

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<https://algs4.cs.princeton.edu>

ADVANCED JAVA

- ▶ *inheritance*
- ▶ *interfaces*
- ▶ *iterators*



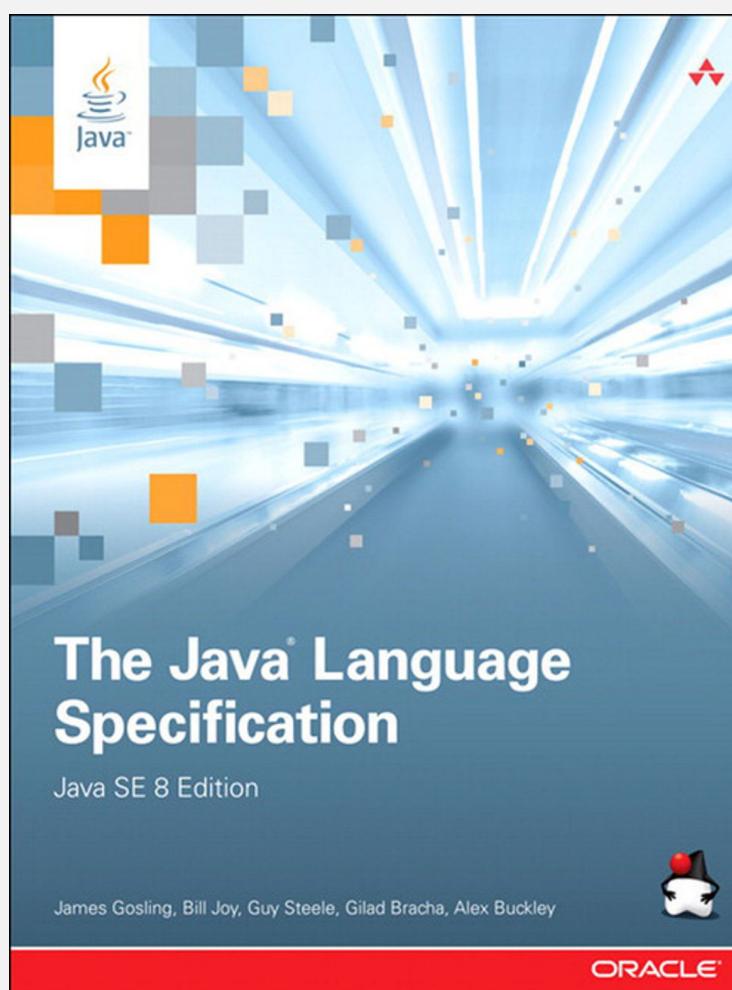
Subtitle. Java features that we (occasionally) use in this course, but don't cover (much) in COS 126.

- Inheritance.
- Generics.
- Interfaces.
- Iterators.

← common theme: promote code reuse

Q. How to take your Java to the next level?

A.



<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>



ADVANCED JAVA

► *inheritance*

► *interfaces*

► *iterators*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<https://algs4.cs.princeton.edu>

Motivation

Q1. How did the Java architects design `System.out.println(x)` so that it works with all reference types?



Q2. How would an Android developer create a custom Java GUI text component, without re-implementing these 400+ required methods?



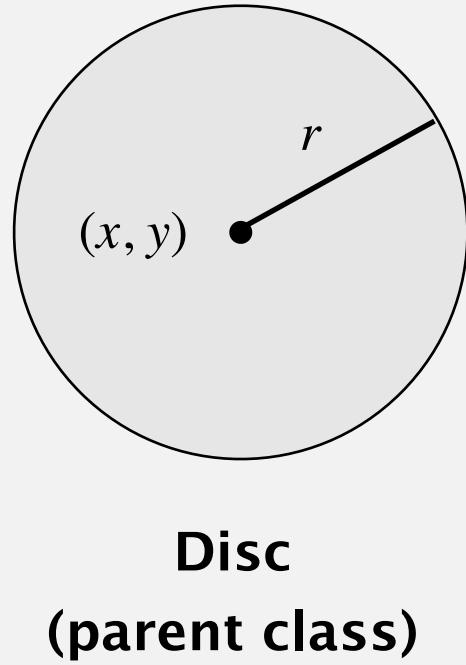
A. Inheritance.

```
action() • add() • addAncestorListener() • addCaretListener() •  
addComponentListener() • addContainerListener() • addFocusListener() •  
addHierarchyBoundsListener() • addHierarchyListener() • addImpl() •  
addInputMethodListener() • addKeyListener() • addKeymap() • addMouseListener() •  
addMouseMotionListener() • addMouseWheelListener() • addNotify() •  
addPropertyChangeListener() • addVetoableChangeListener() •  
applyComponentOrientation() • areFocusTraversalKeysSet() • bounds() • checkImage() •  
coalesceEvents() • computeVisibleRect() • contains() • copy() • countComponents() •  
createImage() • createToolTip() • createVolatileImage() • cut() • deliverEvent() •  
disable() • disableEvents() • dispatchEvent() • doLayout() • enable() •  
enableEvents() • enableInputMethods() • findComponentAt() • fireCaretUpdate() •  
firePropertyChange() • fireVetoableChange() • getActionForKeyStroke() •  
getActionMap() • getAlignmentX() • getAlignmentY() • getAncestorListeners() •  
getAutoscrolls() • getBackground() • getBaseline() • getBaselineResizeBehavior() •
```

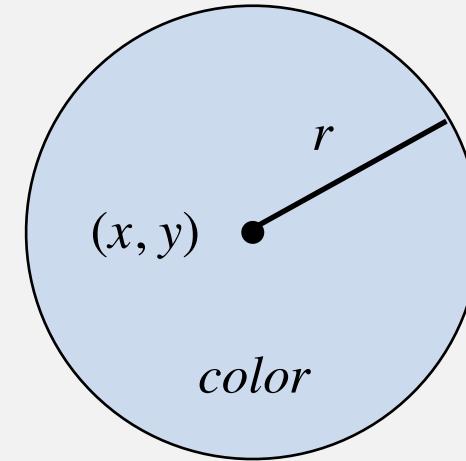
Inheritance overview

Implementation inheritance (subclassing).

