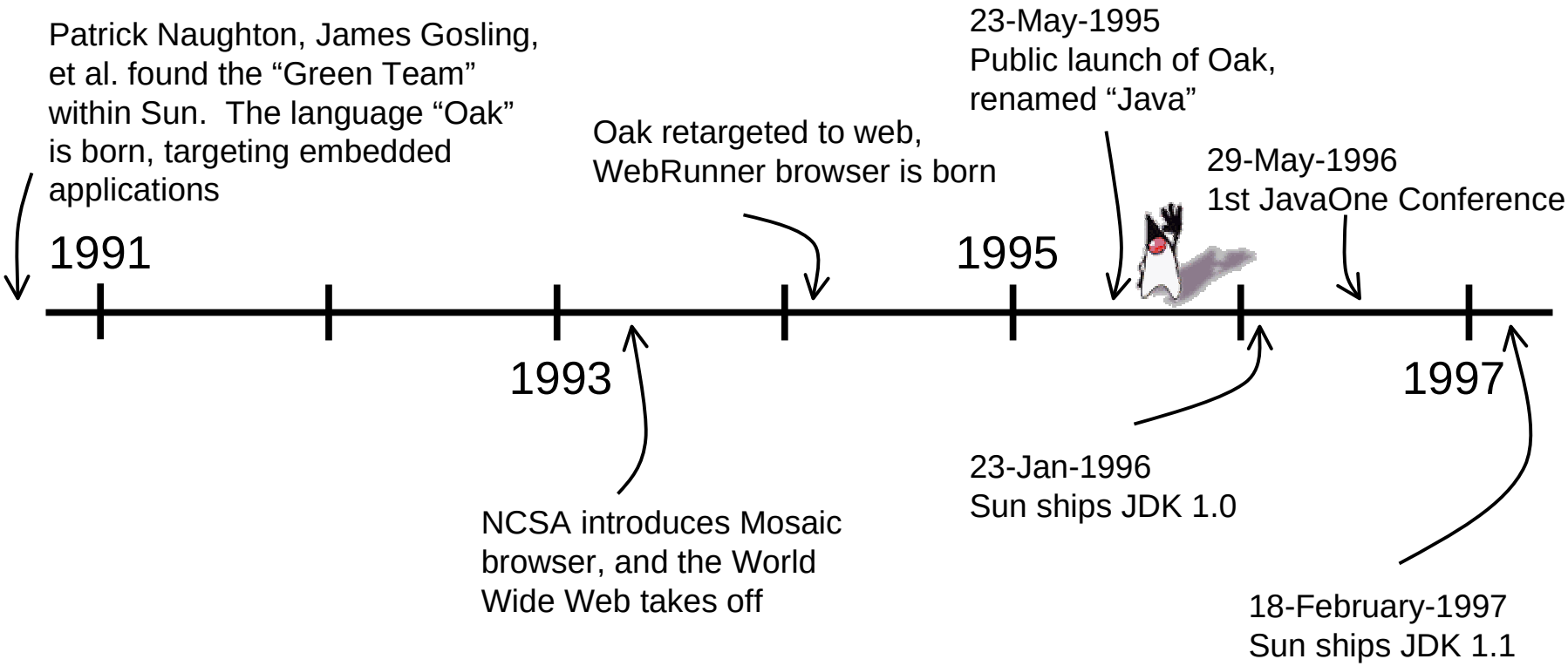
Java

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language."

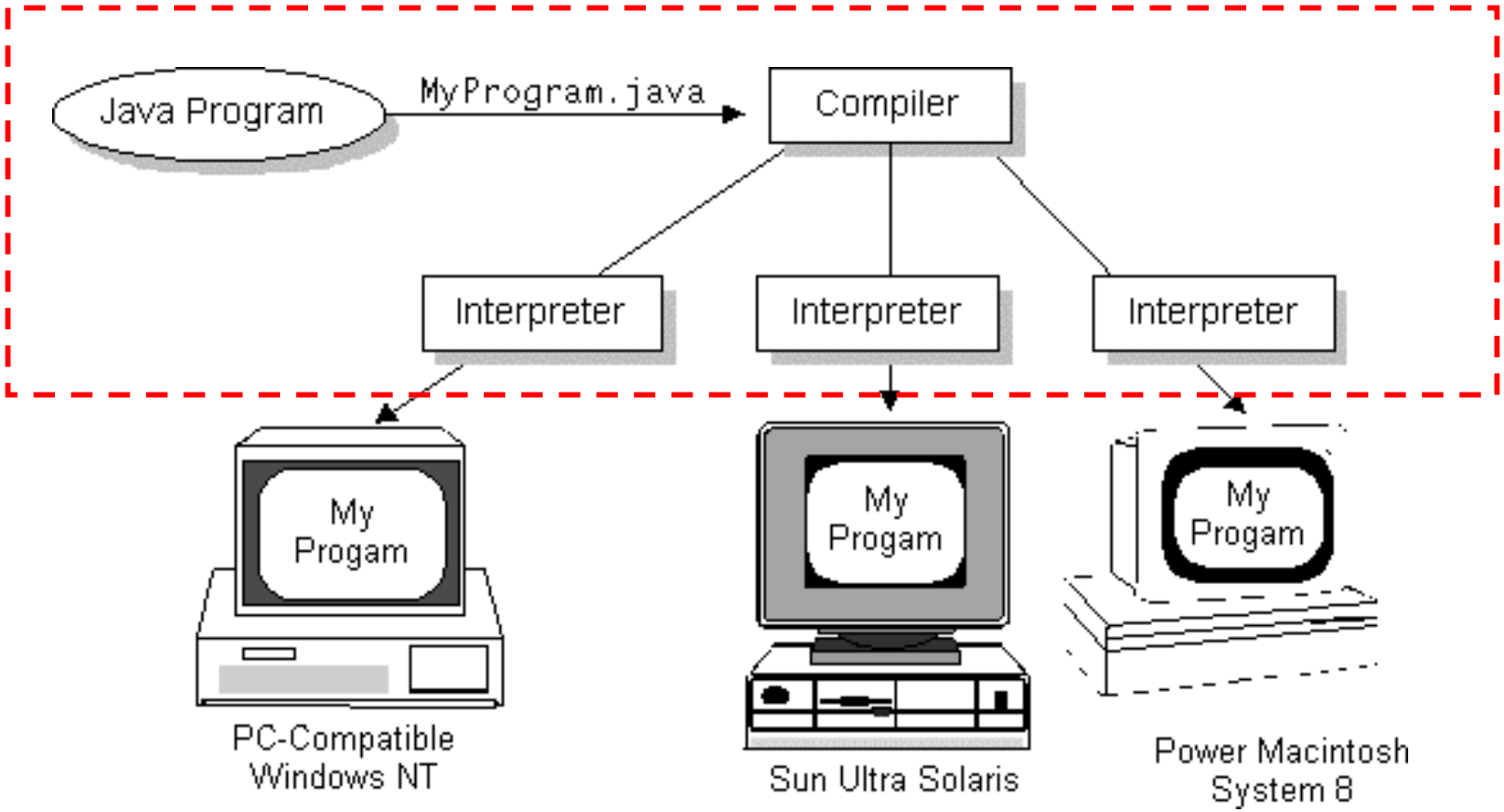
— Sun

Greg J. Badros
University of Washington
CSE-341, Spring 1998

Java: A Timeline



Java: The Language



Hello World!

