Advanced Java, part 1

CS121: Data Structures

START RECORDING

Outline

- Attendance quiz
- Inheritance
- Interfaces
- Iterators
- Exceptions

Attendance Quiz

Attendance Quiz: Stacks and Queues

- Scan the QR code, or find today's attendance quiz under the "Quizzes" tab on Canvas
- Password: to be announced

```
public class Stack<Item>
{
   private Node first = null;
   private int N = 0;

   private class Node
   {
      private Item item;
      private Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public int size()
   {  return N;  }

   public void push(Item item) {
   // TODO
   }

   public Item pop() {
   // TODO
   }
}
```

Attendance Quiz: Stacks and Queues

- Write your name
- Briefly describe one similarity and one difference between Stacks and Queues
- Complete the Stack implementation by writing code for push() and pop()

```
public class Stack<Item>
{
   private Node first = null;
   private int N = 0;

   private class Node
   {
      private Item item;
      private Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public int size()
   {  return N;  }

   public void push(Item item) {
   // TODO
   }

   public Item pop() {
   // TODO
   }
}
```

Sunday TA Hours Change

Motivation

- How can `System.out.println()` accept any type of object?
- How to iterate over arrays, linked lists, stacks, queues, etc., using an approach that works for any collection?
- How to gracefully handle exceptions (e.g., division by zero)?

https://algs4.cs.princeton.edu

Advanced Java

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

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

Advanced Java

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

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).

- Define a new class (subclass) from another class (base class or superclass).
- The subclass inherits from the base class:
  - instance variables (state)
  - instance methods (behavior)
- The subclass can override instance methods in the base class (replacing with own versions).

Main benefits.

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

Inheritance example

```
public class Disc {
   protected 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);
   }
}
```

inherited by subclass

base class

```
import java.awt.Color;

public class ColoredDisc extends Disc {

   protected Color color;

   public ColoredDisc(int x, int y, int r, Color color) {
      super(x, y, r);
      this.color = color;
   }

   public Color getColor() {
      return color;
   }

   public void draw() {
      StdDraw.setPenColor(color);
      StdDraw.filledCircle(x, y, r);
   }

}
```

defines new state

calls constructor in base class

defines new behavior

overrides method in base class

subclass

Advanced Java: quiz 1

Which color will be stored in the variable color?

```java
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. 💣

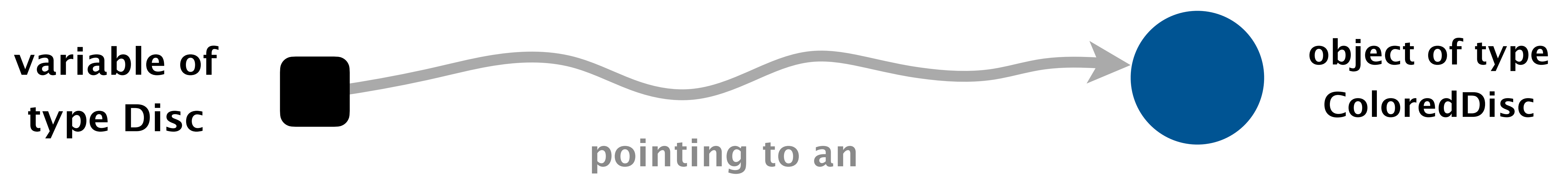
Polymorphism

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, ...

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

```
Disc disc = new ColoredDisc(x, y, r, color);
```



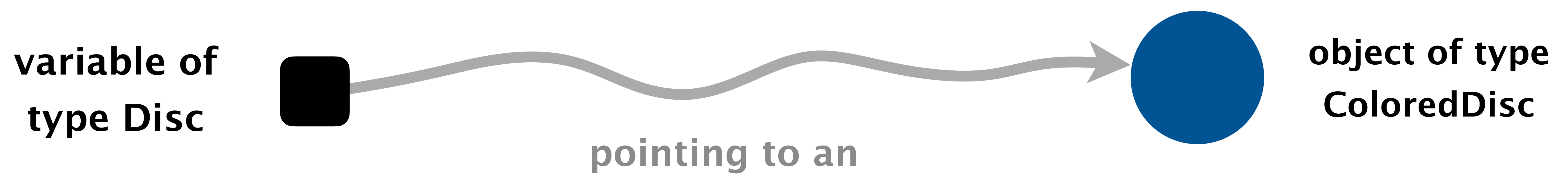
```
double area = disc.area();
boolean disc.intersects(disc);
~~Color color = disc.getColor();~~
```

can call only Disc methods
(compile-time error)

Polymorphism

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).