- Derive a new class (**child class**) from an existing class (**parent class**).
- The child class **inherits** properties from its parent class:
 - state (instance variables)
 - behavior (instance methods)
- The child class can **override** instance methods in the parent class.
(replacing those methods with its own versions)



Disc
(parent class)



ColoredDisc
(child class)

Main benefits.

- Facilitates code reuse.
- Enables the design of extensible libraries.

Inheritance example

```
public class Disc {  
    private final int x, y, r;  
  
    public Disc(int x, int y, int r) {  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    public double area() {  
        return Math.PI * r * r;  
    }  
  
    public boolean intersects(Disc that) {  
        int dx = this.x - that.x;  
        int dy = this.y - that.y;  
        int dr = this.r + that.r;  
        return dx*dx + dy*dy <= dr*dr;  
    }  
  
    public void draw() {  
        StdDraw.filledCircle(x, y, r);  
    }  
}
```

parent class

```
import java.awt.Color;  
  
public class ColoredDisc extends Disc {  
  
    private final Color color; ← defines new state  
  
    public ColoredDisc(int x, int y, int r, Color color) {  
        super(x, y, r); ← calls constructor in parent class  
        this.color = color;  
    }  
  
    public Color getColor() { ← defines new behavior  
        return color;  
    }  
  
    public void draw() {  
        StdDraw.setPenColor(color);  
        super.draw(); ← calls method in parent class  
    }  
}
```

child class

Inheritance demo (in JShell)



```
~/Desktop/advanced-java> jshell-algs4
/open Disc.java
/open ColoredDisc.java

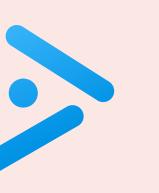
StdDraw.setScale(0, 800);

Disc disc1 = new Disc(400, 400, 200);
disc1.area();
disc1.draw();

ColoredDisc disc2 = new ColoredDisc(225, 575, 100, StdDraw.BLUE);
ColoredDisc disc3 = new ColoredDisc(575, 575, 100, StdDraw.RED);
disc2.getColor();
disc2.draw();
disc3.draw();
disc2.area();

disc1.intersects(disc2);
disc2.intersects(disc3);

Disc disc = disc2;      // child also inherits type from parent
disc.area();
```



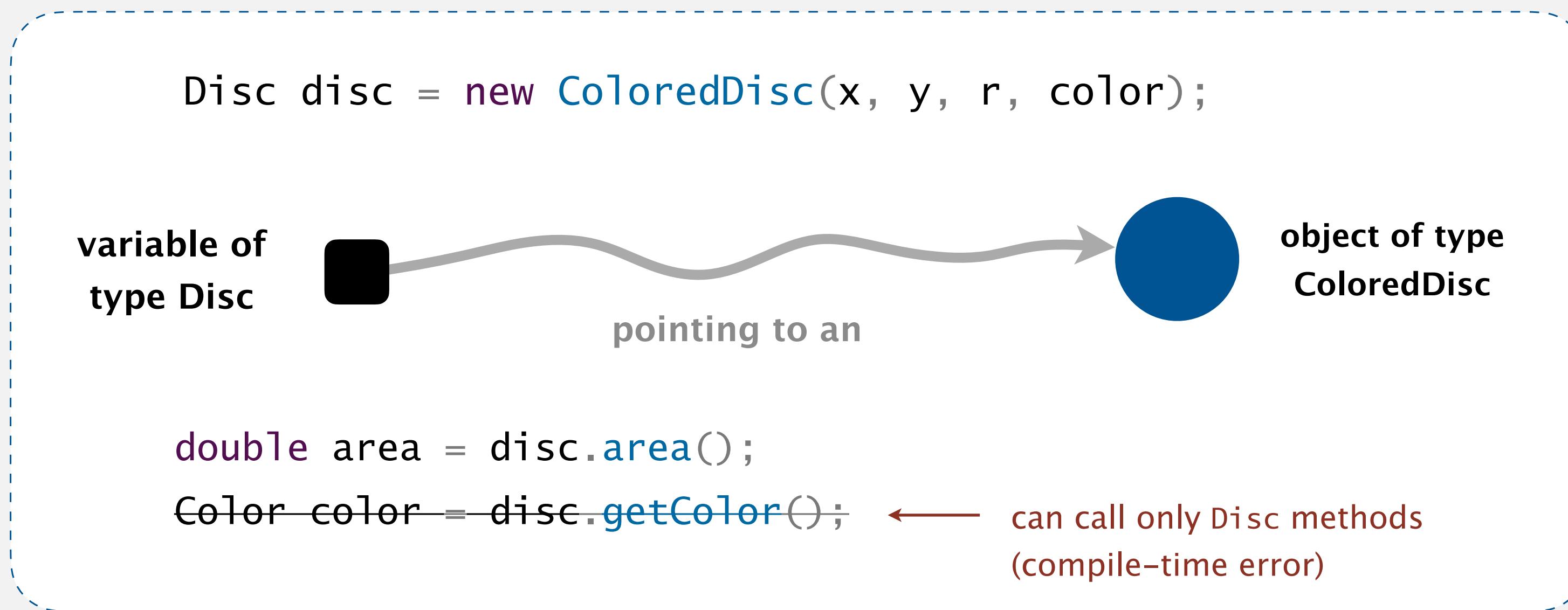
Which color will be stored in the variable `color`?

```
Disc disc = new ColoredDisc(200, 300, 100, StdDraw.BLUE);  
Color color = disc.getColor();
```

- A. Blue.
- B. Black.
- C. Compile-time error.
- D. Run-time error.
- E. 

Polymorphism

Ex. A reference variable can refer to any object of its declared type or any of its subclasses.