HelloWorld.java

```
/** Application HelloWorld
    Just output "Hello World!" */

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

```
% javac HelloWorld.java
% java HelloWorld
Hello World!
```

Java vs. C++

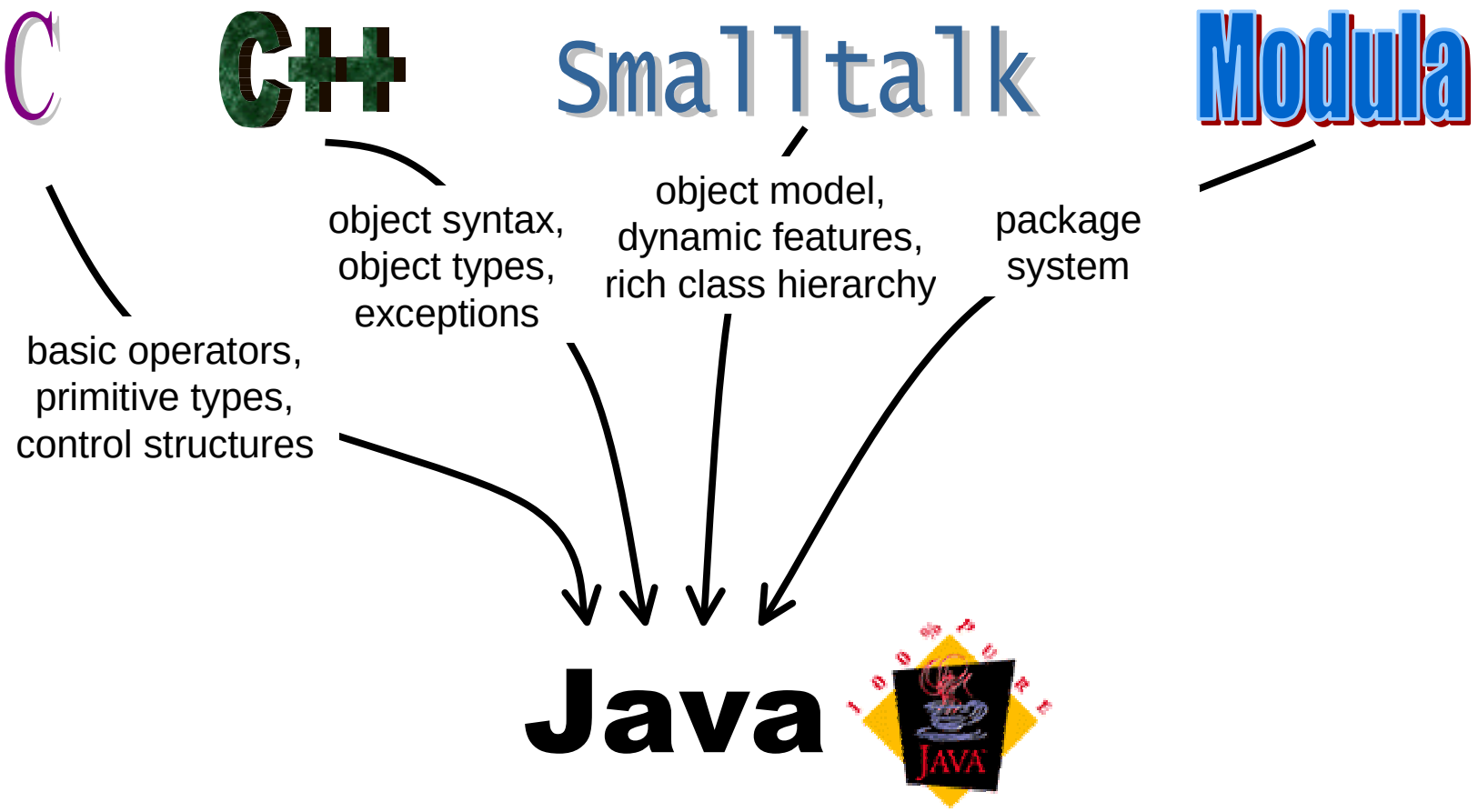
```
/** Application HelloWorld  */

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

```
// Application HelloWorld

#include <iostream.h>

int main(int argc, char* argv[]) {
   cout << “Hello World!” << endl;
}
```

Unlike C++, Java has....

- No global functions — everything is in a class!
- Real String objects — not just char[]
- No pointers — everything is a reference
- No operator-overloading
- No preprocessor — cpp not needed

Brewing Java



Java vs. Smalltalk

```
int x = 50, y = 50;
Ball ball = new Ball(x, y);
PinballAnimationPane pap = new PinballAnimationPane();
pap.addObject(ball);
ball.animate();
```

```
| ball  pap  x  y  |

x := 50.  y := 50.
ball := Ball newX: x centerY: y.
pap := PinballAnimationPane new.
pap addObject: ball.
ball animate
```

Unlike Smalltalk, Java...

- Specifies types for all variables
- Permits primitive types such as int
- Has new keyword for creating objects
- Does not have keyword arguments
- Uses C operators
  - = is assignment
  - . operator for message sends

Java vs. C++, Revisited

```
Ball ball = new Ball(50, 50);
PinballAnimationPane pap = new PinballAnimationPane();
pap.addObject(ball);
ball.animate();
```

(pointer-style)

```
Ball *pball = new Ball(50,50);
PinballAnimationPane *pxpap = new PinballAnimationPane();
pxpap->addObject(pball);
pball->animate();
```

(reference-style)

```
Ball ball(50,50);                    // creates ball on stack
PinballAnimationPane xpap();         // creates xpap on stack
xpap.addObject(ball);                // calls: addObject(Ball &b);
ball.animate();
```