```
Disc disc = new ColoredDisc(x, y, r, color);
```



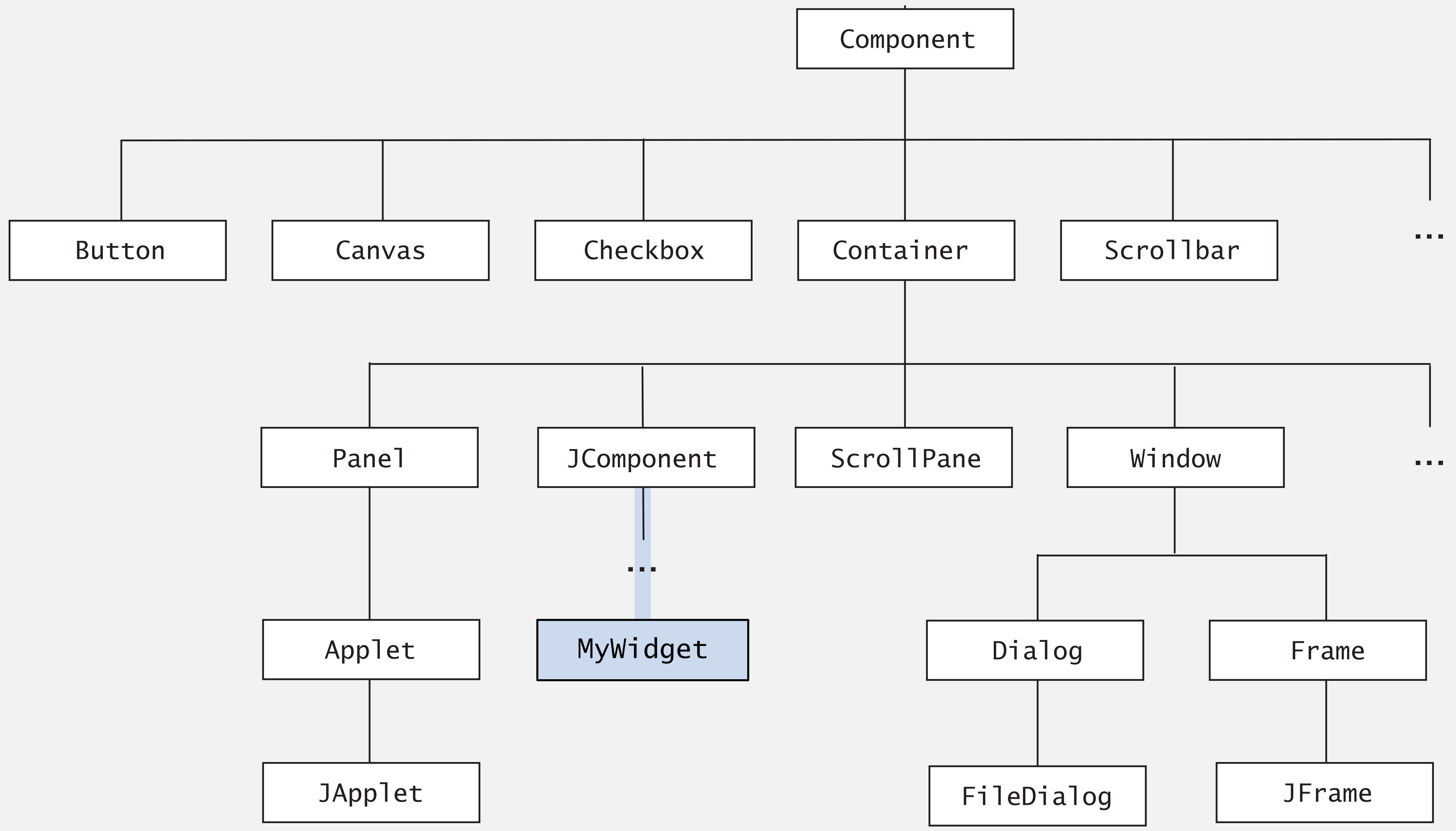
```
disc.draw();
```
← calls `ColoredDisc` version of `draw()`

a "polymorphic" method call

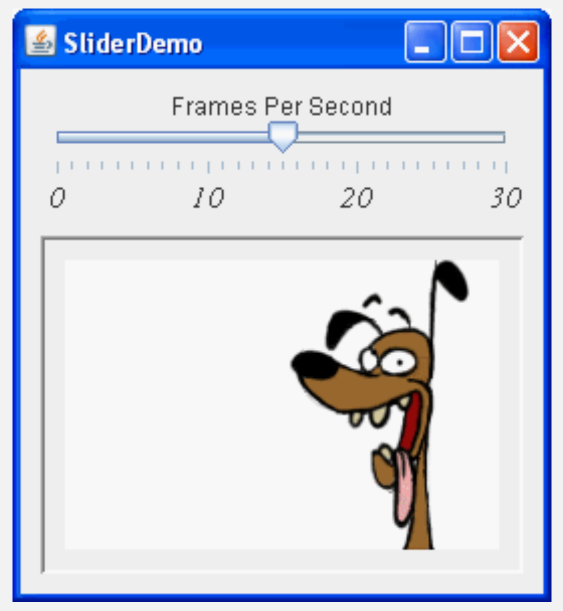
Subclass hierarchy for Java GUI components

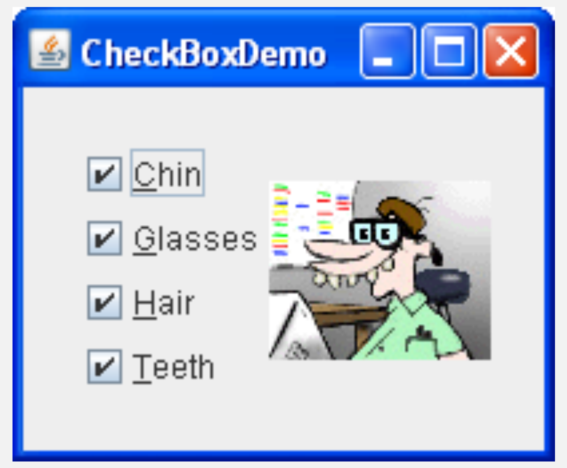
Typical use case. Design an extensible library.

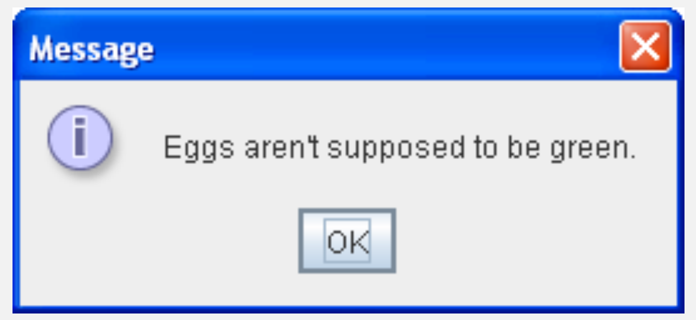
Ex. Android developer design a new GUI widget for their app.



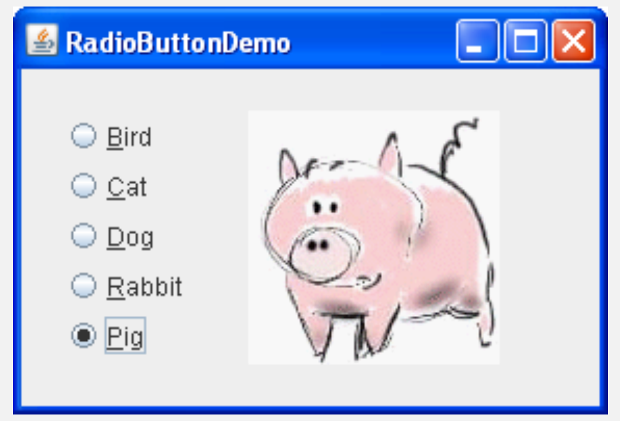
Java GUI class hierarchy

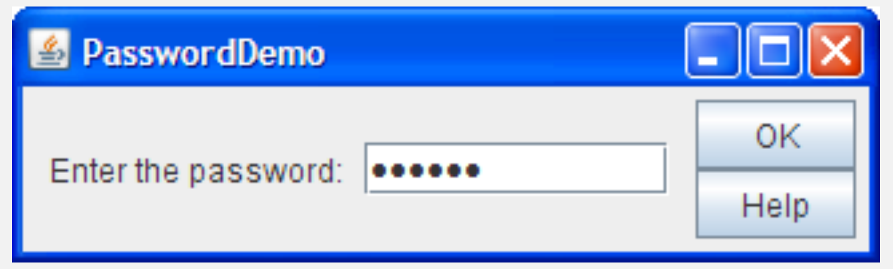