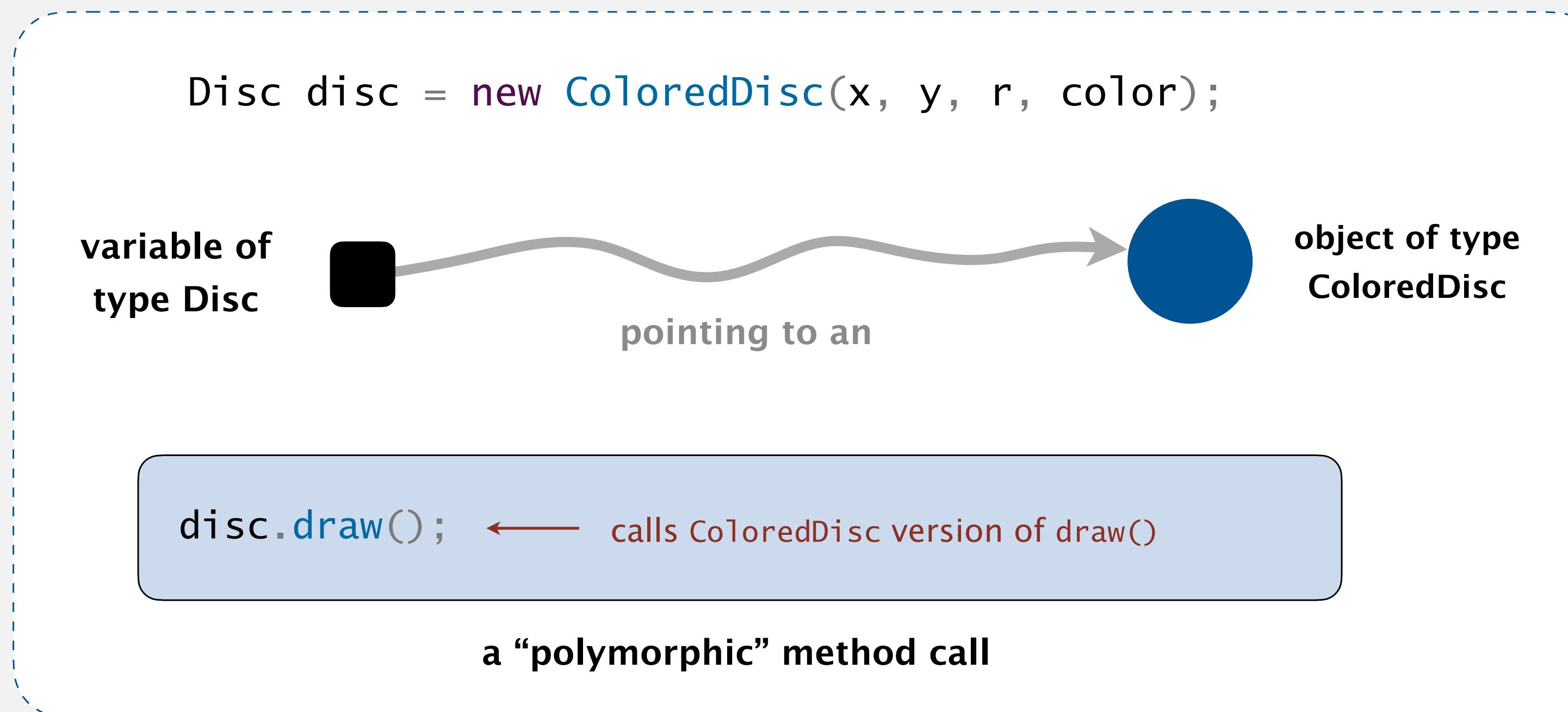


Subtype polymorphism. A subclass is a **subtype** of its superclass:
objects of the subtype can be used anywhere objects of the superclass are allowed.

RHS of assignment statement,
method argument, return value, expression, ...

Poly**morphism**

Dynamic dispatch. Java determines which version of an **overridden method** to call using the type of the referenced object at runtime (not necessarily the type of the variable).



Poly**morphism**: why useful?

Ex 1. Design a method that can take either `Disc` or `ColoredDisc` objects as arguments.

```
Disc disc1 = new Disc(x1, y1, r1);
ColoredDisc disc2 = new ColoredDisc(x2, y2, r2, color);
boolean result1 = disc1.intersects(disc2);
```



passes an object of type `ColoredDisc`
to a method that takes as an object of type `Disc`

Ex 2. Store a mixed collection of `Disc` and `ColoredDisc` objects.

```
Disc disc1 = new Disc(x1, y1, r1);
Disc disc2 = new Disc(x2, y2, r2);
ColoredDisc disc3 = new ColoredDisc(x3, y3, r3, color);
Disc[] discs = { disc1, disc2, disc3 };

for (int i = 0; i < discs.length; i++)
    discs[i].draw();
```



