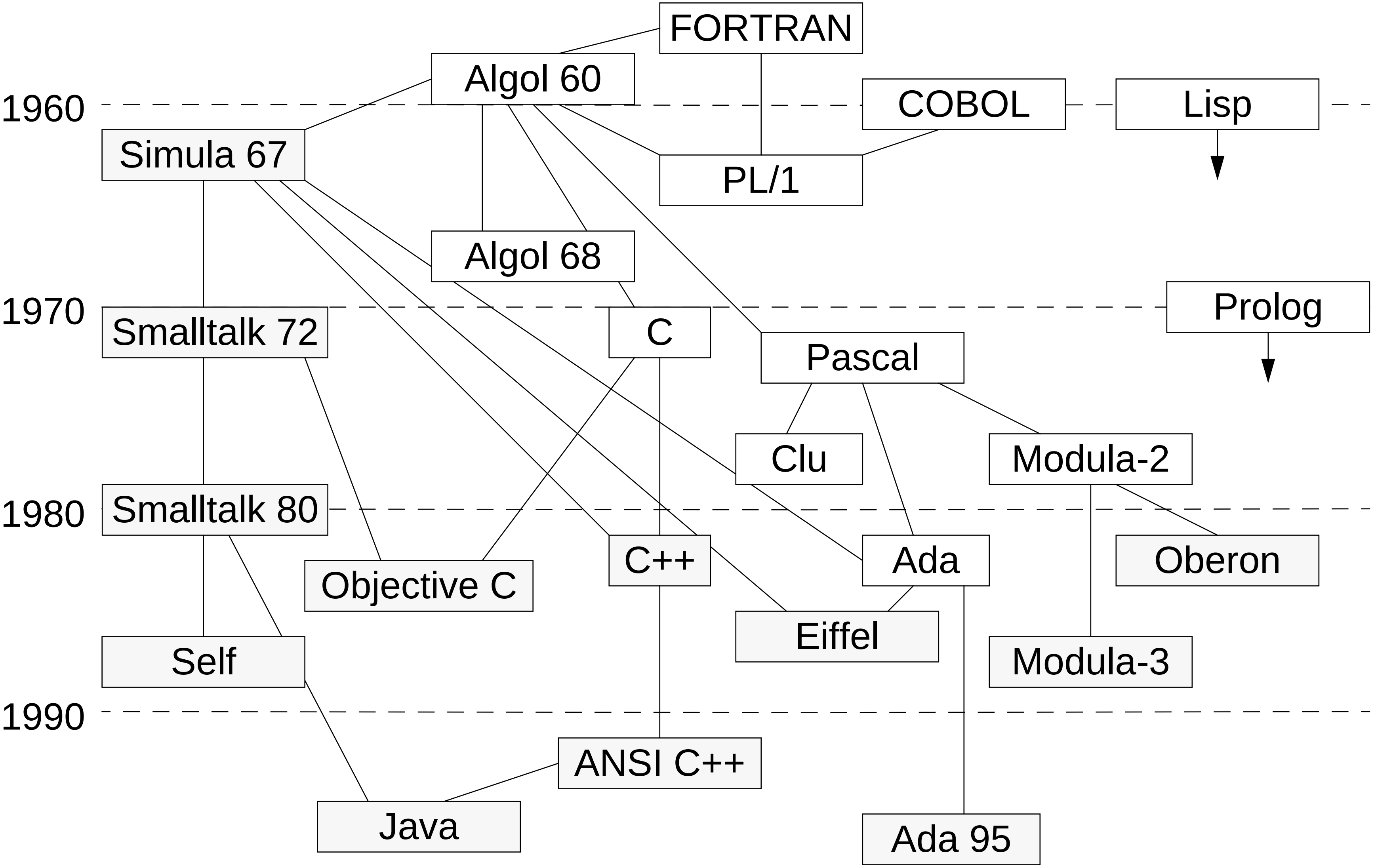
- ❑ Locally installed Java resources (on-line tutorial, language spec, etc):

  `http://iamwww.unibe.ch/~scg/Java/`

- ❑ Free Java implementations and documentation (Swiss mirror):

  `ftp://sunsite.cnlab-switch.ch/mirror/javasoft/`

The Evolution of OOP



Dimensions of Object-Oriented Languages

- ❑ **Object-Based** languages (e.g., Ada) support *encapsulation* of behaviour and state (objects)
- ❑ **Class-Based** languages (e.g., Clu) support *instantiation* of objects from object classes
- ❑ **Object-Oriented** languages (e.g., Objective C) support *inheritance* between classes
- ❑ **Pure Object-Oriented** languages (e.g., Smalltalk) model *all* data types as objects (vs. Hybrid OOLs like C++)
- ❑ **Strongly-Typed** object-oriented languages (e.g., Eiffel) guarantee that all expressions are type-consistent
- ❑ **Concurrent** object-oriented languages (e.g., Java) allow multiple objects to serve requests concurrently; individual objects can schedule and synchronize concurrent requests
- ❑ **Persistent** object-oriented languages support objects whose lifetime may span multiple user sessions

— Wegner, OOPS Messenger, Vol. 1, #1, 1990

Java

Language design influenced by existing OO languages (C++, Smalltalk ...):

- ❑ Strongly-typed, concurrent, pure object-oriented language
- ❑ Syntax, type model influenced by C++
- ❑ Single-inheritance but multiple subtyping
- ❑ Garbage collection

Innovation in support for network applications:

- ❑ Standard API for language features, basic GUI, IO, concurrency, network
- ❑ Compiled to bytecode; interpreted by portable abstract machine
- ❑ Support for native methods
- ❑ Classes can be dynamically loaded over network
- ❑ Security model protects clients from malicious objects

Java applications do not have to be installed and maintained by users

Java and C++ — Similarities and Extensions

Java resembles C++ only superficially:

Similarities:

- primitive data types (in Java, platform independent)
- syntax: control structures, exceptions ...
- classes, visibility declarations (`public`, `private`)
- multiple constructors, `this`, `new`
- types, type casting

Extensions:

- garbage collection
- standard classes (Strings, collections ...)
- packages
- standard abstract machine
- final classes