Eggs aren't supposed to be green.





Is-A relationship

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

subclass	base class
`ColoredDisc`	`Disc`
`ArithmeticException`	`RuntimeException`
`JPasswordField`	`JTextField`
`Jeans`	`Clothing`
`SamsungGalaxyS10`	`SmartPhone`

Barbara Liskov
Turing Award 2008

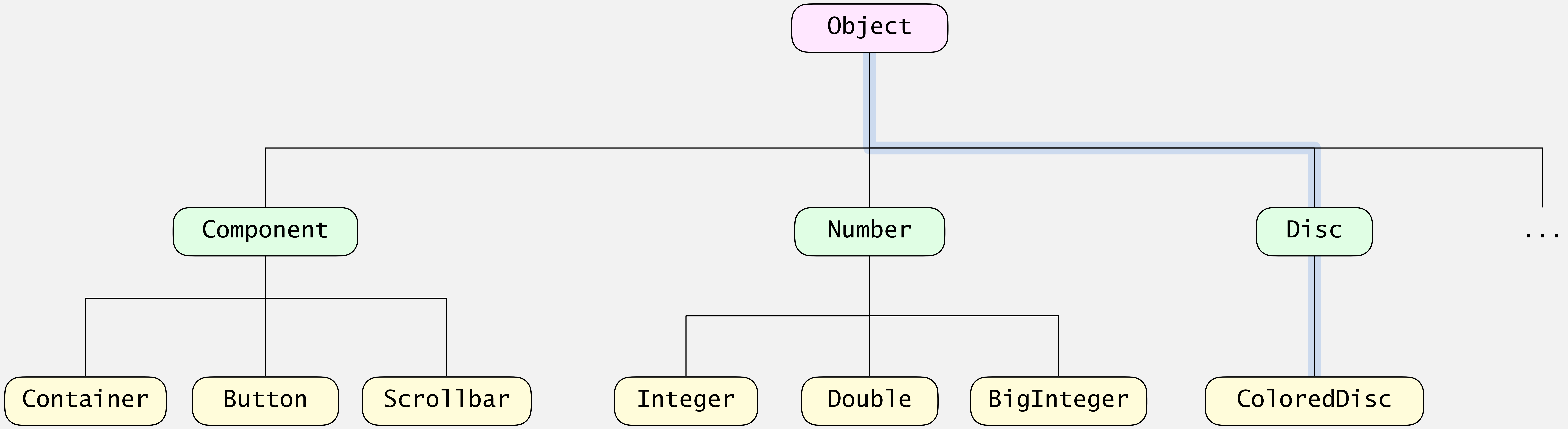
Liskov substitution principle. Subclass objects must always be substitutable for base class objects, without altering desirable properties of program.

Java's Object superclass

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

`public class Object`	
`String toString()`	*string representation*
`boolean equals(Object x)`	*is this object equal to x ?*
`int hashCode()`	*hash code of this object*
`Class getClass()`	*runtime class of this object*
`...`	*copying, garbage collection, concurrency*

Inherited methods. Often not what you want ⇒ override them.

- Equals: reference equality (same as ==).
- Hash code: memory address of object.
- String representation: name of class, followed by `@`, followed by memory address.

The toString() method

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