assigns an object of type `ColoredDisc`
to a variable of type `Disc`

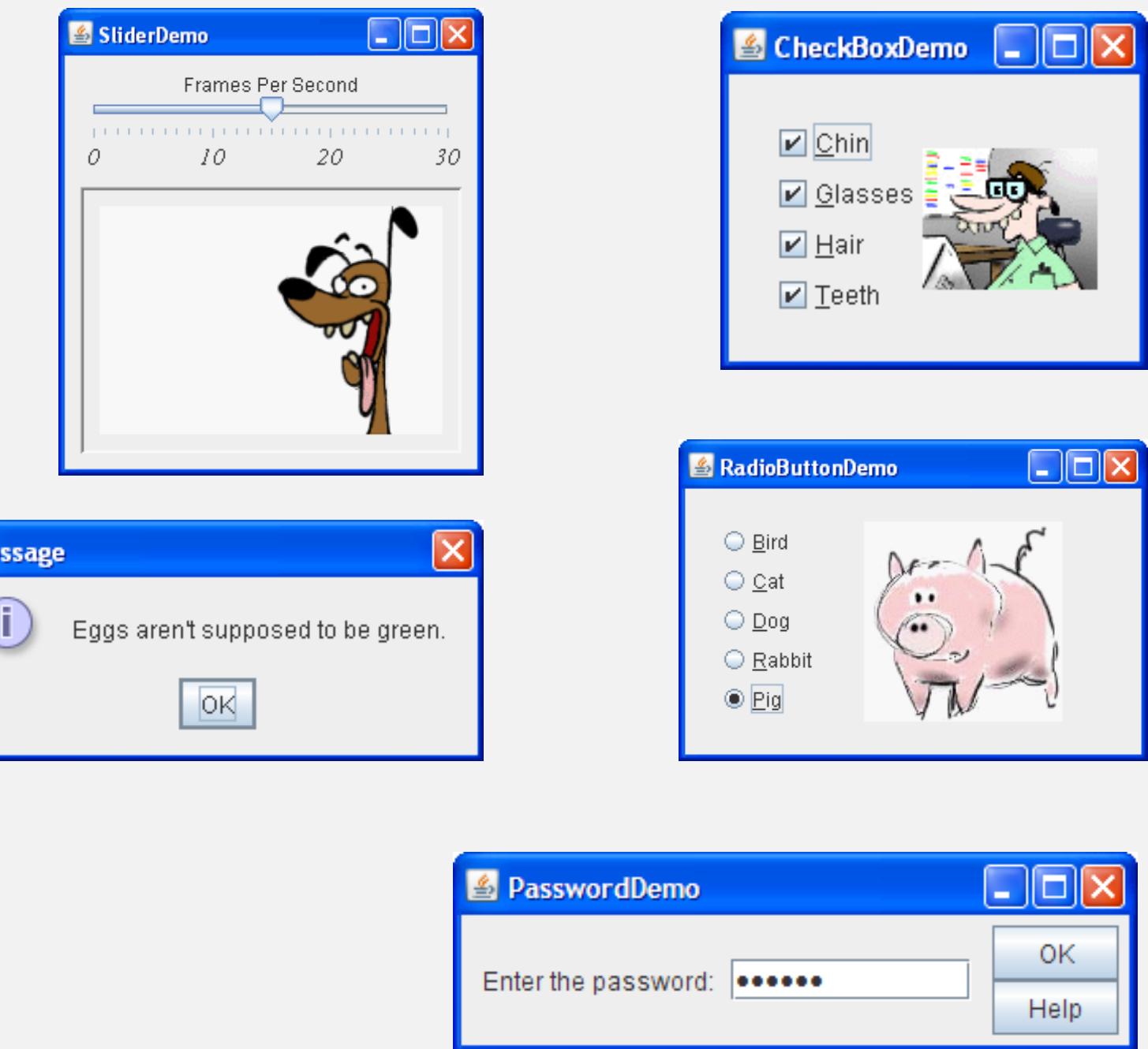
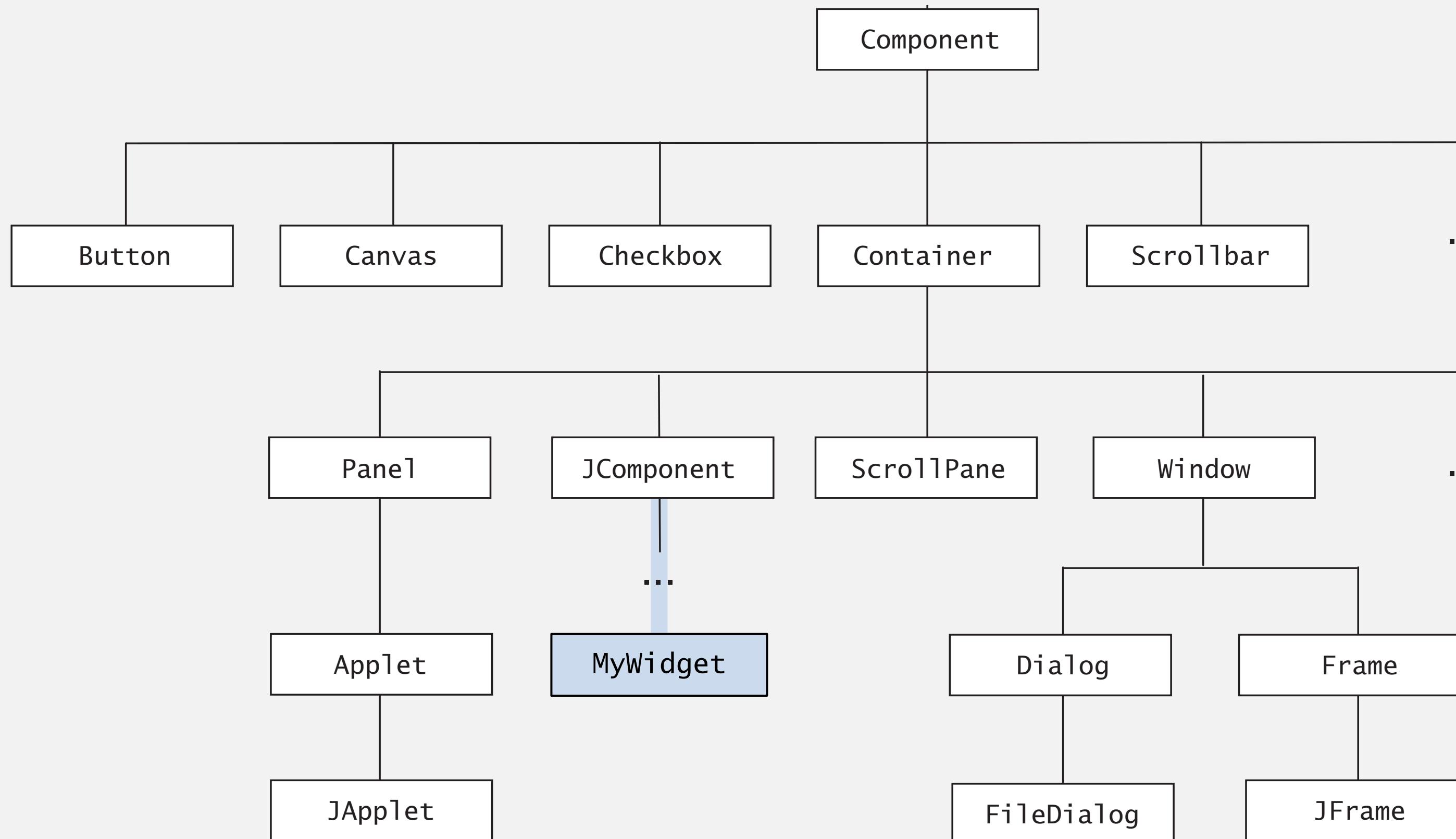