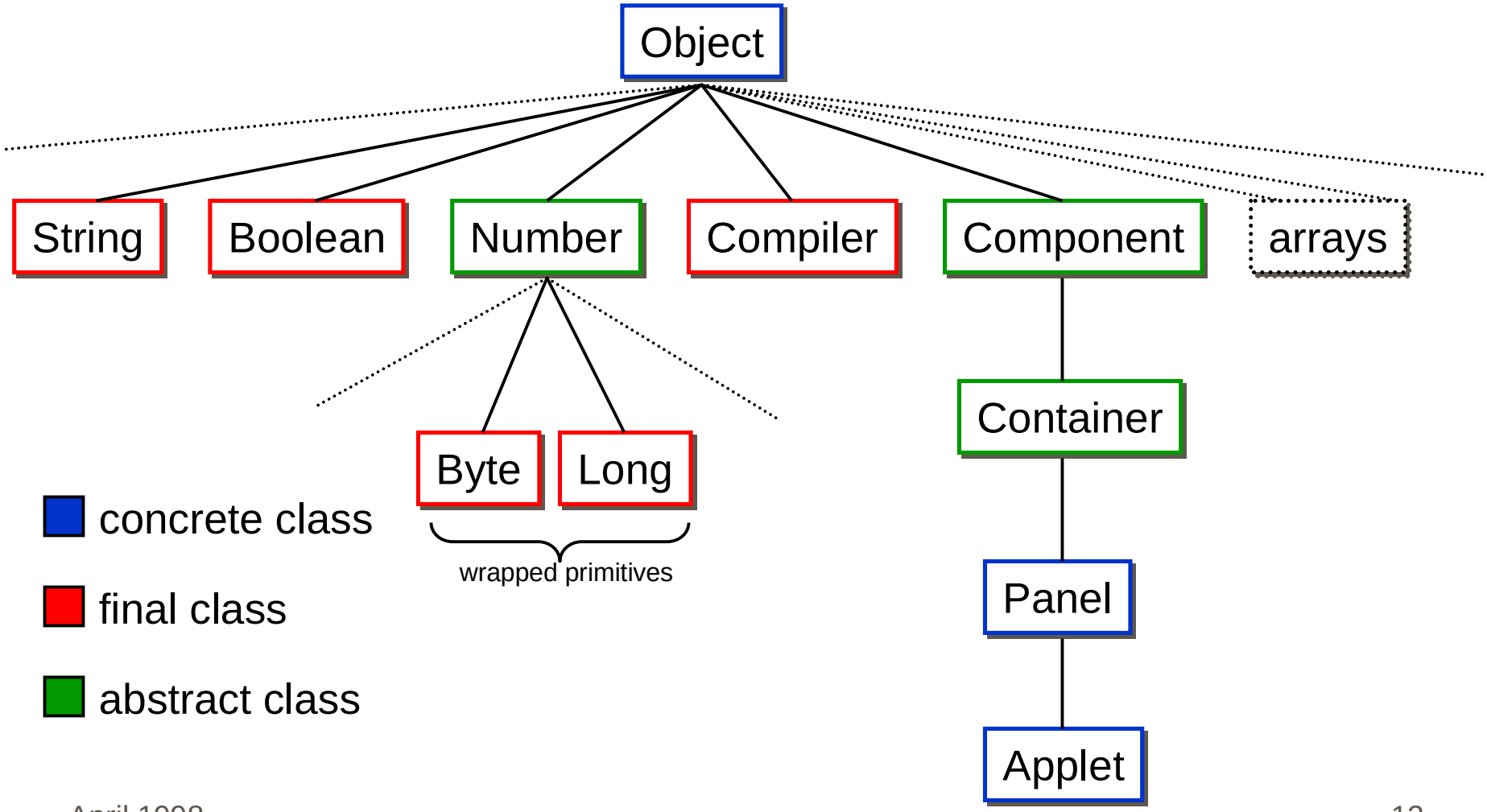
Java's Hybrid Object Model

- Primitive types on stack
  - May be *wrapped* or *boxed* into a real object

    Integer anInteger = new Integer(43);

    (useful for storing in java.util.*'s collections)
  - Unboxed primitives very similar to in C++
- All object instances live in the heap (**not** stack)
  - all object creation is done with new
  - No "delete" — Java uses garbage collection like Smalltalk, but also provides finalize() method

Java's Class Hierarchy



Java Documentation

HelloWorld Applet

HelloWorldApplet.java

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

public class HelloWorldApplet extends Applet {
    static final String message = "Hello World";
    private Font font;
    public void init() {   // one-time initialization
        font = new Font("Helvetica", Font.BOLD, 48);  }
    public void paint(Graphics g) {
        g.setColor(Color.purple);          g.fillOval(10, 10, 330, 100);
        g.setColor(Color.red);
        g.drawOval(10, 10, 330, 100);    g.drawOval(9, 9, 332, 102);
        g.drawOval(8, 8, 334, 104);      g.drawOval(7, 7, 336, 106);
        g.setColor(Color.black);           g.setFont(font);
        g.drawString(message, 40, 75);
    }  }
```


Ref: *Java In a Nutshell*, O'Reilly

Running the HelloWorld Applet

HelloWorldApplet.html

```
<APPLET   code="HelloWorldApplet.class"
          width=350 height=120> Java Missing
</APPLET>
```

Add "." to your $CLASSPATH, then
% appletviewer HelloWorldApplet.html

Run on the .html file

Methods: A Closer Look

```
public class Point {
   . . .
   public void move(int dx) {
      x += dx;
      moved();
   }
   private void moved() { . . }
   private int x, y;
}
```