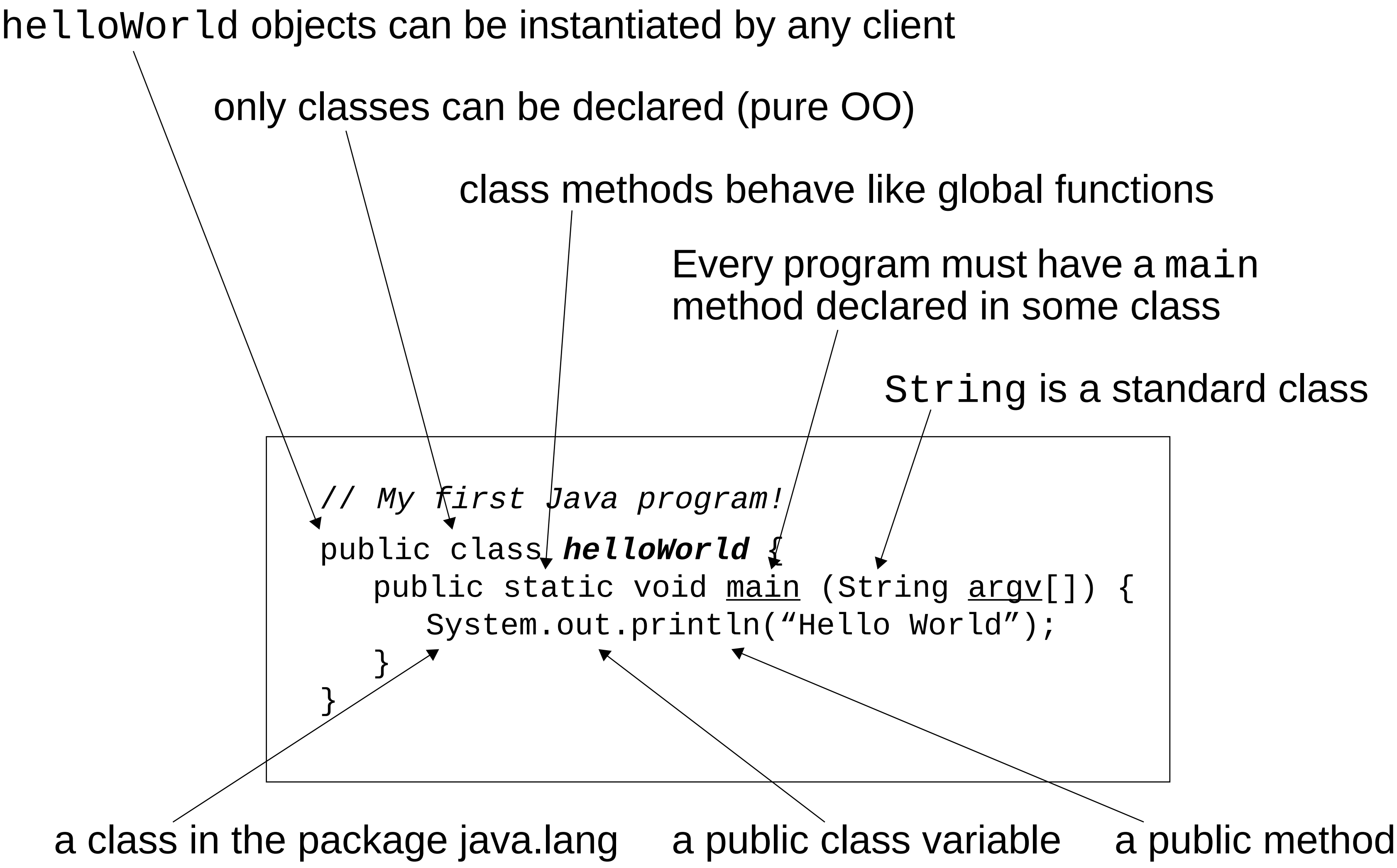
Java and C++ — Simplifications

Whereas C++ is a hybrid language, Java is a pure object-oriented language that eliminates many of the complex features of C++:

Simplifications:

- no pointers — just references
- no functions — can declare `static` methods
- no global variables — can declare `public static` variables
- no destructors — garbage collection and `finalize` methods
- no linking — dynamic class loading
- no header files — can define `interface`
- no operator overloading — only method overloading
- no member initialization lists — `super` constructor can be called
- no preprocessor — `static final` constants and automatic inlining
- no multiple inheritance — can implement multiple interfaces
- no structs, unions, enums — typically not needed
- no templates — but generics will likely be added ...

The “Hello World” Program



helloWorld objects can be instantiated by any client

only classes can be declared (pure OO)

class methods behave like global functions

Every program must have a main method declared in some class

String is a standard class

```
// My first Java program!
public class helloWorld {
   public static void main (String argv[]) {
      System.out.println(“Hello World”);
   }
}
```

a class in the package java.lang

a public class variable

a public method

Packages

A Java program is a collection of classes organized into *packages*

- At least one class must have a `public static void main()` method
- The first statement of a source file may declare the package name:

  ```
  package games.tetris;
  ```

- Source files (e.g., helloWorld.java) are compiled to bytecode files (e.g., helloWorld.class), one for each target class
- Class files must be stored in subdirectories corresponding to the package hierarchy
- When using classes, either the full package name must be given:

  ```
  java.lang.System.out.println(“Hello World”);
  ```

  or classes from the package may be *imported:*

  ```
  import java.lang.*; // this package is always imported by default
  ```

- Class names are usually capitalized for readability:

  ```
  a.b.c.d.e.f(); // which is the name of the class?!
  ```

Java Basics

Java’s primitive data types and control statements resemble those of C/C++:

Primitive Data Types:

```
boolean byte char double float int long short void
```

Literals:

```
false null true
```

Control flow:

```
if ( boolean ) { Statements } else { Statements }
for ( boolean ) { Statements }
while ( boolean ) { Statements }
do { Statements } while ( boolean )
switch ( variable ) {
   case label : Statements;
      break; ...
   default : ... break;
}
```

Classes and Objects

The encapsulation boundary is a class (not an object):

```
public class Point {
   private double x, y;   // not accessible to other classes (even subclasses)
   // constructors:
   public Point (double xCoord, double yCoord) { x = xCoord; y = yCoord; }
   public Point (Point p) { x = p.x; y = p.y; } // can access private data here
   // public methods:
   public double getX ( )             { return x; }
   public void   setX (double xCoord){ x = xCoord; }
   public double getY ( )             { return y; }
   public void   setY (double yCoord){ y = yCoord; }
   public double distance ( )         { return Math.sqrt(x*x + y*y); }
}
```