```
public class Disc {
   protected 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-algs4
/open Disc.java
Disc disc = new Disc(100, 100, 20);
StdOut.println("disc = " + disc.toString());
disc = Disc@239963d8
```

after overriding toString() method

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

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

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

string concatenation operator

Inheritance summary

Subclassing. Powerful OOP mechanism for code reuse.

Limitations.

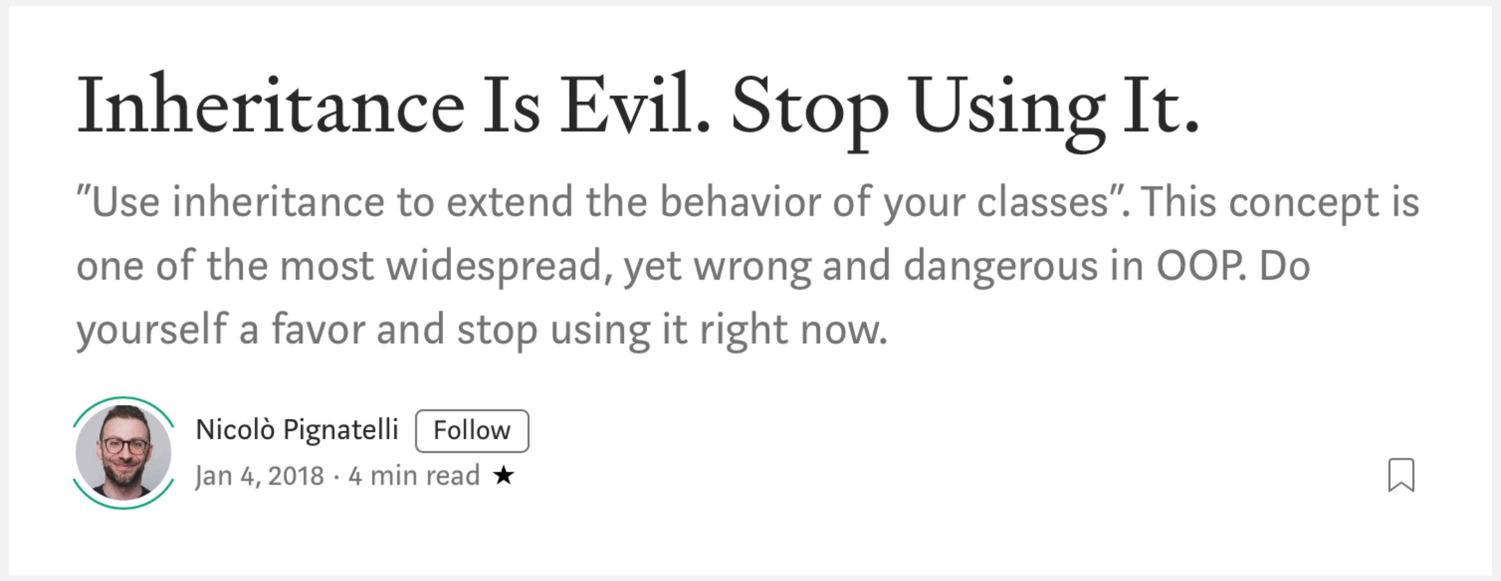
- Violates encapsulation.
- Stuck with inherited instance variables and methods forever.
- Subclasses may break with seemingly innocuous change to superclass.

Best practices.

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

This course.

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



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 ★

https://codeburst.io/inheritance-is-evil-stop-using-it-6c4f1caf5117

https://algs4.cs.princeton.edu

Advanced Java

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

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> = new Stack<>();
stack.push("First");
stack.push("Whitman");
stack.push("Mathey");

for (String s : stack)
   StdOut.println(s);
```

