Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

Advanced Java

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

Last updated on 9/9/20 4:54 PM

Advanced Java

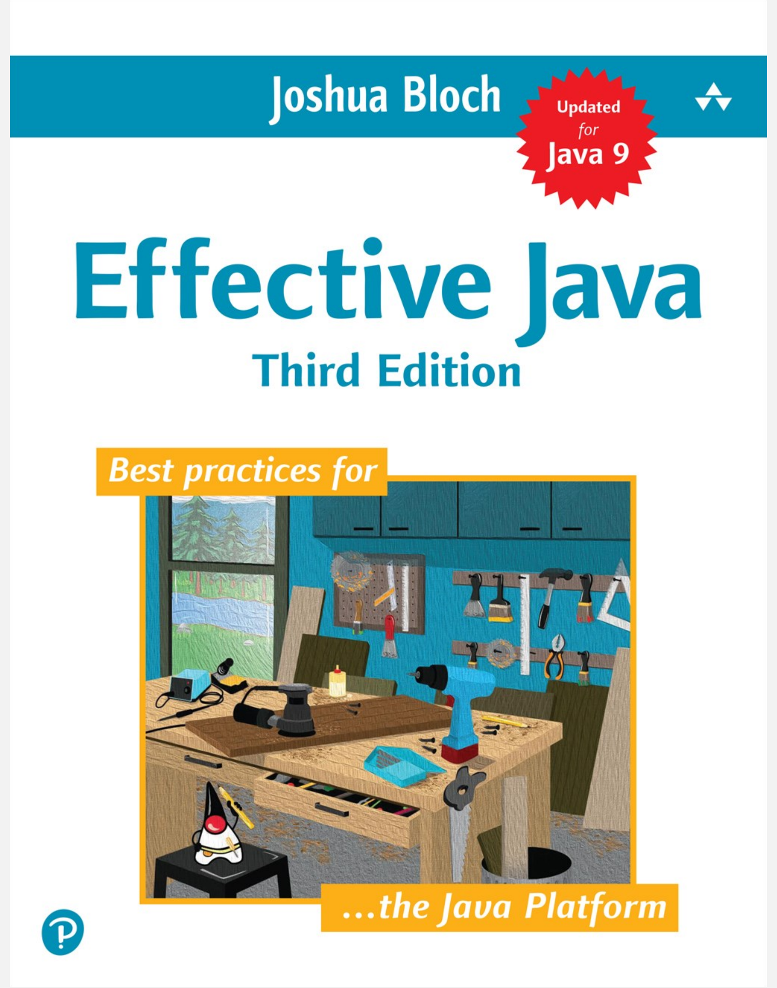
Subtitle. Java features that we (occasionally) use in this course, but don't cover (much) in COS 126.

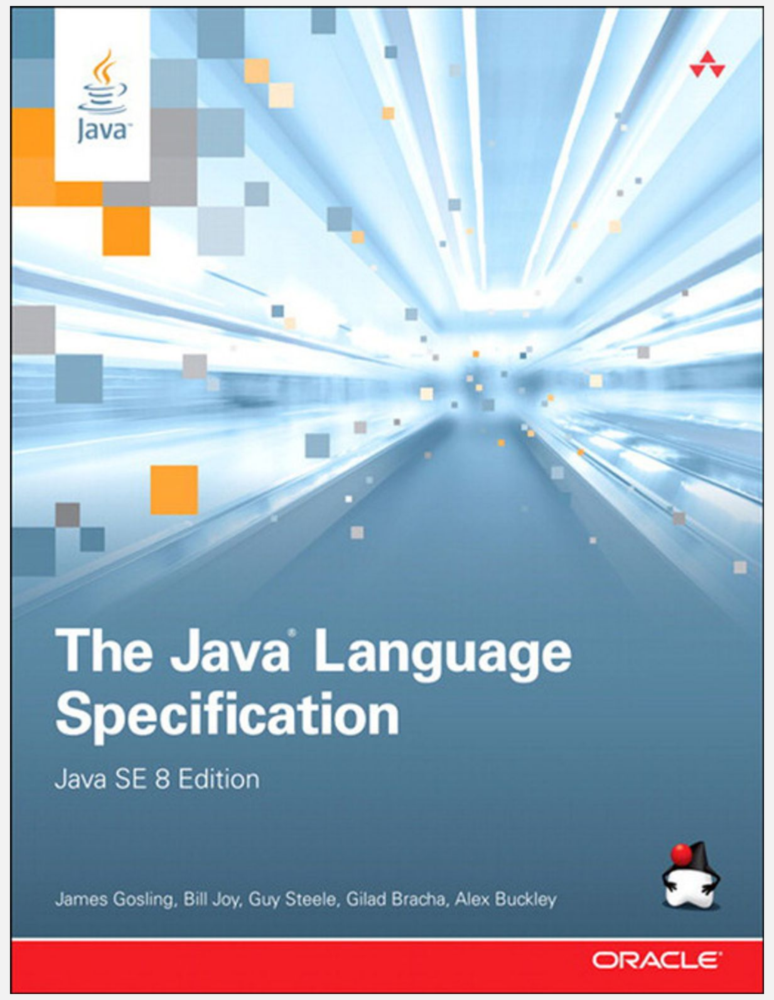
- Inheritance.
- Generics.
- Interfaces.