In pure OOLs, (non-primitive) objects are passed by reference, not by value:

```
int a = 3, b = 4;         // a and b are primitive objects
Point p1 = new Point(a,b);// p1 is a reference to an object (NB: a & b coerced!)
int c = a;                // c gets value of a
c = 8;                    // c gets new value; a is unchanged
Point p2 = p1;            // p2 refers to p1
Point p3 = new Point(p1); // p3 is a copy of p1
p2.setX(c);               // The object p1 and p2 refer to is modified
```

Garbage Collection

In Java (as in Smalltalk and Eiffel), objects no longer referred to are automatically garbage-collected:

- ❑ no need to explicitly `delete` objects
- ❑ no destructors need to be defined
- ❑ no need to write reference-counting code
- ❑ no danger of accidentally deleting objects that are still in use

You can still exercise extra control:

- ❑ Cleanup activities can be specified in a `finalize` method
  - ☞ useful for freeing external resources (files, sockets etc.)
- ❑ Objects you no longer need can be explicitly “forgotten”
  - ☞ you can explicitly forget objects by assigning the value `null` to a variable (this is the initial value of declared, but unassigned variables)

Inheritance

A subclass *extends* a superclass, inheriting all its features, and possibly overriding some or adding its own:

```
public class Circle extends Point {
   private double r;
   public Circle (double xCoord, double yCoord, double radius) {
      super(xCoord, yCoord);    // call Point constructor
      r = radius;
   }
   public Circle (Circle c) {
      super(c);                // call Point constructor with c as Point
      r = c.r;
   }
   public double getR ( )            { return r; }
   public void   setR (double radius){ r = radius; }
   public double distance ( )        { return super.distance() - r; }
}
```

Public superclass features can always be accessed, even if overridden.

Dynamic Binding

One of the key features of object-oriented programming is *dynamic binding* — the actual method that will be executed in response to a request depends on the dynamic type of target, not the static type of the reference:

```
Point p = new Circle(5, 12, 4);
System.out.println("p.distance() = " + p.distance());
```

yields:

```
p.distance() = 9
```

In pure OOLs, all methods are dynamically bound by default.

Static binding is the exception:

- ❑ `static` methods belong to classes, so are statically bound
- ❑ `private` methods have purely local scope
- ❑ `final` methods cannot be overridden, so are statically bound

Downcasting

Dynamic binding can cause type information to be lost:

```
Point p = new Circle(5, 12, 4);   // p refers to a Circle – upcast ok
Circle c1 = p;                    // compile-time error! – can’t downcast
```

Type information can be recovered at run-time by explicit tests and casts:

```
if (p instanceof Circle) {        // run-time test
   c1 = (Circle) p;               // explicit run-time downcast ok
}
```

An attempt to cast to an invalid type will raise an exception at run-time:

```
p = new Point(3,4);
c1 = (Circle) p;                  // invalid downcast raises run-time exception
```

Feature Visibility

Features (can be declared with different degrees of visibility:

- `private` — accessible only within the class body
- `public` — accessible everywhere
- `protected` — accessible to subclasses *and* to members of the same package
  - ☞ allows access to cooperating classes
- default (no modifier) — accessible throughout the package only
  - ☞ allows package access but prevents all external access

Modifiers

In addition to feature visibility, modifiers can specify several other important attributes of classes, methods and variables:

- ❑ `abstract` — unimplemented method; class must also be declared abstract
  - ☞ method signature is followed by semi-colon instead of body
- ❑ `final` — class/method/variable cannot be overridden by subclass
- ❑ `static` — method/variable belongs to class, not instances; implicitly final
- ❑ `native` — method implemented in some other language, usually C

Exceptions

A class must declare which exceptions it `throws`, or it must `catch` them:

```
public class TryException {
   public static void main(String args[]) {
      try {
         alwaysThrow(0);           // NB: we never get past this point
         alwaysThrow("hello");
      } catch (NumException e) {
         System.out.println("Got NumException: " + e.getMessage());
      } catch (StringException e) {
         System.out.println("Got StringException: " + e.getMessage());
      } finally {
         System.out.println("Cleaning up");
      }
   }
   public static void alwaysThrow(int arg) throws NumException {
      throw new NumException("don't call me with an int arg!");
   }
   public static void alwaysThrow(String arg) throws StringException {
      throw new StringException("don't call me with a String arg!");
   }
}
```

Defining Exceptions

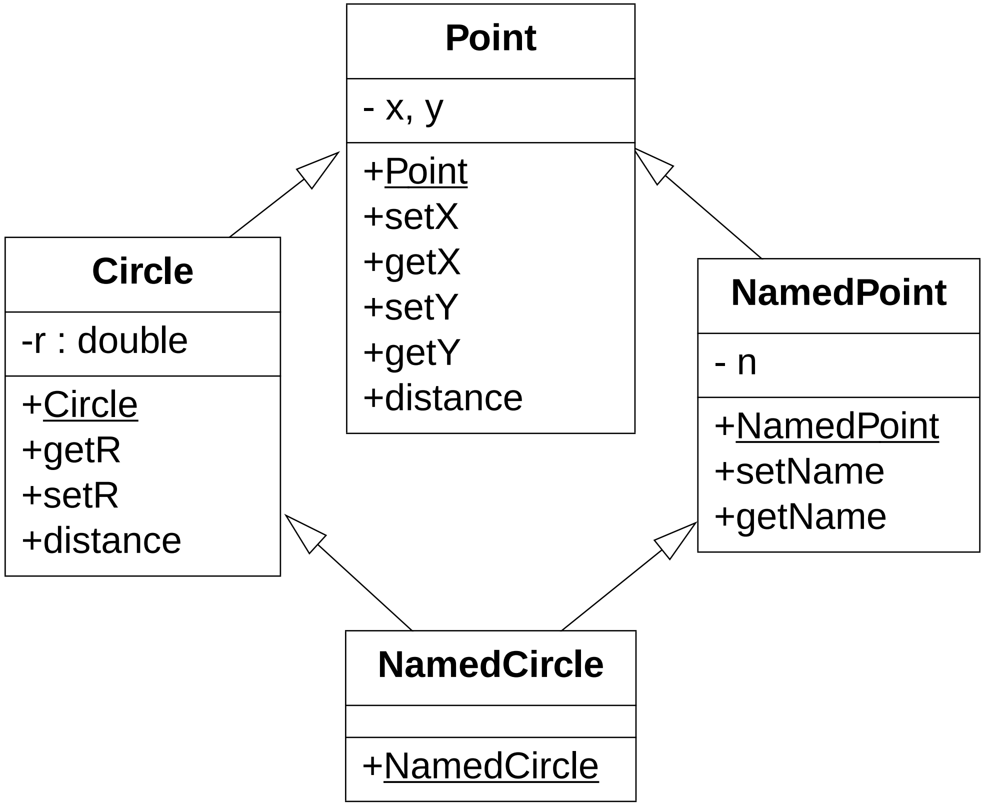
You can define your own exception classes that inherit from Exception