manipulate objects in a uniform manner

Inheritance hierarchy for Java GUI components

Typical use case. Design an extensible library.

Ex. Android developer design a new GUI widget for their app;
new widget can be used anywhere a `JComponent` is expected.



Java GUI class hierarchy

Is-A relationship

Informal rule. Inheritance should represent an **Is-A** relationship.

child class	parent class
ColoredDisc	Disc
ArithmeticException	RuntimeException
JPasswordField	JTextField
SamsungGalaxyS24	SmartPhone

“ Objects of subtypes should behave like those of supertypes if used via supertype methods.” — Barbara Liskov



Liskov substitution principle

Inheritance oops

[java.util.Stack](#). Inherits from [java.util.Vector](#).

[Java 1.3 bug report \(June 27, 2001\)](#)

The iterator method on `java.util.Stack` iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.



[status: closed, won't fix \(June 11, 2004\)](#)

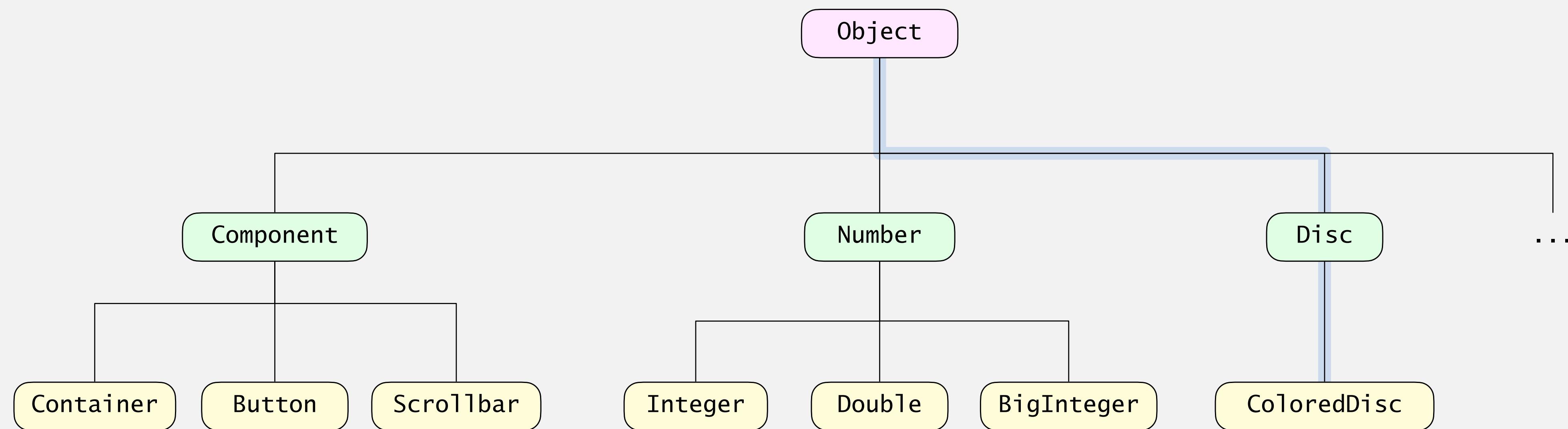
It was an incorrect design decision to have Stack extend Vector ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

Java's Object superclass

Object data type. Every other class has `Object` as a (direct or indirect) superclass.

```
public class Disc extends Object {  
    ...  
}
```

↑
added implicitly
(if no `extends` clause)



Java class hierarchy

Java's Object superclass

Object data type. Every other class has `Object` as a (direct or indirect) superclass.

<code>public class Object</code>	
<code>boolean equals(Object x)</code>	<i>is this object equal to x ?</i>
<code>int hashCode()</code>	<i>hash code of this object</i>
<code>String toString()</code>	<i>string representation</i>
<code>...</code>	<i>copying, garbage collection, concurrency</i>

Inherited methods. Behavior inherited from `Object` is rarely what you want ⇒ **override** them.

- Equals: reference equality (same as `==`).
- Hash code: arbitrary integer associated with object. ←— think of as a memory address
- String representation: name of class, followed by @, followed by hash code.

The `toString()` method

Best practice. Override the `toString()` method.

```
public class Disc {  
    private final int x, y, r;  
    ...  
  
    public String toString() {  
        return String.format("(%.d, %.d, %.d)", x, y, r);  
    }  
}  
  
works like printf() but returns string  
(instead of printing it)
```

without overriding `toString()` method

```
~/Desktop/inheritance> jshell-a1gs4  
/open Disc.java  
Disc disc = new Disc(100, 100, 20);  
StdOut.println("disc = " + disc.toString());  
disc = Disc@239963d8
```

after overriding the `toString()` method

```
disc = (100, 100, 20)
```

String concatenation operator. Java calls an object's `toString()` method implicitly.

```
StdOut.println("disc = " + disc);
```

string concatenation operator

Inheritance summary

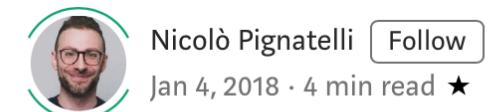
Subclassing. Powerful OOP mechanism for **code reuse**.