← common theme: promote code reuse

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

A.




Java
The Java Language Specification
Java SE 8 Edition
James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley
ORACLE

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

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

Advanced Java

- *inheritance*
- *generics*
- *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 {
   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);
   }
}
```

base class

```
import java.awt.Color;

public class ColoredDisc extends Disc {

   private final 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);
      super.draw();
   }

}
```

subclass

← defines new state (private final Color color;)

← calls base class constructor (super(x, y, r);)

← defines new behavior; inherits area() and intersects() (public Color getColor())

← overrides method in base class (public void draw())

Inheritance demo (in JShell)

```
~/Desktop/advanced-java> jshell-algs4
/open Shape2D.java
/open Disc.java
/open ColoredDisc.java

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

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

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


Disc disc = disc2     // downcast
disc.area()
```

Advanced Java: quiz 1

Which color will be stored in the variable x?

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

A. Blue.

B. Black.

C. Compile-time error.

D. Run-time error.

E. 💣

Polymorphism

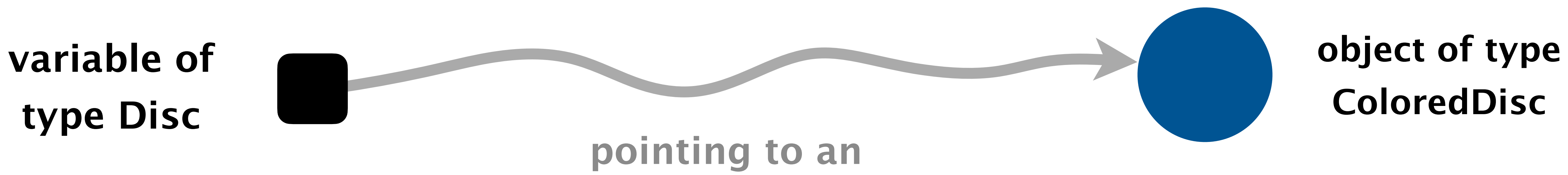
Subtype polymorphism. A subclass is a subtype of its superclass:

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);
```



variable of type Disc

pointing to an

object of type ColoredDisc

```
double area = disc.area();
boolean disc.intersects(disc);
```