Typically, you will only define constructors:

```
// Most exception classes look like this:
public class NumException extends Exception {
   public NumException() { super(); }
   public NumException(String s) { super(s); }
}
public class StringException extends Exception {
   public StringException() { super(); }
   public StringException(String s) { super(s); }
}
```

Multiple Inheritance

Although conceptually elegant, multiple inheritance poses significant pragmatic problems for language designers:



Which version of distance() should be inherited by NamedCircle?

Interfaces

An interface declares methods but provides no implementation:

```
interface Named {
   public void setName (String name);
   public String getName ( );
}
```

A Java class can extend at most one superclass, but may implement multiple interfaces:

```
public class NamedCircle extends Circle implements Named {
   private NamedObject n; // object composition vs. inheritance
   public NamedCircle (double xCoord, double yCoord, double radius, String name) {
      super(xCoord, yCoord, radius); // call Circle constructor
      n = new NamedObject(name);     // compose a NamedObject instance
   }
   public void setName (String name) { n.setName(name); } // forwarding
   public String getName ( ) { return n.getName(); }
}
```

Reusable behaviour can be encapsulated as a separate class:

```
public class NamedObject implements Named {
   private String n;
   public NamedObject (String name) { n = name; }
   public void setName (String name) { n = name; }
   public String getName ( ) { return n; }
}
```

Overriding and Overloading

Overridden methods have the same name and argument types

Overloaded methods have the same name but different argument types

```
public class A {
   public void f (float x)   { System.out.println("A.f(float)"); }
   public void g (float x)   { System.out.println("A.g(float)"); }
}

public class B extends A {
   public void f (float x)   { System.out.println("B.f(float)"); }
   public void g (int x)     { System.out.println("B.g(int)"); }
}
```

Overloaded methods are disambiguated by their arguments:

```
B b = new B();    // both dynamic and static type B
A a = b;          // static type is A but dynamic type is B

b.f(3.14f);       // B.f(float) -- overridden
b.f(3);           // B.f(float) -- 3 is converted to 3.0
b.g(3.14f);       // A.g(float) -- not overridden
b.g(3);           // B.g(int) -- overloaded

a.f(3.14f);       // B.f(float) -- overridden
a.f(3);           // B.f(float) -- 3 is converted to 3.0
a.g(3.14f);       // A.g(float) -- not overridden
a.g(3);           // A.g(float) -- g(int) does not exist in SuperClass!
```

Arrays

Arrays are polymorphic objects:

- ❑ Can declare arrays of any type

```
int[] array1;
MyObject s[];
```

- ❑ Can build array of arrays

```
int a[][] = new int[10][3];
a.length --> 10
a[0].length --> 3
```

Creating arrays

- ❑ An empty array:

```
int list[] = new int [50];
```

- ❑ Pre-initialized:

```
String names[] = { “Marc”, “Tom”, “Pete” };
```

- ❑ Cannot create static compile time arrays

```
int nogood[20];  // compile time error
```

Arrays and Generics

Arrays are the only polymorphic containers in Java:

```
Point [] pa = new Point[3];
pa[0] = new Point(3,4);
pa[1] = new Point(5,12);
Point p = pa[0];             // ok -- pa is an array of Points
```

It is not possible to program other kinds of polymorphic containers:

```
Stack s = new Stack();       // defined in package java.util
s.push(pa[0]);
s.push(pa[1]);
// p = s.pop();              // compile-time error -- s.pop() returns an Object
p = (Point) s.pop();         // ok -- run-time cast
```

Summary

You Should Know The Answers To These Questions:

- ❑ What are the similarities and differences between Java and C++?
- ❑ What role do packages play in Java?
- ❑ When can an object access a `private` instance variable of another object?
- ❑ Why should a `super` constructor be called when constructing a subclass instance?
- ❑ What is dynamic binding? Why are `static` methods not dynamically bound?
- ❑ What is the difference between `protected` and `private protected`?
- ❑ What is the difference between overriding and overloading?

Can You Answer The Following Questions?

- ✎ *What are the similarities between Java and Eiffel?*
- ✎ *How can an object gain access to a* `private` *instance variable of another object?*
- ✎ *What exactly is the difference between a pointer and a reference?*
- ✎ *Why does Java (need to) support explicit type-casting?*
- ✎ *What is the difference between an* `interface` *and an* `abstract` *class?*

2. Java Applets and Threads

Overview

- ❑ The Java API
- ❑ Applets and events
- ❑ Creating and synchronizing threads

Texts:

- ❑ David Flanagan, *Java in a Nutshell*, O'Reilly, 1996
- ❑ Mary Campione and Kathy Walrath, *The Java Tutorial* , The Java Series, Addison-Wesley, 1996
- ❑ Doug Lea, *Concurrent Programming in Java — Design principles and Patterns*, The Java Series, Addison-Wesley, 1996

The Java API

java.lang contains essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads. This is the only package that is automatically imported into every Java program.

java.awt Abstract Windowing Toolkit

java.applet enables the creation of applets through the Applet class.

java.io provides classes to manage input and output streams to read data from and write data to files, strings, and other sources.