Caveats.

- Stuck with inherited instance variables and methods forever.
- Subclass may break with seemingly innocuous change to superclass.

Inheritance Is Evil. Stop Using It.

"Use inheritance to extend the behavior of your classes". This concept is one of the most widespread, yet wrong and dangerous in OOP. Do yourself a favor and stop using it right now.



Nicolò Pignatelli [Follow](#)
Jan 4, 2018 · 4 min read ★



Best practices.

- Use with extreme care.
- Favor composition (or interfaces) over subclassing.

This course.

- Yes: override inherited methods: `toString()`, `hashCode()`, and `equals()`.
- No: define inheritance hierarchies.



ADVANCED JAVA

- ▶ *inheritance*
- ▶ **interfaces**
- ▶ *iterators*

<https://algs4.cs.princeton.edu>

Motivation

- Q1. How to design a single method that can sort arrays of strings, integers, or dates?
- Q2. How to iterate over a collection without knowing the underlying representation?
- Q3. How to intercept and process mouse clicks in a Java app?

A. Java interfaces.

```
String[] a = {  
    "Apple", "Orange", "Banana"  
};  
Arrays.sort(a);  
  
Integer[] b = { 3, 1, 2 };  
Arrays.sort(b);
```

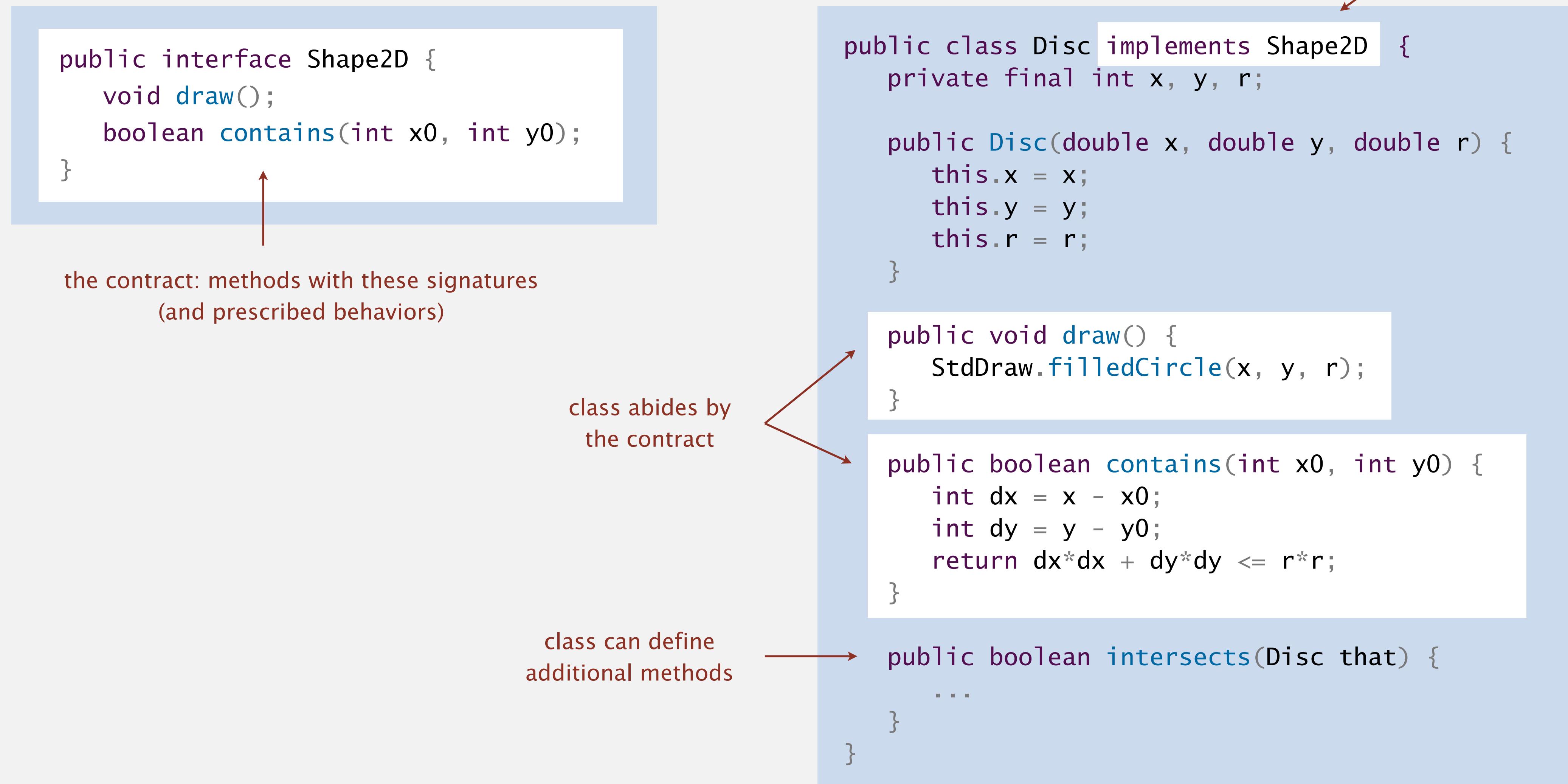
sort arrays

```
Stack<String> stack = new Stack<>();  
stack.push("First");  
stack.push("Whitman");  
stack.push("Mathey");  
  
for (String s : stack)  
    StdOut.println(s);
```

iterate over items in a collection

Java interfaces overview

Interface. A set of related methods that define some **behavior** (partial API) for a class.