```
public class Point {
   . . .
   public void move(int dx) {
      this.x += dx;
      this.moved();
   }
   private void moved() { . . }
   private int x, y;
}
```

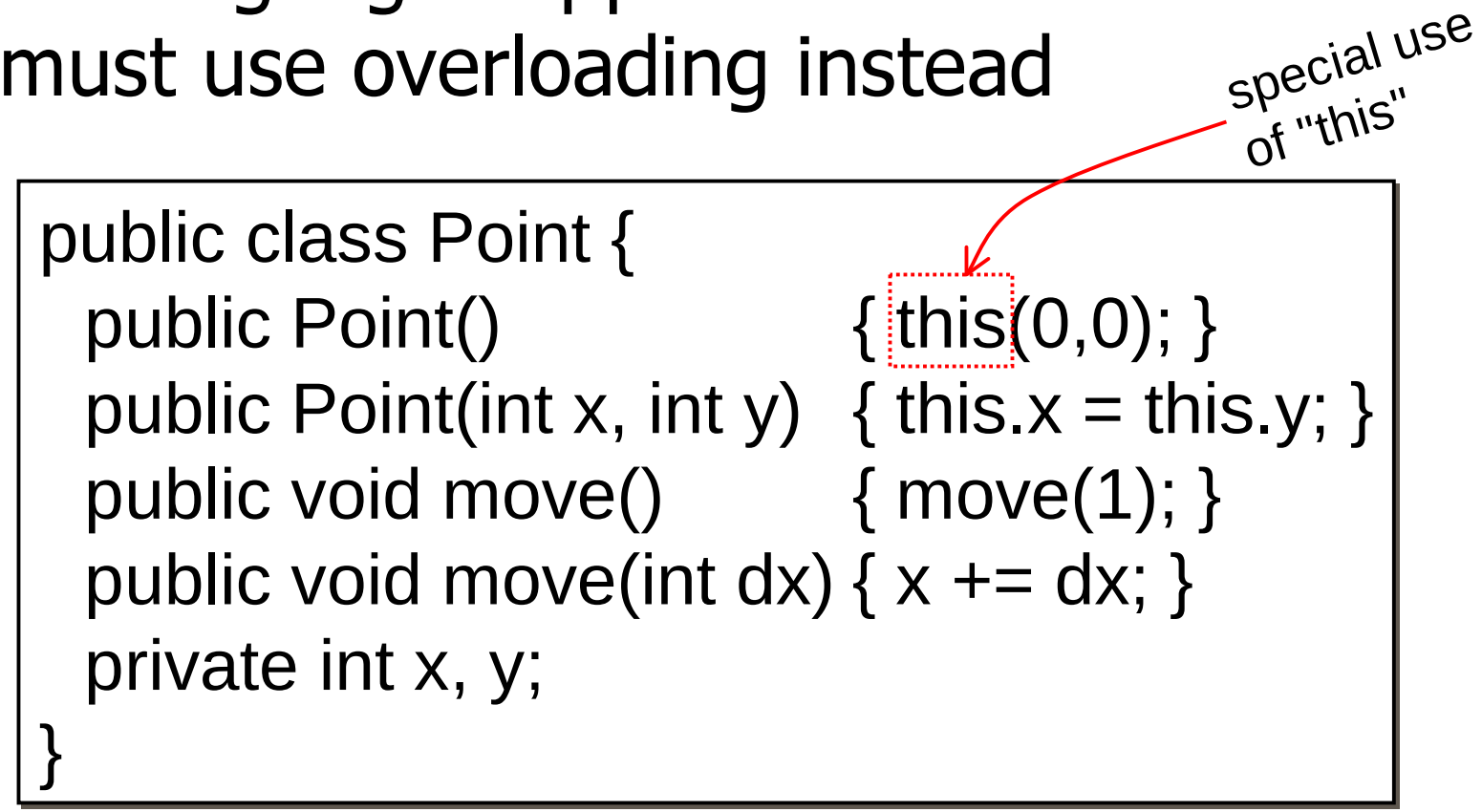
- this is implicit on instance fields and methods
  - can be explicit if the field is hidden by a local or formal
  - analogous to self in Smalltalk (though self is necessarily explicit)
- also super keyword, as in Smalltalk (no C++ :: operator)
  - also used for constructor chaining with arguments

More on Methods

- Instance methods (no static keyword)
  - have implicit this argument
  - can use super keyword
  - no need to use "->" operator as in C++ just . operator since this, super are references
- static (class) methods
  - do not have implicit this argument
  - cannot use the super keyword

Default Arguments

- No language support—
  must use overloading instead

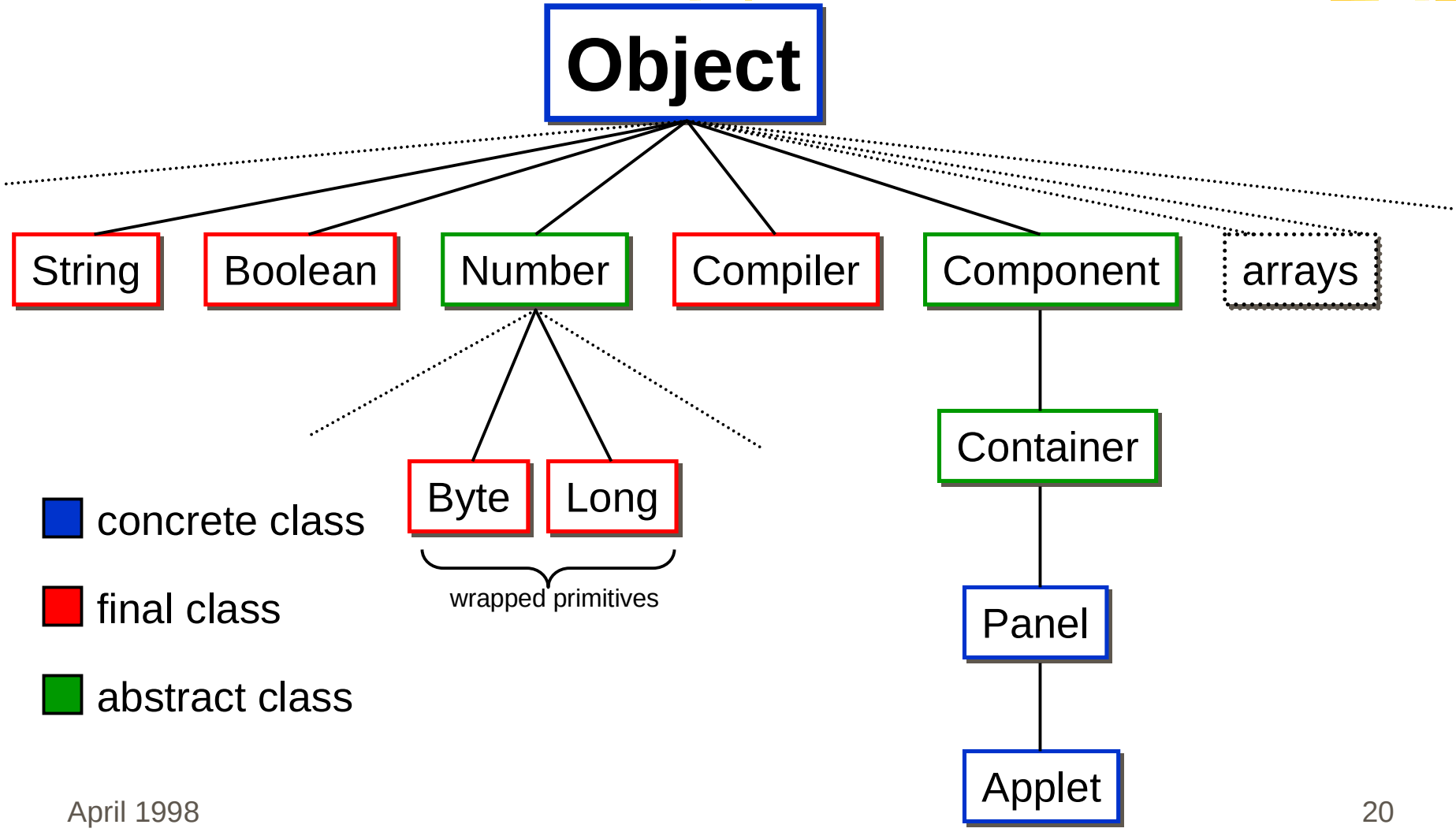


“Override” vs. “Overload”

- **Override**
  - replace a superclass’s method with a specialized version
  - signatures must match
    (including return type; C++ permits narrowing of return types, Java does not)

- **Overload**
  - write several methods for a given class with the same name
  - language can disambiguate based on number or types of arguments

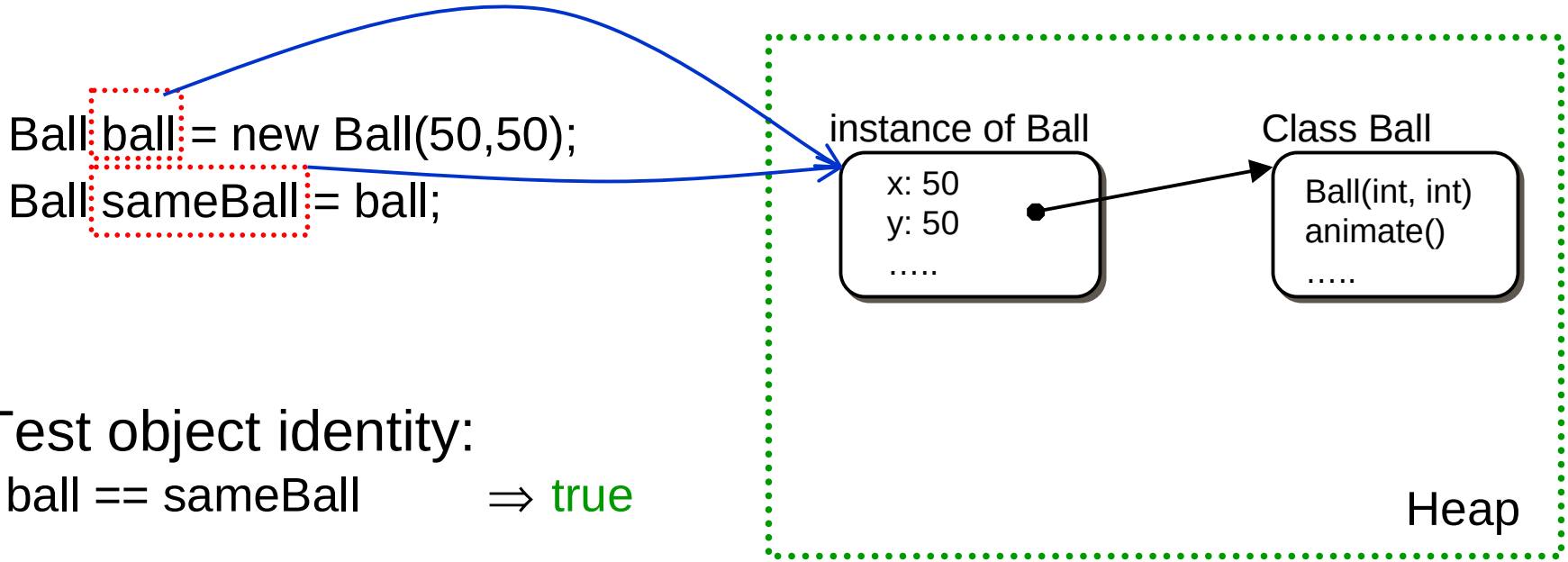
Java's Class Hierarchy



What can an Object do for you today?

- Object clone()