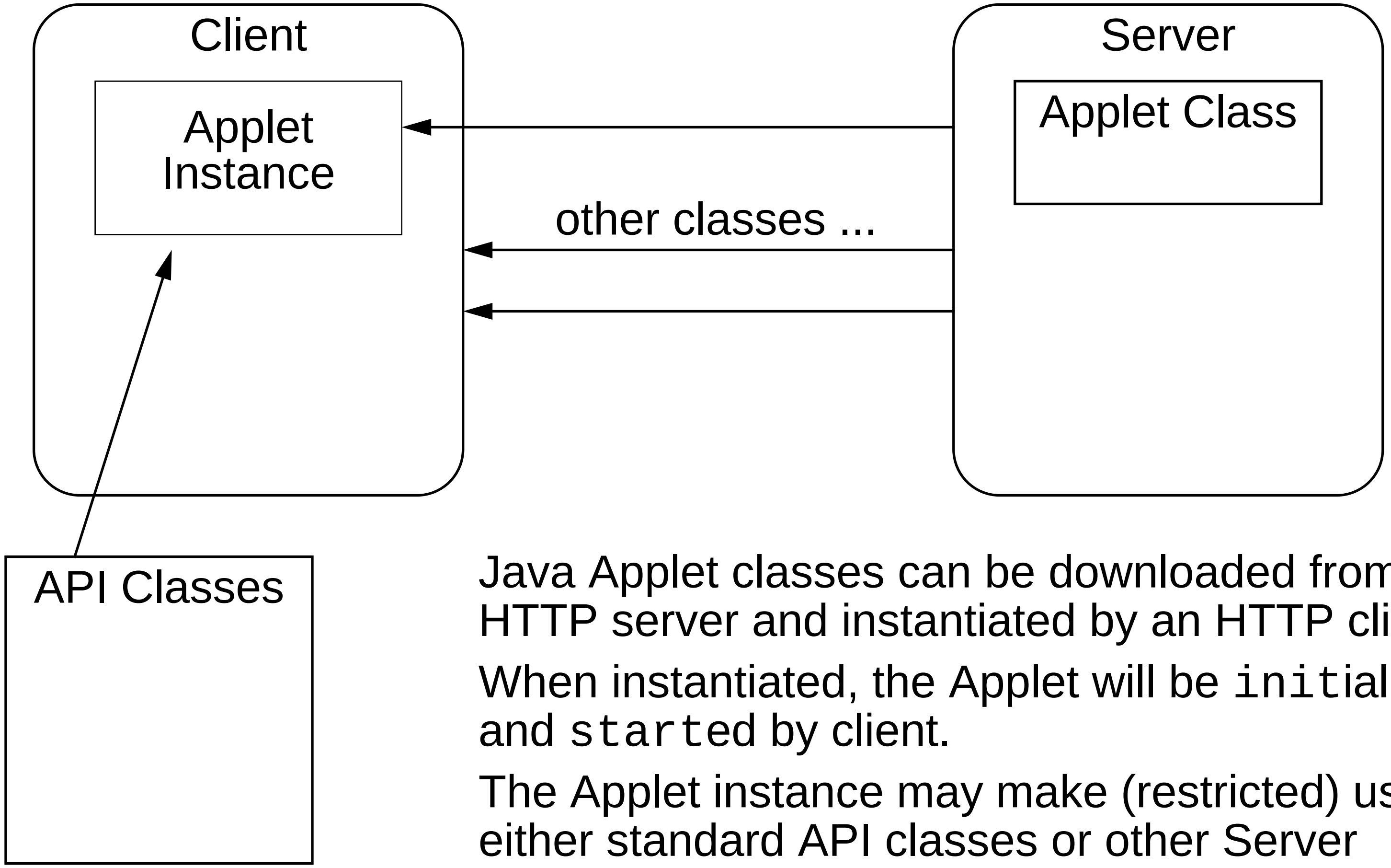
java.util contains miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, etc.

java.net provides network support, including URLs, TCP sockets, UDP sockets, IP addresses, and a binary-to-text converter.

java.awt.image classes for managing image data.

java.awt.peer connects AWT components to their platform-specific implementations (such as Motif widgets or Microsoft Windows controls).

Applets



Java Applet classes can be downloaded from an HTTP server and instantiated by an HTTP client.

When instantiated, the Applet will be `initialized` and `started` by client.

The Applet instance may make (restricted) use of either standard API classes or other Server classes to be downloaded dynamically.

NB: objects are *not* downloaded, only classes!

The Hello World Applet

The simplest Applet:

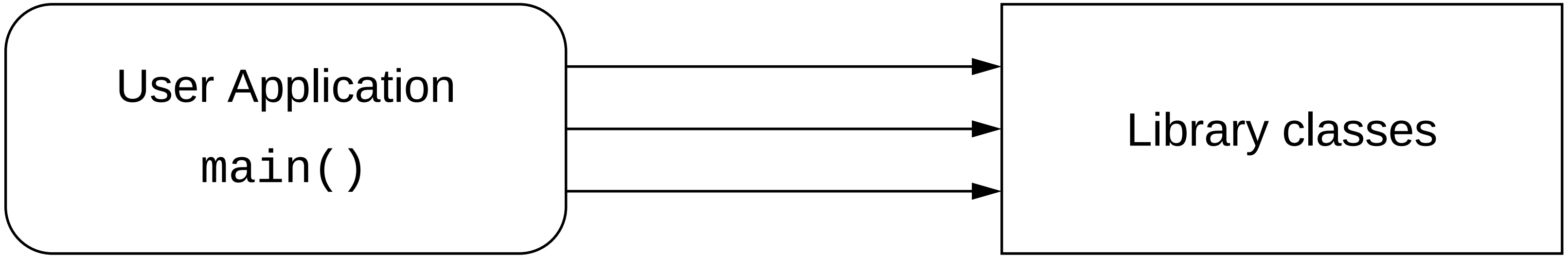
```
// From Java in a Nutshell, by David Flanagan.

import java.applet.*;  // To extended Applet
import java.awt.*;     // Abstract windowing toolkit

public class HelloApplet extends Applet {
   // This method displays the applet.
   // The Graphics class is how you do all drawing in Java.
   public void paint(Graphics g) {
      g.drawString("Hello World", 25, 50);
   }
}                      // NB: there is no main() method!
```

HTML applet inclusion:

