

Java

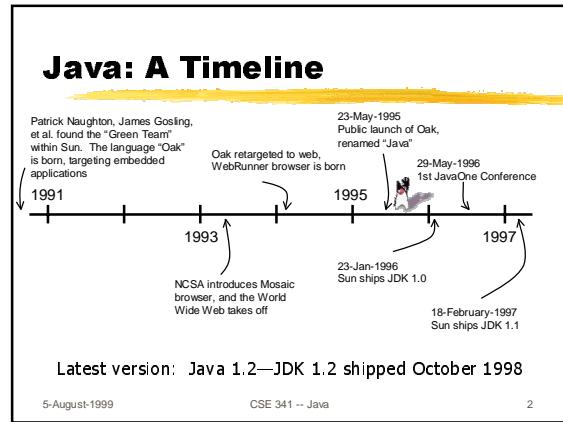


"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language."

— Sun

Greg J. Badros
badros@cs.washington.edu
University of Washington, Seattle
CSE-341, Summer 1999
(C) 1999, Greg J. Badros—All Rights Reserved

Java: A Timeline



Patrick Naughton, James Gosling, et al. found the "Green Team" within Sun. The language "Oak" is born, targeting embedded applications

1991

Oak retargeted to web, WebRunner browser is born

1993

NCSA introduces Mosaic browser, and the World Wide Web takes off

1995

23-May-1995 Public launch of Oak, renamed "Java"
29-May-1996 1st JavaOne Conference

1996

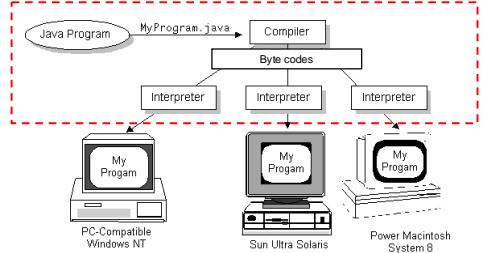
23-Jan-1996 Sun ships JDK 1.0
18-February-1997 Sun ships JDK 1.1

1997

Latest version: Java 1.2—JDK 1.2 shipped October 1998

5-August-1999 CSE 341 -- Java 2

Java: The Language



A Java Program (MyProgram.java) is compiled by a Compiler into Byte codes. These byte codes are then interpreted by three different interpreters running on three different platforms: PC-Compatible Windows NT, Sun Ultra Solaris, and Power Macintosh System 8. All three platforms produce the same output: "My Program".

5-August-1999 CSE 341 -- Java Ref: Sun- What is Java? page 3

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!

5-August-1999 CSE 341 -- Java 4

Java vs. C++



Java

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

C++

```
// Application HelloWorld
#include <iostream.h>
int main(int argc, char* argv[]) {
    cout << "Hello World!" << endl;
}
```

5-August-1999 CSE 341 -- Java 5

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** is not as necessary

5-August-1999 CSE 341 -- Java 6

Brewing Java

The diagram illustrates the influences on Java from four programming languages:

- C**: basic operators, primitive types, control structures.
- C++**: object syntax, object types, exceptions.
- Smalltalk**: object model, dynamic features, rich class hierarchy.
- Modula**: package system.

Arrows point from each language to a central Java logo, representing how their features were incorporated into Java's design.

5-August-1999 CSE 341 -- Java 7

Java vs. Smalltalk

The diagram compares a Java code snippet with its equivalent Smalltalk code.

Java

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

Smalltalk

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

5-August-1999 CSE 341 -- Java 8

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

5-August-1999 CSE 341 -- Java 9

Java vs. C++, Revisited

The diagram compares Java code with C++ code using both pointer-style and reference-style syntax.

Java

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

C++ (pointer-style)

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

C++ (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();
```

5-August-1999 CSE 341 -- Java 10

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

5-August-1999 CSE 341 -- Java 11

Java's Class Hierarchy

The diagram shows the Java class hierarchy starting from the `Object` class.

- `Object` is the root class.
- `String`, `Boolean`, `Number`, `Compiler`, `Component`, and `arrays` are final classes.
- `Container` is an abstract class.
- `Panel` and `Applet` are concrete classes.
- `Byte` and `Long` are wrapped primitives.
- `Component` has subclasses `Container`, `Panel`, and `Applet`.
- `Container` has a subclass `Panel`.
- `Panel` has a subclass `Applet`.

5-August-1999 CSE 341 -- Java 12

Java Documentation

Class java.lang.Boolean

```

  public final class Boolean
    extends Object
    implements Serializable
}

The Boolean class wraps a value of the primitive type boolean in an object. An object of type Boolean contains a single field whose type is boolean.