  Return a duplicate copy of self
- boolean equals(Object obj)
  Return true if and only if self is value-equal to obj
- String toString()
  Return printable representation of self
- int hashCode()
  Return a reasonable hash code for self
- Class getClass()
  Return the class object for self

Objects and Identities



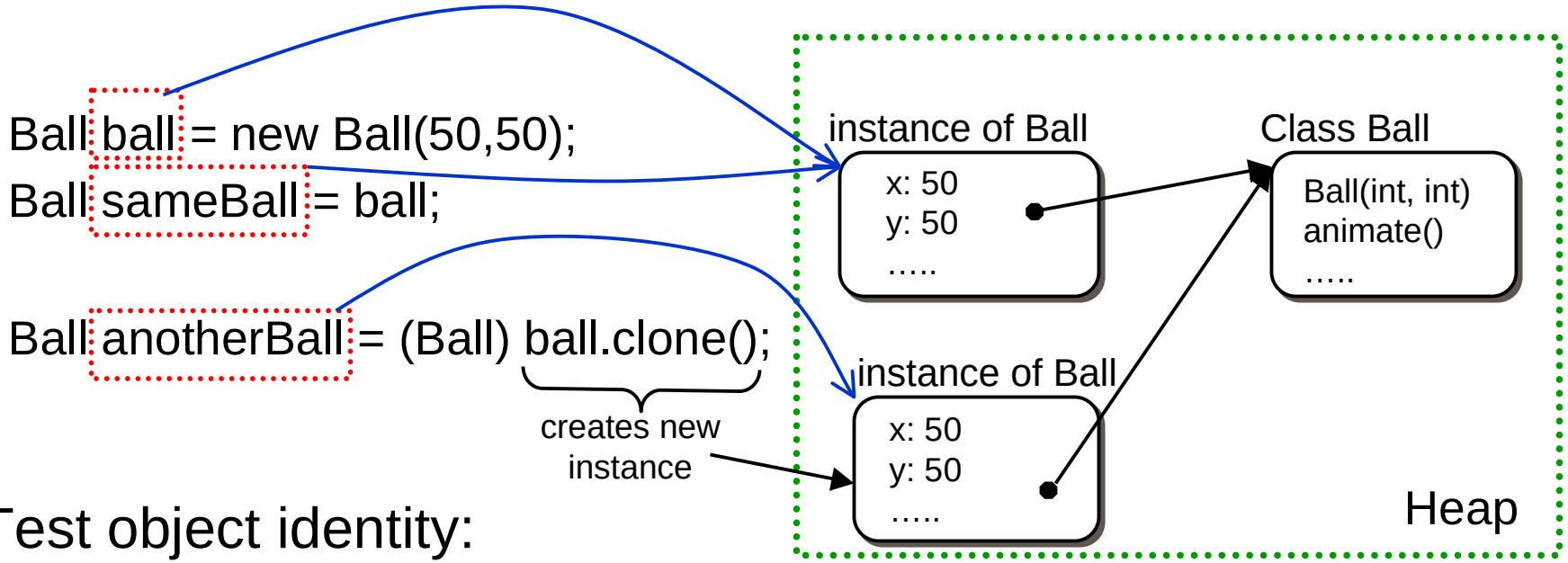
Test object identity:

ball == sameBall ⇒ true

Test object value's equality:

ball.equals(sameBall) ⇒ true

Cloning Objects



Test object identity:

ball == anotherBall ⇒ false

Test object value's equality:

ball.equals(anotherBall) ⇒ true

Changing a Ball

ball.setPosition(20,35);

sameBall

anotherBall

Test object identity:

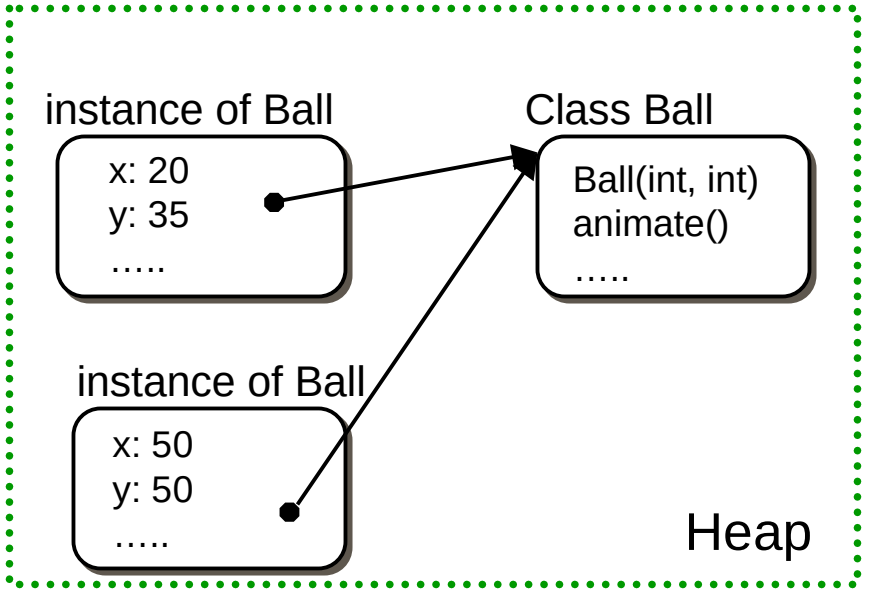
ball == sameBall ⇒ ???

ball == anotherBall ⇒ ???

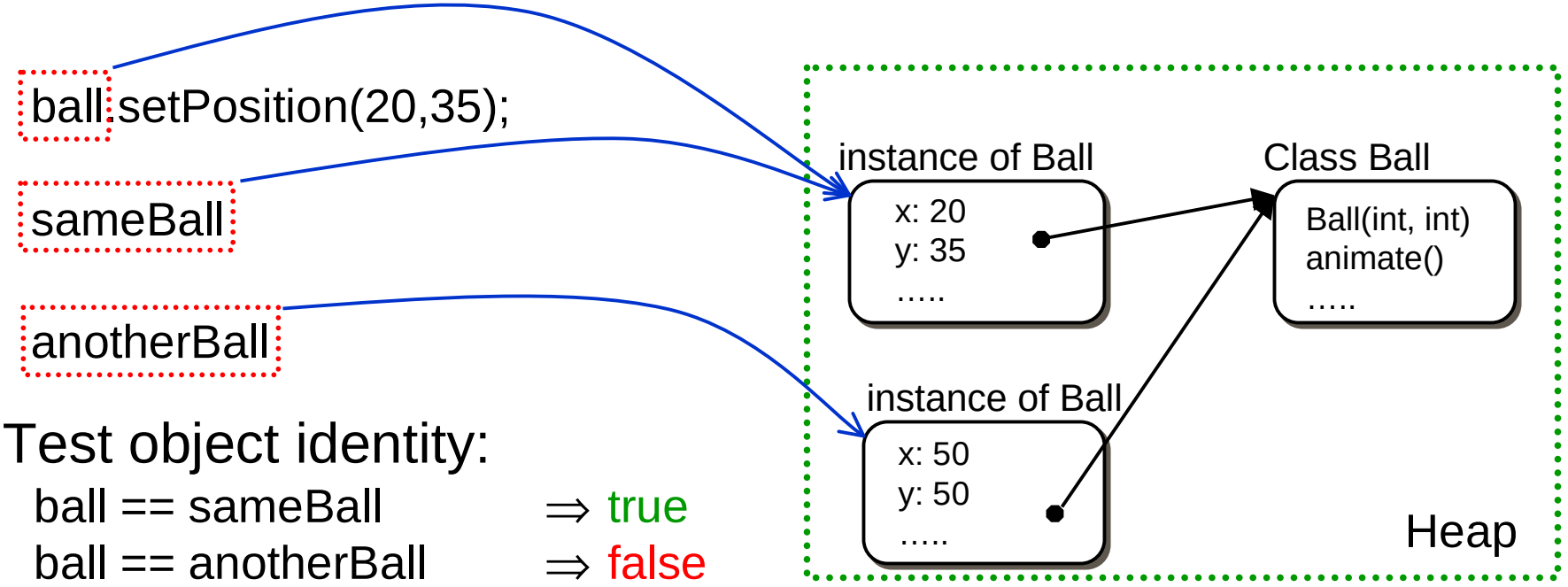
Test object value's equality:

ball.equals(sameBalll) ⇒ ???

ball.equals(anotherBall) ⇒ ???



Inequality in Balls!



Test object identity:

ball == sameBall ⇒ true

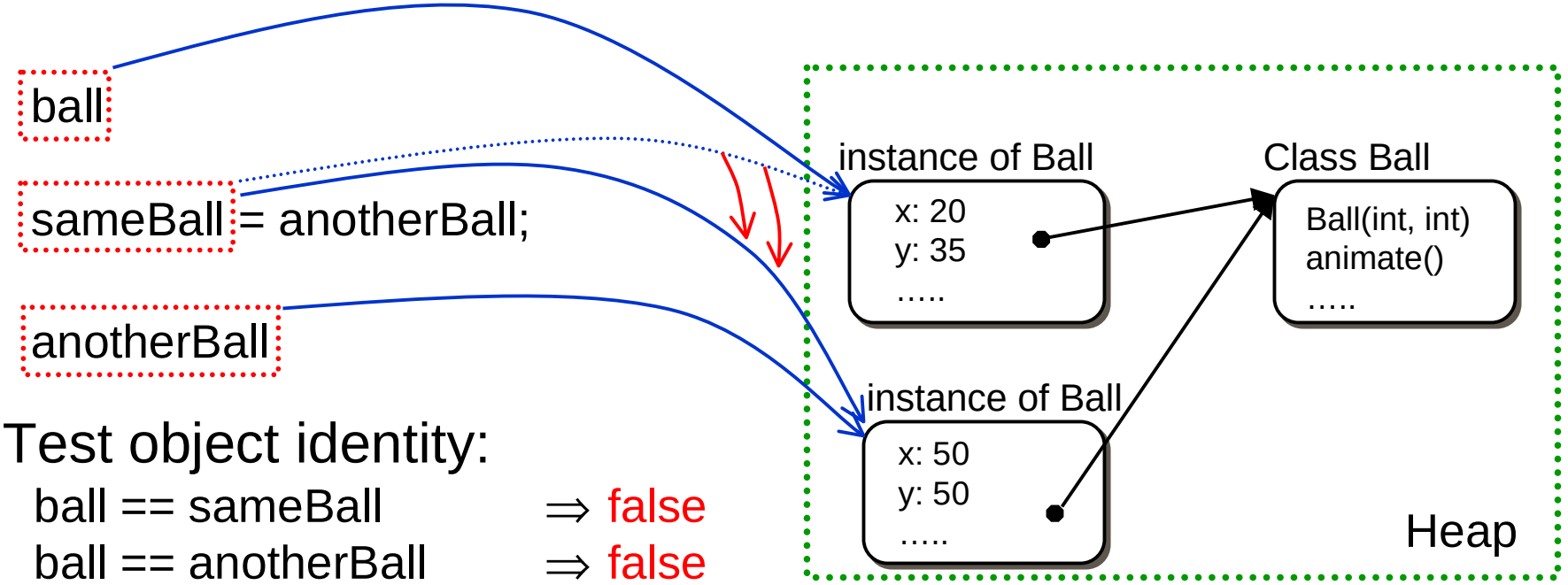
ball == anotherBall ⇒ false

Test object value's equality:

ball.equals(sameBalll) ⇒ true

ball.equals(anotherBall) ⇒ false

Assignment just changes the pointer



Test object identity:

ball == sameBall ⇒ false

ball == anotherBall ⇒ false

Test object value's equality:

ball.equals(sameBalll) ⇒ false

ball.equals(anotherBall) ⇒ false

Java variables hold...

- primitive