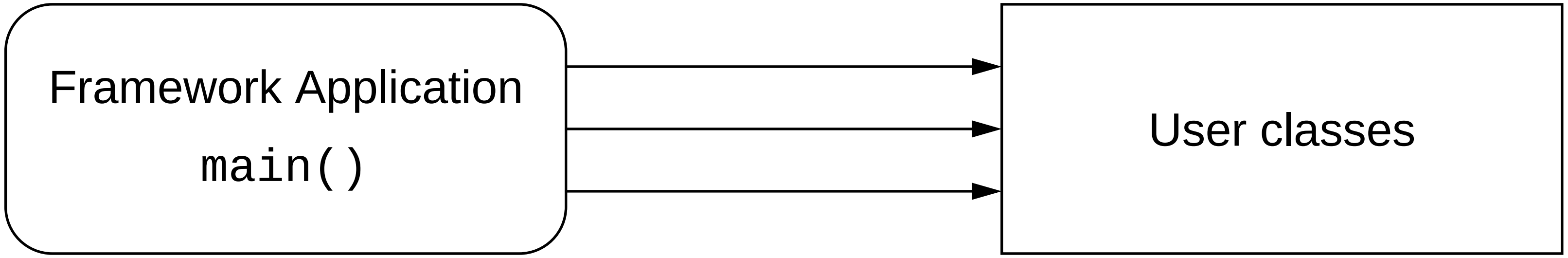
```
<title>Hello Applet</title>
<hr>
<applet codebase="HelloApplet.out" code="HelloApplet.class" width=200 height=200>
</applet>
<hr>
<a href="HelloApplet.java">The source.</a>
```

Frameworks vs. Libraries

In traditional application architectures, user applications make use of library functionality in the form of procedures or classes:



A framework reverses the usual relationship between generic and application code. Frameworks provide *both* generic functionality *and* application architecture:



Essentially, a framework says: “Don’t call me — I’ll call you.”

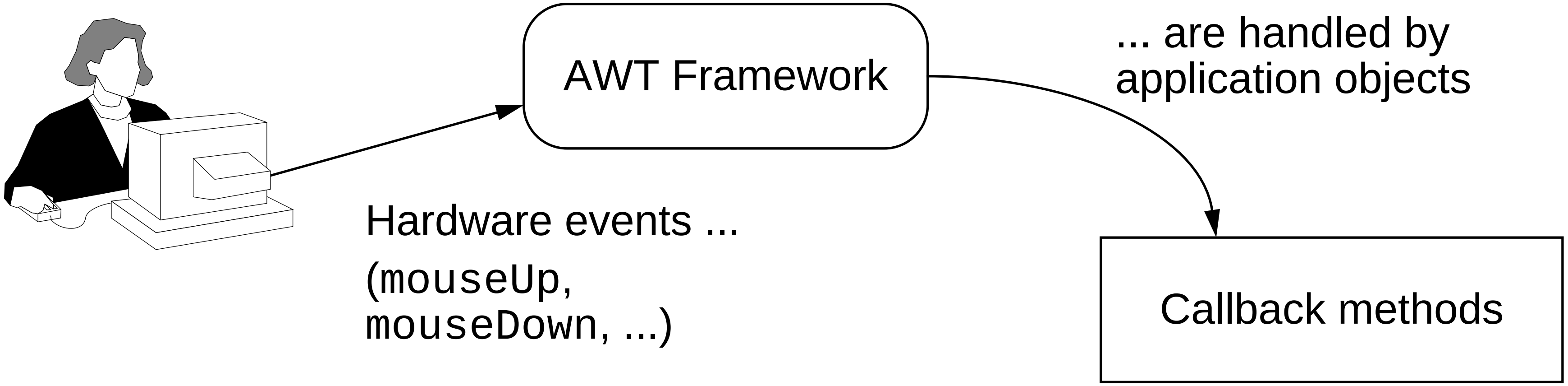
Standalone Applets

An Applet is just a user object instantiated by the Applet framework:

```
// Adapted from Java in a Nutshell, by David Flanagan.
// A simple example of directly instantiating an Applet.

import java.applet.*;
import java.awt.*;

public class HelloStandalone {
   public static void main(String args[]) {
      Applet applet = new HelloApplet();
      Frame frame = new AppletFrame("Hello Applet", applet, 300, 300);
   }
}

class AppletFrame extends Frame {
   public AppletFrame(String title, Applet applet, int width, int height) {
      super(title);                // Create the Frame with the specified title.

      this.add("Center", applet);  // Add the applet to the window.
      this.resize(width, height);  // Set the window size.
      this.show();                 // Pop it up.

      applet.init();               // Initialize and start the applet.
      applet.start();
   }
}
```

Events

Instead of actively checking for GUI events, you can define callback methods that will be invoked when your GUI objects receive events:


AWT Framework
... are handled by application objects
Hardware events ...
(mouseUp, mouseDown, ...)
Callback methods

`Component` is the superclass of all GUI components (including `Frame` and `Applet`) and defines all the callback methods that components must implement.