Java interfaces overview

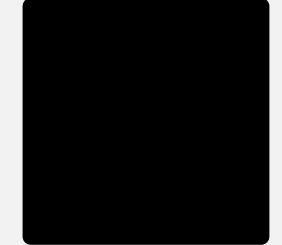
Interface. A set of related methods that define some **behavior** (partial API) for a class.

```
public interface Shape2D {  
    void draw();  
    boolean contains(int x0, int y0);  
}
```

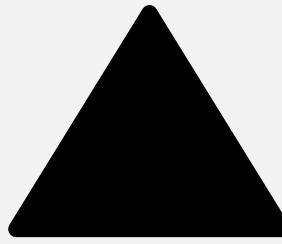
the contract: methods with these signatures
(and prescribed behaviors)

Many classes can implement the same interface.

```
public class Square implements Shape2D {  
    ...  
}
```



```
public class Triangle implements Shape2D {  
    ...  
}
```



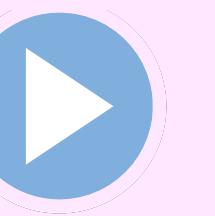
```
public class Star implements Shape2D {  
    ...  
}
```



```
public class Heart implements Shape2D {  
    ...  
}
```



Java interfaces demo (in JShell)



```
~/Desktop/inheritance> jshell-algs4
/open Shape2D.java
/open Disc.java
/open Square.java
/open Heart.java

Square square = new Square(400, 400, 200);
Shape2D disc = new Disc(400, 700, 100);
Shape2D heart = new Heart(400, 400, 100);           | ← implicit type conversion
                                                       (upcasting)

Shape2D s = "Hello, World";                         // compile-time error (incompatible types)

square.area();
square.contains(400, 300);
disc.contains(400, 300);
disc.area();                                         // compile-time error (not a Shape2D method)

StdDraw.setScale(0, 800);
Shape2D[] shapes = { disc, square, heart };
for (int i = 0; i < shapes.length; i++)
    shapes[i].draw();
```

Java interface properties

Interfaces are reference types. Can use interface name anywhere you can use a data type name.

variable declarations, argument types, return types

Subtype polymorphism. A class that implements an interface is a subtype of that interface:
objects of the subtype can be used anywhere objects of the interface are allowed.

RHS of assignment statements, method arguments, return types, ...

Key differences with inheritance.

- Uses keyword `implements` instead of `extends`.
- No instance variables or instance methods inherited.
- **Multiple inheritance:** a class can implement many interfaces (but extend only one class).

```
public class MovableDisc extends Disc implements Shape2D, Movable {  
    ...  
}
```



Which of the following statements leads to a compile-time error?

- A. `Shape2D shape = new Shape2D();`
- B. `Shape2D[] shapes = new Shape2D[10];`
- C. Both A and B.
- D. Neither A nor B.

Java interfaces in the wild

Interfaces are essential for industrial-strength programming in Java.

purpose	built-in interfaces
sorting	java.lang.Comparable
	java.util.Comparator
iteration	java.lang.Iterable
	java.util.Iterator
collections	java.util.List
	java.util.Map
	java.util.Set
GUI events	java.awt.event.MouseListener
	java.awt.event.KeyListener
	java.awt.event.MenuListener
lambda expressions	java.util.function.Consumer
	java.util.function.Supplier
	java.util.function.BinaryOperator
concurrency	java.lang.Runnable
	java.lang.Callable

← this course

Java interfaces summary

Java interface. A set of methods that define some behavior (partial API) for a class.

Design benefits.

- Enables **callbacks**, which promotes code reuse.
- Facilitates **lambda expressions** for functional programming.

This course.

- Yes: use interfaces built into Java (for sorting and iteration).
- No: define our own interfaces; lambda expressions.

ADVANCED JAVA

► *inheritance*

► *interfaces*

► *iterators*

Algorithms

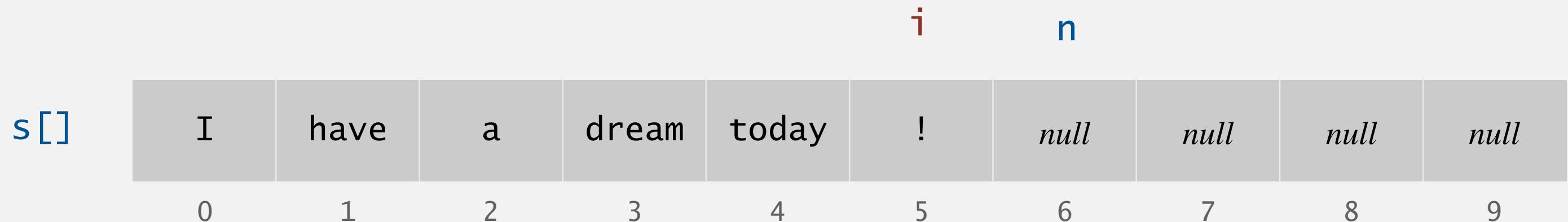
ROBERT SEDGEWICK | KEVIN WAYNE

<https://algs4.cs.princeton.edu>

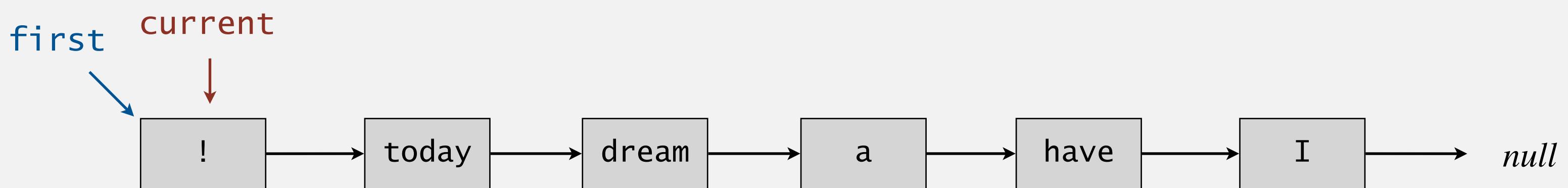
Iteration

Design challenge. Allow client to iterate over the items in a collection, without exposing the collection's internal representation.

stack (resizing-array representation)



stack (linked-list representation)



Java solution. Use a `foreach` loop.

Foreach loop

Java provides elegant syntax for iterating over the items in a collection.