```
boolean foo;            // boolean, not bool as in C++
char aChar = 'a';       // 16 bit char (unicode)
```

- Object reference (may be null)

```
ColoredBall cball = new Ball();
Ball ball = cball;
```

- Array reference

```
int[] intArray = { 1, 2, 3, 4, 5, };
String[] strArray = { "Hello", "World", };
// same as
String[] strArray = new String[2];
strArray[0] = new String("Hello");
strArray[1] = new String("World");
```

String literals actually invoke constructor e.g., new String("World")

Arrays

Similar to:

- Java arrays are 1st-class Objects
- 0-indexed
- Bounds checking performed
- Store/Retrieve using [] operator

  strArray[0] = strArray[1];
- Have implicit length field

  strArray.length ⇒ 2

Identifiers

- Everything has a globally-unique name

Java.lang.String
Java.util.Hashtable
Java.applet.Applet
EDU.Washington.grad.gjb.cassowary.Variable.toString()

Package name — Class name — Method name

- Pretty wordy, so...

import statement

- Two forms:
  - import java.util.HashTable;
    Just make the HashTable class available from package java.util
  - import EDU.Washington.grad.gjb.cassowary.*;
    Make all classes from package available on demand
- Always an implicit "import java.lang.*"
- Permits using simple (short) names
  - Not like C++'s "#include"
  - More like C++'s "using *namespace* "

How Java Finds a Class...

- Package names mirror the directory structure
- package statement informs the compiler