The Scribble Applet

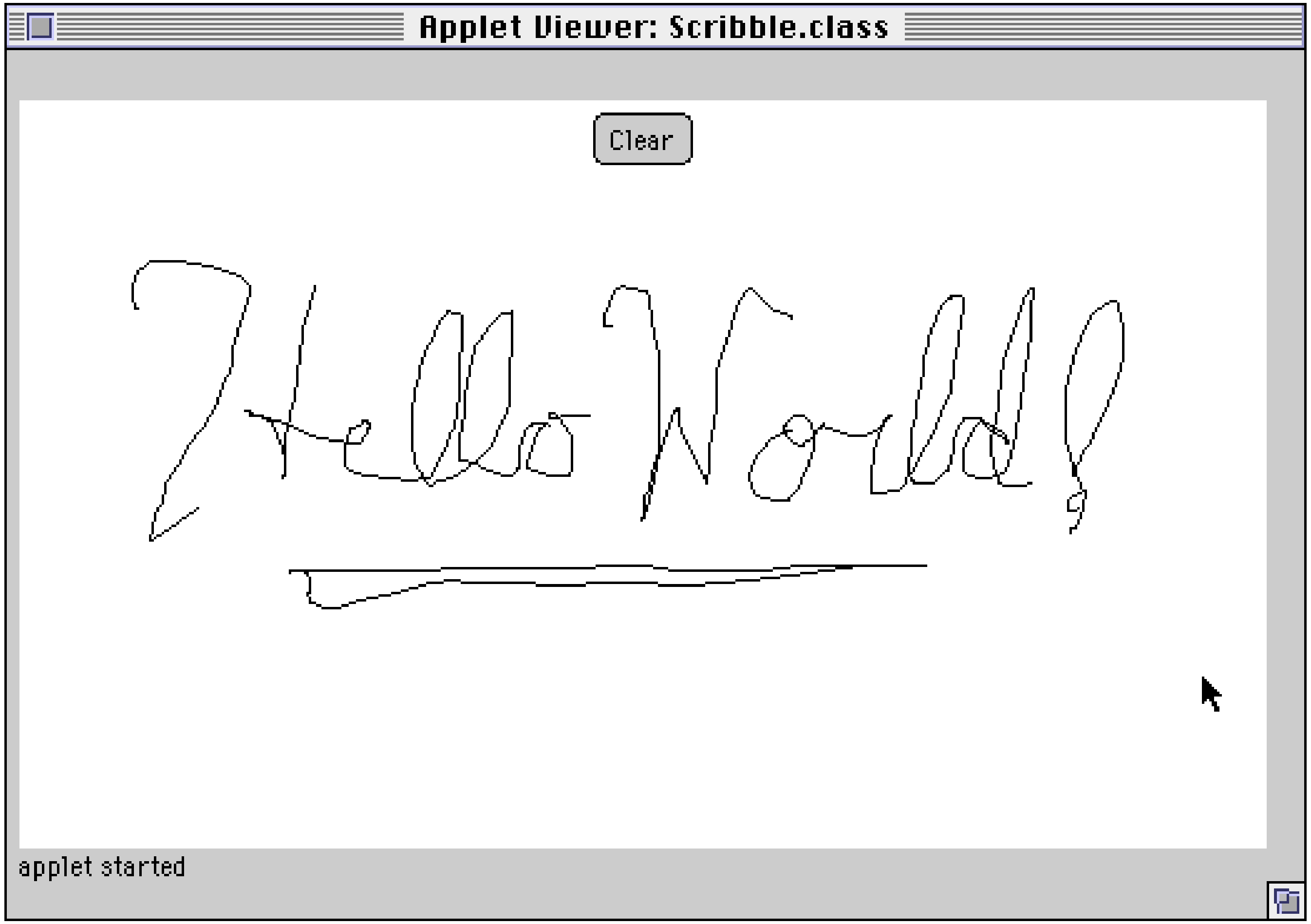
`Scribble` is a simple Applet that supports drawing by dragging the mouse:

```
// Adapted from Java in a Nutshell, by David Flanagan.

import java.applet.*;
import java.awt.*;

public class Scribble extends Applet {
   private int last_x = 0;
   private int last_y = 0;
   private Button clear_button;

   // Called to initialize the applet.
   public void init() {
      this.setBackground(Color.white);            // Set the background colour
      clear_button = new Button("Clear");         // Create a Button
      clear_button.setForeground(Color.black);
      clear_button.setBackground(Color.lightGray);
      this.add(clear_button);                     // Add it to the Applet
   }
```

Responding to Events

```
   // Called when the user clicks the mouse to start a scribble
   public boolean mouseDown(Event e, int x, int y) {
      last_x = x; last_y = y; return true; // Always return true if event handled
   }
   // Called when the user scribbles with the mouse button down
   public boolean mouseDrag(Event e, int x, int y) {
      Graphics g = this.getGraphics();
      g.setColor(Color.black); g.drawLine(last_x, last_y, x, y);
      last_x = x; last_y = y; return true;
   }
   // Called when the user clicks the button
   public boolean action(Event event, Object arg) {
      // If the Clear button was clicked on, handle it.
      if (event.target == clear_button) {
         Graphics g = this.getGraphics();
         Rectangle r = this.bounds();
         g.setColor(this.getBackground());
         g.fillRect(r.x, r.y, r.width, r.height);
         return true;
      } // Otherwise, let the superclass handle it.
      else return super.action(event, arg);
   }
}
```

Running the Scribble Applet



applet started

Threads

A `Thread` defines its behaviour in its `run` method, but is started by calling `start()`:

```
// Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.

class TwoThreadsTest {
   public static void main (String[] args) {
      new SimpleThread("Jamaica").start();    // Instantiate, then start
      new SimpleThread("Fiji").start();
   }
}

class SimpleThread extends Thread {
   public SimpleThread(String str) {
      super(str);                              // Call Thread constructor
   }
   public void run() {                         // What the thread does
      for (int i = 0; i < 10; i++) {
         System.out.println(i + " " + getName());
         try {
            sleep((int)(Math.random() * 1000));
         } catch (InterruptedException e) { }
      }
      System.out.println("DONE! " + getName());
   }
}
```

Running the TwoThreadsTest

```
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Jamaica
2 Fiji
3 Jamaica
3 Fiji
4 Jamaica
4 Fiji
5 Jamaica
6 Jamaica
5 Fiji
6 Fiji
7 Fiji
7 Jamaica
8 Jamaica
9 Jamaica
8 Fiji
DONE! Jamaica
9 Fiji
DONE! Fiji
```

In this implementation of Java, the execution of the two threads is interleaved.

This is *not* guaranteed for all implementations!

✎ *Why are the output lines never garbled?*

E.g.