In addition, this class provides many methods for converting a boolean to a string and a string to a boolean, as well as other constants and methods useful when dealing with a boolean.
  
```

Field Index

- **FALSE**: The Boolean object corresponding to the primitive value false.
- **TRUE**: The Boolean object corresponding to the primitive value true.
- **TYPE**: The Class object representing the primitive type boolean.

Constructor Index

- **Boolean()**: Creates a Boolean object representing the value argument.
- **Boolean(String s)**: Creates a Boolean object representing the value true if the string argument is not null and is equal (ignoring case, to the string "true").

Method Index

5-August-1999 13 Ref: Java In a Nutshell, O'Reilly JAVA

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(8, 334, 104); g.drawOval(7, 336, 106);
        g.setColor(Color.black); g.setFont(font);
        g.drawString(message, 40, 75);
    }
}
  
```

5-August-1999 CSE 341 -- Java Ref: Java In a Nutshell, O'Reilly 14 JAVA

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

Applet Viewer: HelloWorldApplet.class

Report started.

Hello World

5-August-1999 CSE 341 -- Java Ref: Java In a Nutshell, O'Reilly 15 JAVA

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

5-August-1999 CSE 341 -- Java 16 JAVA

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

5-August-1999 CSE 341 -- Java 17 JAVA

Default Arguments

► No language support—must use overloading instead

special use of "this"

```

public class Point {
    public Point() { this(0,0); }
    public Point(int x, int y) { this.x = x; this.y = y; }
    public void move() { move(1); }
    public void move(int dx) { x += dx; }
    private int x, y;
}
  
```

5-August-1999 CSE 341 -- Java 18 JAVA

“Override” vs. “Overload”



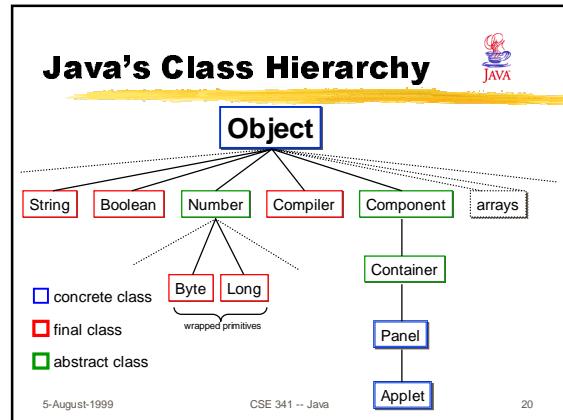
Override

- 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

5-August-1999 CSE 341 -- Java 19

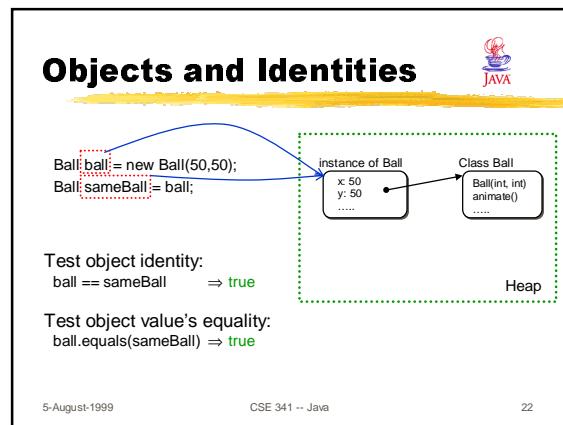


What can an Object do for you today?



- `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`

5-August-1999 CSE 341 -- Java 21

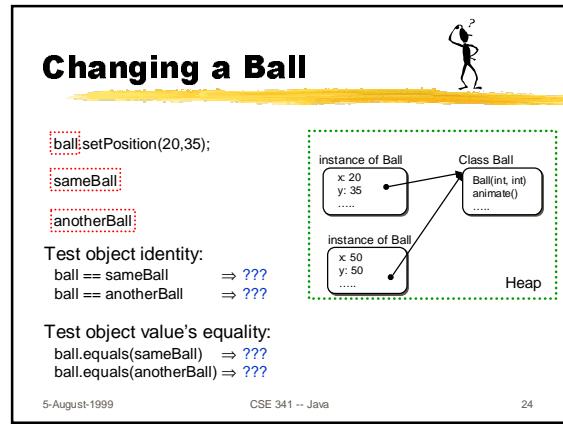


Cloning Objects

Test object identity:
`ball == anotherBall` \Rightarrow false

Test object value's equality:
`ball.equals(anotherBall)` \Rightarrow true

5-August-1999 CSE 341 -- Java 23



Inequality in Balls!

The diagram illustrates the difference between object identity and value equality in Java. It shows two instances of the `Ball` class on the heap, each with its own unique memory address. The first instance has coordinates `x: 20, y: 35`, and the second has `x: 50, y: 50`. Both instances have the same methods: `Ball(int, int)` and `animate()`.

Test object identity:

- `ball == sameBall` ⇒ true
- `ball == anotherBall` ⇒ false

Test object value's equality:

- `ball.equals(sameBall)` ⇒ true
- `ball.equals(anotherBall)` ⇒ false

5-August-1999 CSE 341 -- Java 25

Assignment just changes the pointer

This diagram shows how assignment in Java changes the reference stored in a variable. It starts with three variables: `ball`, `sameBall`, and `anotherBall`, all pointing to the same heap location for the first `Ball` object (x: 20, y: 35). When `sameBall = anotherBall;` is executed, the `sameBall` pointer is updated to point to the second `Ball` object (x: 50, y: 50) on the heap.