```
./EDU/Washington/grad/gjb/cassowary/Variable.java
```

```
package EDU.Washington.grad.gjb.cassowary;

public class Variable extends AbstractVariable {
    . . .
}

class Helper { ... }
```

Compilation of Source File

```
% ls
Variable.java
% javac Variable.java
% ls
Variable.java
Variable.class
Helper.class
```

One java source file may create multiple .class files containing the byte-compiled code

Class Access Protection

```
package EDU.Washington.grad.gjb.cassowary;

public class Variable extends AbstractVariable {
    . . .
}

class Helper { ... }
```

- Only one public class per file
- No specifier ⇒ package protection

  visible to all classes in the package
  no "package" keyword — remember it is a statement

Not even friends can touch Java's private parts

```
public class Point {
   private int x, y;
   void setXY(int x, int y) {
      this.x = x; this.y = y;
   }
   protected void move(int x, int y) {
      setXY(this.x+x, this.y+y);
   }
   public int getX() { return x; }
   public int getY() { return y; }
}
```

same class	class in same package	subclass in different package	non-subclass, different package	
➡	➡	➡	➡	private
➡	➡	➡	➡	*package*
➡	➡	➡	➡	protected
➡	➡	➡	➡	public

Java Accessibility vs. C++

- No “friend” keyword
- Every field or method has an access specifier (no “public:” sections)
- Default is package-visibility which has no associated keyword (not private)

No Need for Forward Declarations

Definition

Use

```
public class Point {
   private PointColor c;
   // setXY(int,int) used below before its definition in the source
   protected void move(int x, int y) { setXY(this.x+x, this.y+y); }
   void setXY(int x, int y) { this.x = x; this.y = y; }
   private int x, y;
} // no trailing semicolon (C++ requires one)

// PointColor already used above before this definition
class PointColor {
   byte red, green, blue;
}
```

Final Fields

```
public final class Circle {
  private final double MY_PI = 3.1415;
  public double area() { return MY_PI * r*r; }
}
```

- final fields correspond to C++'s "const"
- final fields cannot be changed once initialized
- cannot use final in function signatures
  (less flexible than C++— const is an unused reserved word in Java)

Ball and CBall Example

BallExample/Ball.java

```
package BallExample;
public class Ball implements Bounceable {
    private int x, y;
    public Ball(int x_, int y_) {
        x = x_; y = y_;
    }
    public void Bounce() {
        System.err.println("Ball bounces");
    }
    static public void ClassFn() {
        System.err.println("Ball.ClassFn()");
    }
}
```

BallExample/CBall.java

```
package BallExample;
public class CBall extends Ball {
    private int colorSelector;
    public CBall(int x, int y) {
        super(x,y);         // chain constructors
        colorSelector = 0; // for black
    }
    public void Bounce() {
        System.err.println(”CBall bounces");
    }
    static public void ClassFn() {
        System.err.println(”CBall.ClassFn()");
    }
}
```

Inheritance Mechanisms

- extends *superclass*
  - similar to ": public" in C++
  - for expressing an "is-a" relation
- implements *superinterface*
  - similar in use to C++'s multiple inheritance
  - for expressing an "is-capable-of" or "knows-how-to" relation

Java Interfaces

```
public interface Bounceable {
    public void Bounce();
    private void BounceNow();  // error
}
```

```
public interface BounceDropable
                        extends Bounceable {
    public void Drop();
}
```

- Interfaces can only specify public methods
- Similar to protocols in Smalltalk
- May be used as a type for a variable
- Can specify sub-interfaces
  and can extend multiple interfaces at a time

Bounceable Interface

BallExample/Bounceable.java

```
package BallExample;
public interface Bounceable {
    public void Bounce();
}
```

Errors?

Output?

BallExample/BallTest.java

```
package BallExample;
public class BallTest {
    public static void main(String[] args) {
        Ball b1 = new Ball(10,10);
        Ball b2 = new CBall(20,20);
        Bounceable b3 = new Ball(30,30);
        Bounceable b4 = new CBall(40,40);

        b1.Bounce();        b2.Bounce();
        b3.Bounce();        b4.Bounce();

        b1.ClassFn();       b2.ClassFn();
        b3.ClassFn();       b4.ClassFn();

        CBall cb1 = (CBall) b1;
        CBall cb2 = (CBall) b2;
        cb2.ClassFn();
    }   } // end class
```

Ball Example Output and Errors

```
% java BallExample.BallTest
Ball bounces
CBall bounces
Ball bounces
CBall bounces
Ball.ClassFn()
Ball.ClassFn()
CBall.ClassFn()
```

BallExample/BallTest.java

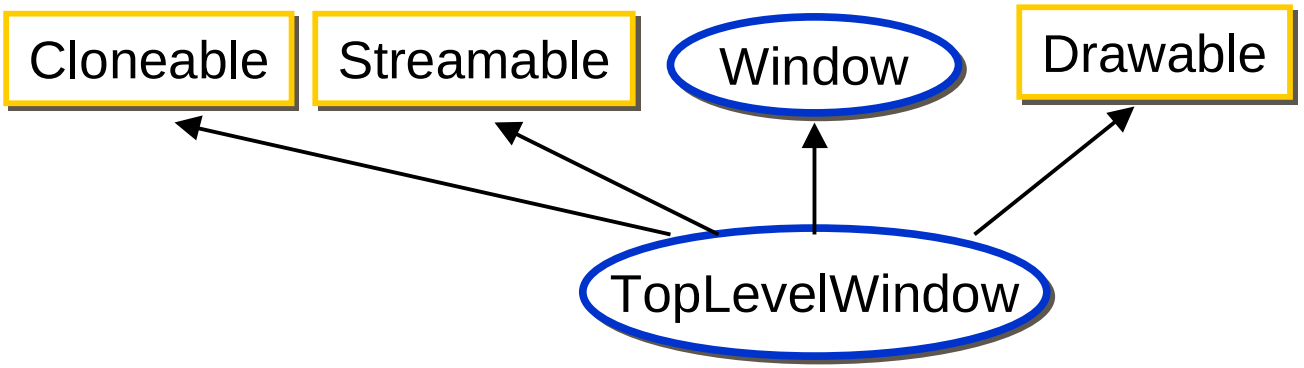
```
package BallExample;
public class BallTest {
    public static void main(String[] args) {
        Ball b1 = new Ball(10,10);
        Ball b2 = new CBall(20,20);
        Bounceable b3 = new Ball(30,30);
        Bounceable b4 = new CBall(40,40);

        b1.Bounce();        b2.Bounce();
// compile-time errors
//      b3.Bounce();        b4.Bounce();

        b1.ClassFn();       b2.ClassFn();
        b3.ClassFn();       b4.ClassFn();

    //  CBall cb1 = (CBall) b1;  ClassCastException
        CBall cb2 = (CBall) b2;  // ok
        cb2.ClassFn();
    }  } // end class
```

Types vs. Classes

- Types are a compile-time notion
  - variables have types
  - used for checking validity of method invocations
  - may be an interface
- Classes are a run-time notion
  - objects (i.e. instances) have classes
  - used for dynamic dispatch
    (binding of non-static function call)
  - Each class has a corresponding type —
    that hierarchy of types mirrors the class hierarchy

Multiple Inheritance in Java

- A Java class can extend (subclass) another class and implement multiple interfaces