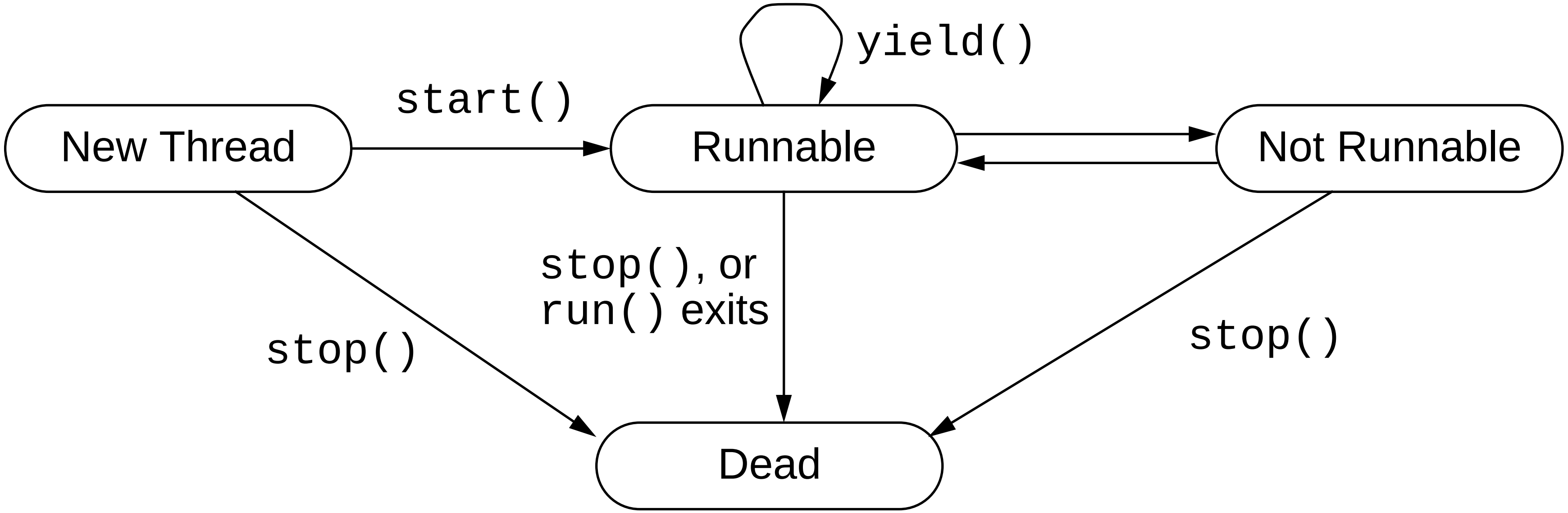
```
00 JaFimajicai
...
```

java.lang.Thread

The Thread class encapsulates all information concerning running threads of control:

```
public class java.lang.Thread
   extends java.lang.Object implements java.lang.Runnable
{
   public Thread();                     // Public constructors
   public Thread(Runnable target);
   public Thread(Runnable target, String name);
   public Thread(String name);
...
   public static void sleep(long millis)// Current thread sleeps
            throws InterruptedException;
   public static void yield();          // Yield control (equal priority)
...
   public final String getName();
   public void run();                   // “main()” method
   public synchronized void start();    // Starts a thread running
   public final void suspend();         // Temporarily halts a thread
   public final void resume();          // Allow to resume after suspend()
   public final void stop();            // Throws a ThreadDeath error
   public final void join()             // Waits for thread to die
            throws InterruptedException;
...
}
```

Transitions between Thread States



Runnable →	← *Not Runnable*
`sleep()`	time elapsed
`suspend()`	`resume()`
`wait()`	`notify()` or `notifyAll()`
blocked on I/O	I/O completed

java.lang.Runnable

Since multiple inheritance is not supported, it is not possible to inherit from both `Thread` and from another class providing useful behaviour (like `Applet`).

In these cases it is sufficient to define a class that implements the Runnable interface, and to call the Thread constructor with an instance of that class as a parameter:

```
public interface java.lang.Runnable
{
   public abstract void run();
}
```

Creating Threads

A Clock object updates the time as an Applet with its own Thread:

```
import java.awt.Graphics; // Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.
import java.util.Date;

public class Clock extends java.applet.Applet implements Runnable {

   Thread clockThread = null;

   public void start() {
      if (clockThread == null) {
         clockThread = new Thread(this, "Clock");      // NB: creates its own thread
         clockThread.start();
      }
   }

   public void run() {
      // loop terminates when clockThread is set to null in stop()
      while (Thread.currentThread() == clockThread) {
         repaint();
         try { clockThread.sleep(1000); }
         catch (InterruptedException e){ }
      }
   }

   public void paint(Graphics g) {
      Date now = new Date();
      g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10);
   }

   public void stop() { clockThread = null; }
}
```

Synchronization

Without synchronization, an arbitrary number of threads may be running at any time within the methods of an object.

One can either declare an entire method to be synchronized with other synchronized methods of an object:

```
public class PrintStream extends FilterOutputStream {
   ...
   public synchronized void println(String s);// Only one may run at a time
   public synchronized void println(char c);
   ...
}
```

or an individual block within a method may be synchronized with respect to some object:

```
synchronized (resource) { // Lock resource before using it
   ...
}
```

wait and notify

Sometimes threads must be delayed until a resource is in a suitable state:

```
class Slot {                            // Implements a one-slot buffer
   private int contents;
   private boolean available = false;   // the condition variable

   public synchronized int get() {      // put contents, if available
      while (available == false) {
         try { wait(); }                // wait until there is something to get()
         catch (InterruptedException e) { }
      }
      available = false;
      notify();                         // wake up the producer
      return contents;
   }

   public synchronized void put(int value) { // put value, if there is room
      while (available == true) {
         try { wait(); }                // wait until there is room to put()
         catch (InterruptedException e) { }
      }
      contents = value;
      available = true;
      notify();                         // wake up the consumer
   }
}
```

java.lang.Object

Unlike `synchronized`, `wait()` and `notify()` are methods rather than keywords:

```
public class java.lang.Object
{
   public Object();
   public boolean equals(Object obj);
   public final Class getClass();
   public int hashCode();
   public String toString();
   public final void wait()
            throws InterruptedException, IllegalMonitorStateException;
   public final void wait(long timeout)
            throws InterruptedException, IllegalMonitorStateException;
   public final void wait(long timeout, int nanos)
            throws InterruptedException, IllegalMonitorStateException;
   public final void notify() throws IllegalMonitorStateException;
   public final void notifyAll() throws IllegalMonitorStateException;
   protected Object clone()
            throws CloneNotSupportedException, OutOfMemoryException;
   protected void finalize() throws Throwable;
}
```

Other Facilities

Java provides a large number of additional facilities for concurrent and distributed programming:

Pipes: Data-flow between threads is supported by various `Pipe` classes.

Thread Priorities: Higher priority threads pre-empt those with lower priority.

Thread Groups: Threads belonging to the same group can be manipulated together.

Security Managers: Downloaded Applets are “untrusted” and are only allowed to perform restricted sets of actions.

Processes: New processes can be started (in a platform-dependent way).

Sockets: Standard classes in java.net.* support URL and socket connections.

Summary

You Should Know The Answers To These Questions:

- ❑ What are Applets and how are they instantiated?
- ❑ Why doesn’t an Applet need a `main()` method?
- ❑ What are events and callbacks?
- ❑ How can you define and start your own threads?
- ❑ How does a thread become *Runnable*?
- ❑ Why do we need a separate `Runnable` interface?
- ❑ Why do we need `wait()` and `notify()` in addition to `synchronized`?

Can You Answer The Following Questions?

- ✎ *Why doesn’t the Java language provide a way to download objects?*
- ✎ *Why should an event handler* `eh` *always call* `super.eh()` *if it fails to handle the event passed to it?*
- ✎ *What happens if we call the* `run()` *method of a thread instead of* `start()`*?*
- ✎ *What might happen if java.io.PrintStream.println weren’t* `synchronized`*?*