Test object identity:

- `ball == sameBall` ⇒ false
- `ball == anotherBall` ⇒ false

Test object value's equality:

- `ball.equals(sameBall)` ⇒ false
- `ball.equals(anotherBall)` ⇒ false

5-August-1999 CSE 341 -- Java 26

Java variables hold...

The slide covers three types of Java variables:

- 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;`. A note states: "String literals actually invoke constructor e.g., new String("World")"
- Array reference**: `int[] intArray = { 1, 2, 3, 4, 5, };`, `String[] strArray = { "Hello", "World", };`. A note states: "String literals actually invoke constructor e.g., new String("World")"

5-August-1999 CSE 341 -- Java 27

Arrays

Similar to:

- Java arrays are 1st-class Objects**
- 0-indexed**
- Bounds checking is performed**
- Store/Retrieve using [] operator**: `strArray[0] = strArray[1];`
- Have implicit length field**: `strArray.length` ⇒ 2. A note states: "A field, not a method!"

5-August-1999 CSE 341 -- Java 28

Identifiers

The slide shows the structure of Java identifiers:

- Everything has a globally-unique name**: `Java.lang.String`, `Java.util.Hashtable`, `Java.applet.Applet`, `EDU.Washington.grad.gjb.cassowary.Variable.toString()`. Brackets group the package name (`Java`), class name (`String`), and method name (`toString`).
- Pretty wordy, so...**

5-August-1999 CSE 341 -- Java 29

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"

5-August-1999 CSE 341 -- Java 30

How Java Finds a Class...

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

```
./src/Washington/grad/gjb/cassowary/Variable.java
package EDU.Washington.grad.gjb.cassowary;

public class Variable extends AbstractVariable {
    ...
}

class Helper { ... }
```

5-August-1999

CSE 341 -- Java

31

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

5-August-1999

CSE 341 -- Java

32

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

5-August-1999

CSE 341 -- Java

33

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	trans-subclass in different package	private
Y	N	N	N	private
Y	Y	N	N	package
Y	Y	Y	N	protected
Y	Y	Y	Y	public

5-August-1999

CSE 341 -- Java

Ref. Java In a Nutshell O'Reilly 34

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)

5-August-1999

CSE 341 -- Java

35

No Need for Forward Declarations

```
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;
}
```

■ Definition
■ Use

5-August-1999

CSE 341 -- Java

36

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
- **final** on formal function parameters is not part of the function signature (just implementation detail)

5-August-1999

CSE 341 -- Java

37

Ball and CBall Example

BallExample/Ball.java

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

5-August-1999

CSE 341 -- Java

38

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

5-August-1999

CSE 341 -- Java

39

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

5-August-1999

CSE 341 -- Java

40

Bounceable Interface



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

```
BallExample/BallTest.java
package BallExample;
public class BallTest {
    public static void main(String[] args) {
        Ball b1 = new Ball(10,10);
        Ball b2 = new Ball(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
}
```



5-August-1999

CSE 341 -- Java

41

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();
        b3.Bounce();      b4.Bounce();
        b1.ClassFn();    b2.ClassFn();
        b3.ClassFn();    b4.ClassFn();
        // compile time errors
        // b3.ClassFn();   b4.ClassFn();
        CBall cb1 = (CBall) b1; // ClassCastException
        CBall cb2 = (CBall) b2; // ok
        cb2.ClassFn();
    } // end class
}
```

5-August-1999

CSE 341 -- Java

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

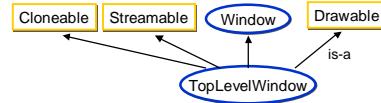
5-August-1999

CSE 341 -- Java

43

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
    { . . . }
  
```

5-August-1999

CSE 341 -- Java

44

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)

5-August-1999

CSE 341 -- Java

45

Final Methods



```

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

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

5-August-1999

CSE 341 -- Java

46

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

5-August-1999

CSE 341 -- Java

47

try { throw } and catch, finally (exceptions)



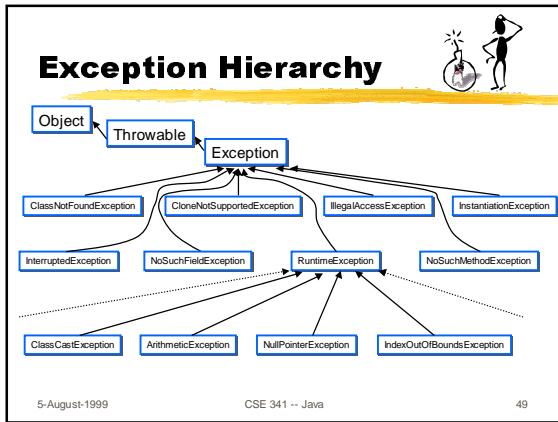
```

class ExceptionExample {
    static public void main(String args[]) {
        try {
            // allocate some resource (besides memory)
            doSomething();
            if (!FTFsAreOkay())
                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 {
            // cleanup resource
        }
    }
}
  
```

5-August-1999

CSE 341 -- Java

48



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

50

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

5-August-1999

CSE 341 -- Java

51

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

5-August-1999

CSE 341 -- Java

52