```
public class TopLevelWindow extends Window
    implements Drawable, Cloneable, Streamable
    { . . . }
```

Abstract Methods and Abstract Classes

```
// Note abstract keyword is used for the class, too
public abstract class Shape {
  public abstract void rotate(int);     // no definition
  public abstract double area();        // no definition
}
```

- abstract methods correspond to C++'s "pure virtual functions"
  (But C++ uses "=0" syntax, and permits an implementation)
- abstract methods must be overridden in concrete subclasses
- Only abstract classes can have abstract methods
  (C++ infers abstract classes, Java requires you mark the class explicitly)

Final Methods

```
public class Circle {
  ....
  public final double area() { return Math.PI * r*r; }
  double r;           // radius
}
```

- final methods cannot be overriden
- final methods may be inlined (no "inline" keyword)
- similar to non-virtual member functions in C++
  (but those can be overriden, they just do not dispatch dynamically)

Final Classes

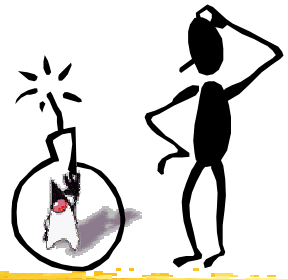
```
public final class Circle {
  ....
  public double area() { return Math.PI * r*r; }
  double r;           // radius
}
```

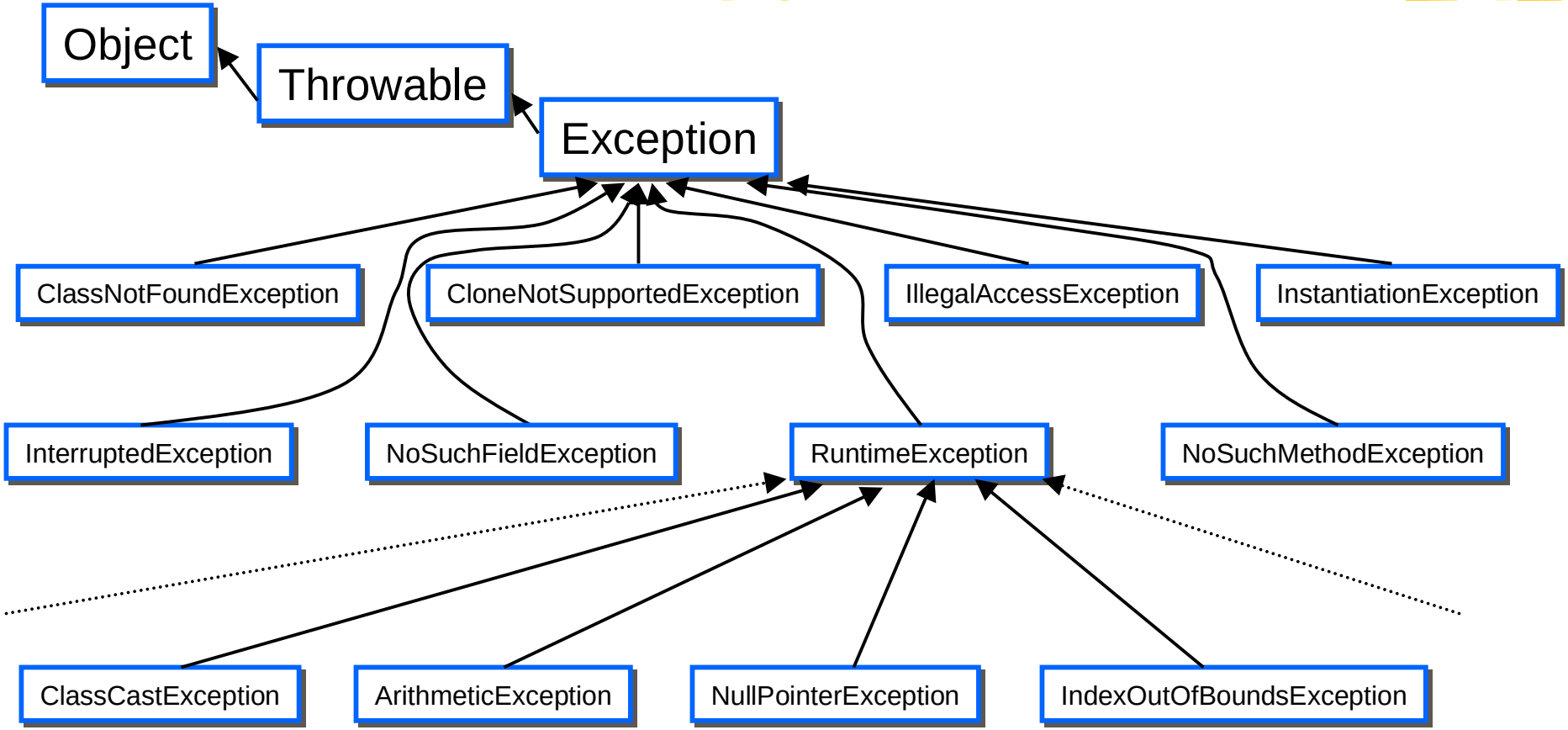
- final classes cannot be subclassed —
  they are leafs in the class hierarchy
- methods in final classes are implicitly final
- provides compiler with optimization opportunities

try { throw } and catch, finally (exceptions)

```
class ExceptionExample {
   static public void main(String args[]) {
      try {
         // allocate some resource (besides memory)
         doSomething();
         if (!FThingsAreOkay()) {
            throw new RuntimeException("Things not ok");
         }
         doSomethingElse();
      } catch (RuntimeException e) {
         System.err.println("Runtime Exception: " + e);
      } catch (Exception e) {              // similar to "catch (..)" in C++
         System.err.println("Exception: " + e);
      } finally {                          // finally is not in C++
         // cleanup resource
      }  }
```

Exception Hierarchy





Threads

```
public class Pendulum extends Applet implements Runnable {
   private Thread myThread;
   public void start() {
      if (myThread == null) {
         myThread = new Thread(this, "Pendulum");
         myThread.start();
      }
   }
   public void run() {
      while (myThread != null) {
         try { myThread.sleep(100); }
         catch (InterruptedException e) { /* do nothing */ }
         myRepaint();
      }
    }
   public void stop() { myThread.stop(); myThread = null; }
}
```

set thread's target to this Pendulum class, and use its run() method

Ref: Boone's *Java Essentials for C and C++ Programmers*

Summary: What Java Left Out from C++

- No stack objects, only heap objects
- No destructors, only finalize() method
- No pointers, everything is a reference
- No delete, garbage collector instead
- No const, only final (methods, fields, classes)
- No templates, no preprocessor
- No operator overloading
- No multiple inheritance of classes
- No enumerations or typedefs

Summary: What Java Put In (vs. C++)

- Garbage collector
- Object-rooted, rich class hierarchy
  Strings, first-class arrays with bounds checking
- Package system with import
- interface, implements, extends, abstract
- finally blocks, static/instance initializers
- Secure and portable JavaVM, threads
- Dynamic reflection capabilities, inner classes
- JavaDoc system