iterate over a collection

Java interfaces overview

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

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

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

class promises to honor the contract

```java
public class Disc implements Shape2D {
   protected int x, y, r;

   public Disc(double x, double y, double r) {
      this.x = x;
      this.y = y;
      this.r = r;
   }

   public void draw() {
      StdDraw.filledCircle(x, y, r);
   }

   public boolean contains(int x0, int y0) {
      int dx = x - x0;
      int dy = y - y0;
      return dx*dx + dy*dy <= r*r;
   }

   public boolean intersects(Disc that) {
      ...
   }
}
```

class abides by the contract

class can define additional methods

Java interfaces overview

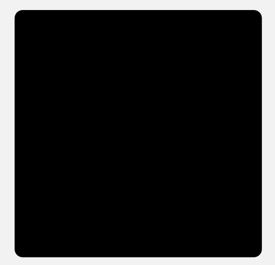
Interface. A set of methods that define some behavior (partial API) for a class.

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

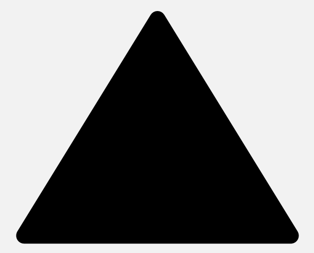
the contract: methods with these signatures
(and prescribed behaviors)