“foreach” loop (shorthand)

```
Stack<String> stack = new Stack<>();  
...  
  
for (String s : stack) {  
    ...  
}
```

equivalent code (longhand)

```
Stack<String> stack = new Stack<>();  
...  
  
Iterator<String> iterator = stack.iterator();  
while (iterator.hasNext()) {  
    String s = iterator.next();  
    ...  
}
```

To provide clients the ability to iterate with a foreach loop:

- Collection must have a method `iterator()`, which returns an `Iterator` object.
- An `Iterator` object represents the state of a traversal.
 - the `hasNext()` returns `false` if the traversal is done
 - the `next()` method returns the next item in the traversal

Iterator and Iterable interfaces

Java defines two interfaces that facilitate foreach loops.

- **Iterable interface:** `iterator()` method that returns an **Iterator**. ←———— “I am a collection that can be traversed with a foreach loop”
- **Iterator interface:** `next()` and `hasNext()` methods. ←———— “I represent the state of one traversal”
- Each interface is parameterized using generics.

java.lang.Iterable interface

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

java.util.Iterator interface

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
}
```

Type safety. Foreach loop won't compile unless collection is **Iterable** (or an array).

Stack iterator: array implementation

```
import java.util.Iterator;

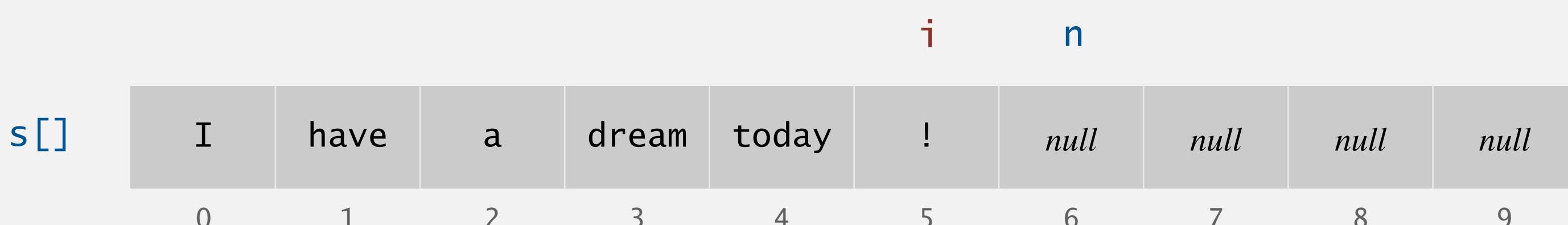
public class ResizingArrayStack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = n-1; // index of next item to return

        public boolean hasNext() { return i >= 0; }
        public Item next() { return s[i--]; }
    }
}
```

Note: next() must throw a NoSuchElementException if called after traversal is done



Stack iterator: linked-list implementation (in IntelliJ)



```
import java.util.Iterator;

public class LinkedStack<Item> implements Iterable<Item>
{
    ...

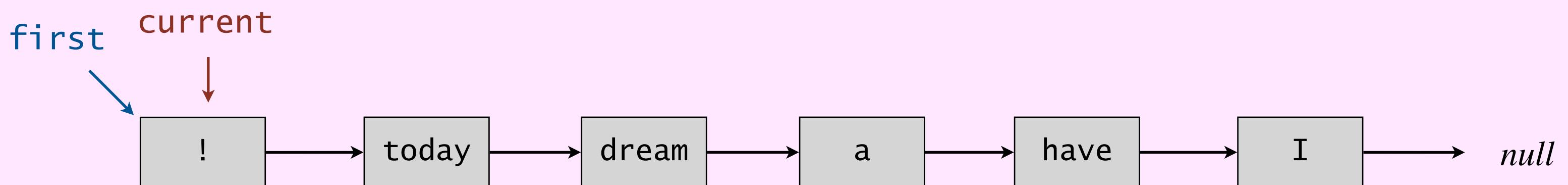
    public Iterator<Item> iterator() { return new LinkedIterator(); }

    private class LinkedIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() { return current != null; }

        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

Note: next() must throw a
NoSuchElementException
if called after traversal is done





Suppose that you add A, B, and C to a **stack** (linked list or resizing array), in that order.

What does the following code fragment do?

```
for (String s : stack)
    for (String t : stack)
        StdOut.println(s + "-" + t);
```

- A. Prints A-A A-B A-C B-A B-B B-C C-A C-B C-C
- B. Prints C-C B-B A-A
- C. Prints C-C C-B C-A
- D. Prints C-C C-B C-A B-C B-B B-A A-C A-B A-A
- E. Depends on the implementation.



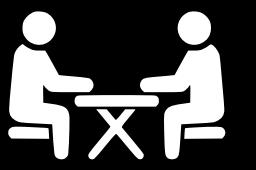
Suppose that you add A, B, and C to a **stack** (linked list or resizing array), in that order.

What does the following code fragment do?

```
for (String s : stack)
{
    StdOut.println(s);
    StdOut.println(stack.pop());
    stack.push(s);
}
```

- A. Prints A A B B C C
- B. Prints C C B B A A
- C. Prints C C B C A B
- D. Prints C C C C C C C C ...
- E. Depends on the implementation.

ITERATION: CONCURRENT MODIFICATION



Q. What should happen if a client modifies a collection **while** traversing it?

A. A **fail-fast iterator** throws a `java.util.ConcurrentModificationException`.

concurrent modification

```
for (String s : stack)
    stack.push(s);
```

Q. How to detect concurrent modification?

A.

Java iterators summary

Iterator and Iterable. Two Java interfaces that allow a client to iterate over the items in a collection, without exposing the collection's internal representation.

```
Stack<String> stack = new Stack<>();  
...  
for (String s : stack) {  
    ...  
}
```

This course.

- Yes: use iterators in client code.
- Yes: implement iterators (Assignment 2 only).

*“I was interested in doing it, there was an opportunity,
so I just did it.” — Barbara Liskov*