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

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

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

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

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

Java interface properties

Interfaces are reference types. Can declare variables or uses as argument/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 {
   ...
}
```

Advanced Java: quiz 2

Which of the following statement(s) 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 writing clear, maintainable code 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.

This course.

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

https://algs4.cs.princeton.edu

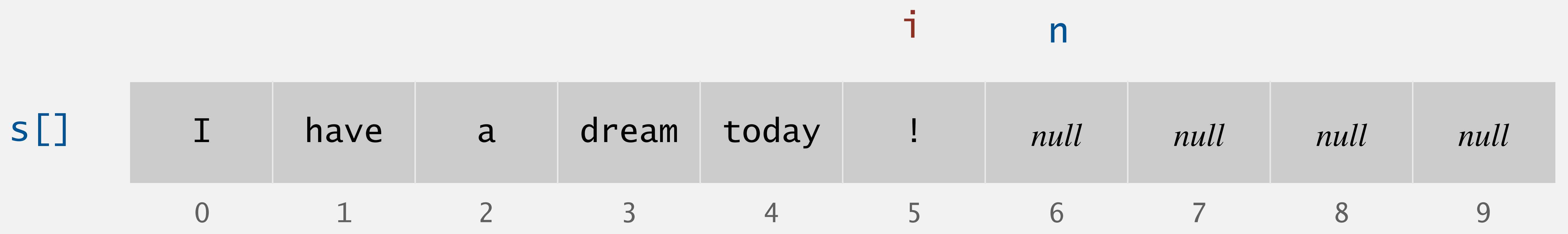
Advanced Java

- inheritance
- interfaces
- iterators

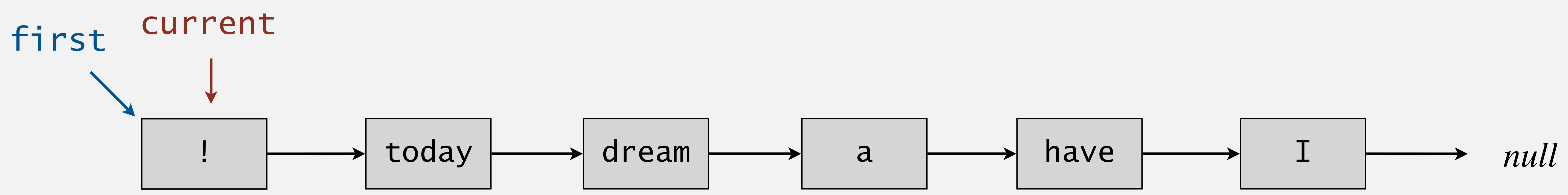
Iteration

Design challenge. Allow client to iterate over items in a collection (e.g., a stack), without exposing its 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 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 make user-defined collection support foreach loop:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an `Iterator` object that has two core method:
  - the `hasNext()` methods returns `false` when there are no more items
  - the `next()` method returns the next item in the collection

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" (supports multiple iterators over the same collection)
- Each interface is generic.

java.lang.Iterable interface

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

java.util.Iterator interface

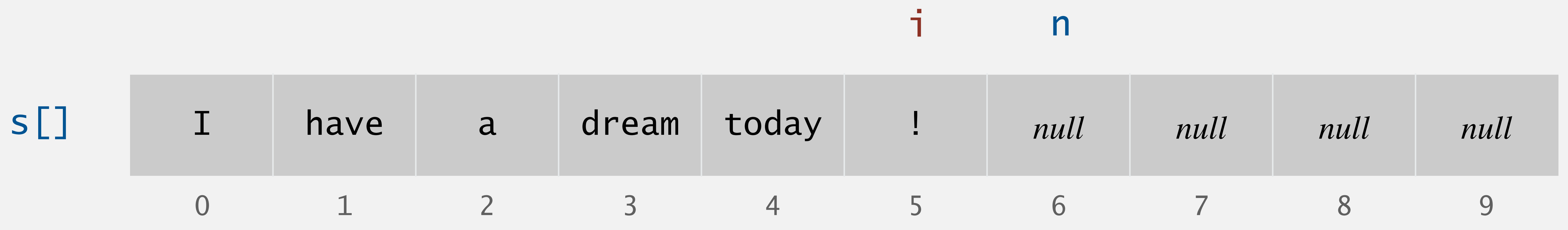
```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
}
```

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--];    }
  }

}
```

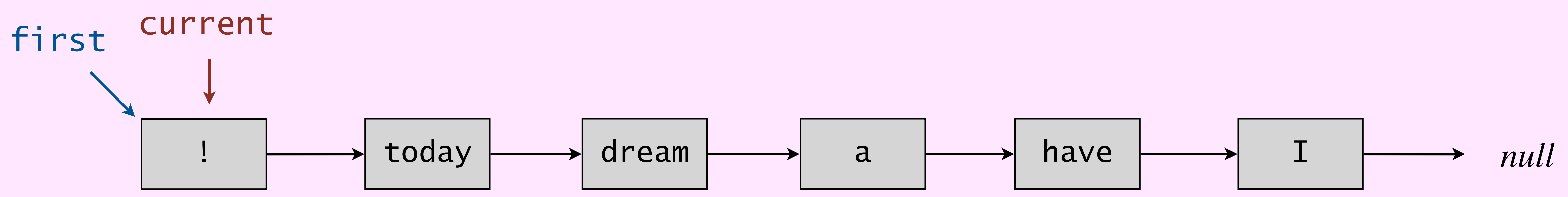
Note: next() must throw a NoSuchElementException if called when no more items in iteration



Stack iterator: linked-list implementation

```
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 when called with no more items in iteration



Advanced Java: quiz 3

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 upon implementation.

Advanced Java: quiz 4

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 implementation.

Java iterators summary

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

```
Stack<String> stack = new Stack<>();
...

for (String s : stack) {
  ...
}
```

This course.

- Yes: use iterators in client code.
- **And implementing some iterators as well!**

https://algs4.cs.princeton.edu

Advanced Java

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

© Copyright 2021 Robert Sedgewick and Kevin Wayne