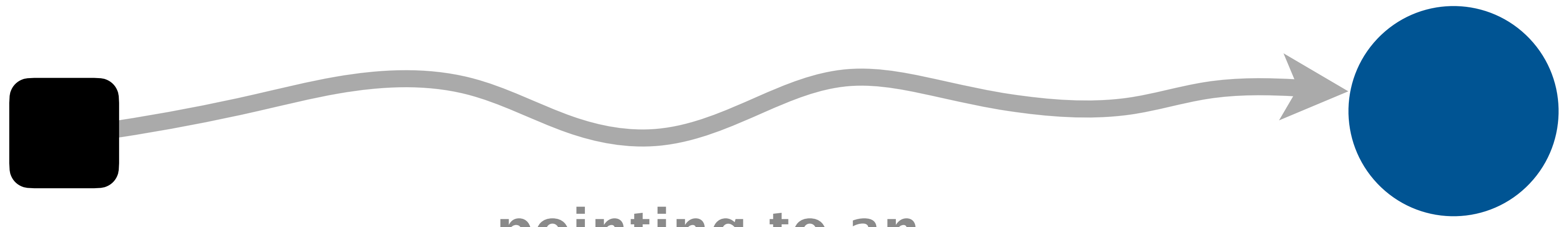
~~Color color = disc.getColor();~~ ← can call only Disc methods (compile-time error)

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);
```

variable of type Disc



pointing to an

object of type ColoredDisc

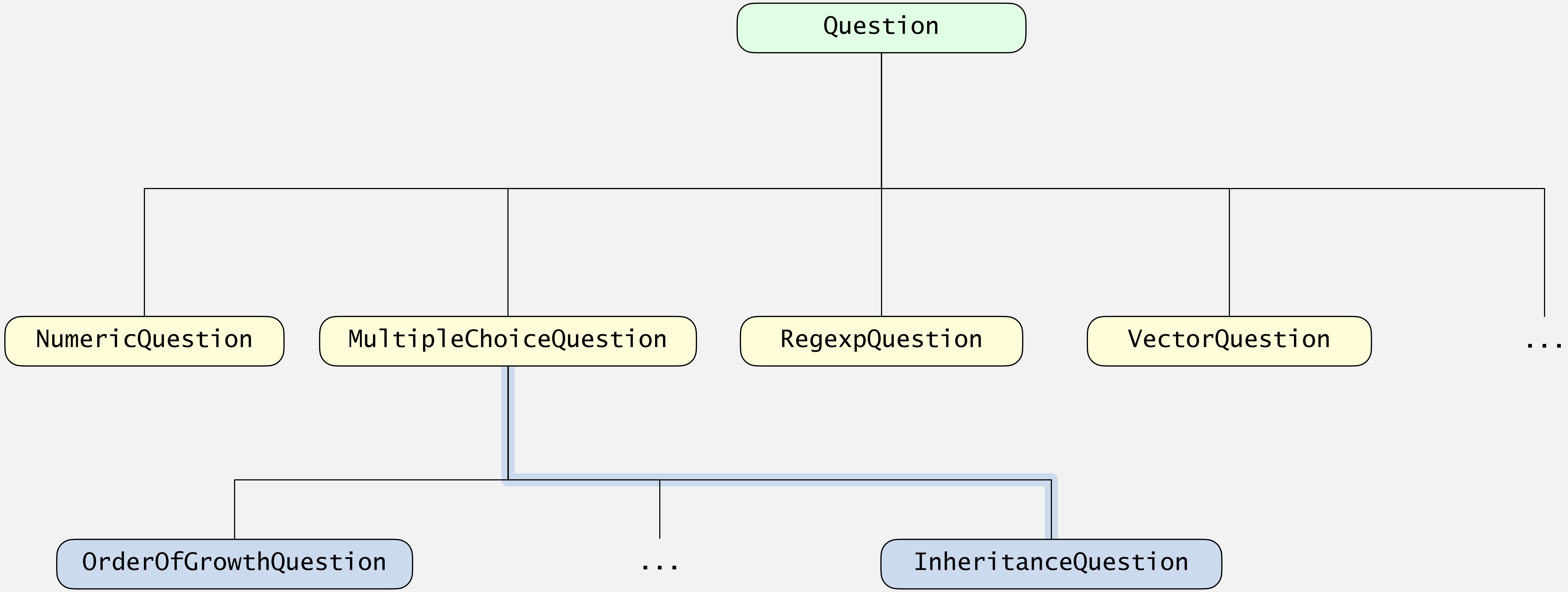
```
disc.draw();   ←   calls ColoredDisc version of draw()
```

a “polymorphic” method call

Subclass hierarchy for Quizzera questions

Typical use case. Design an extensible library.

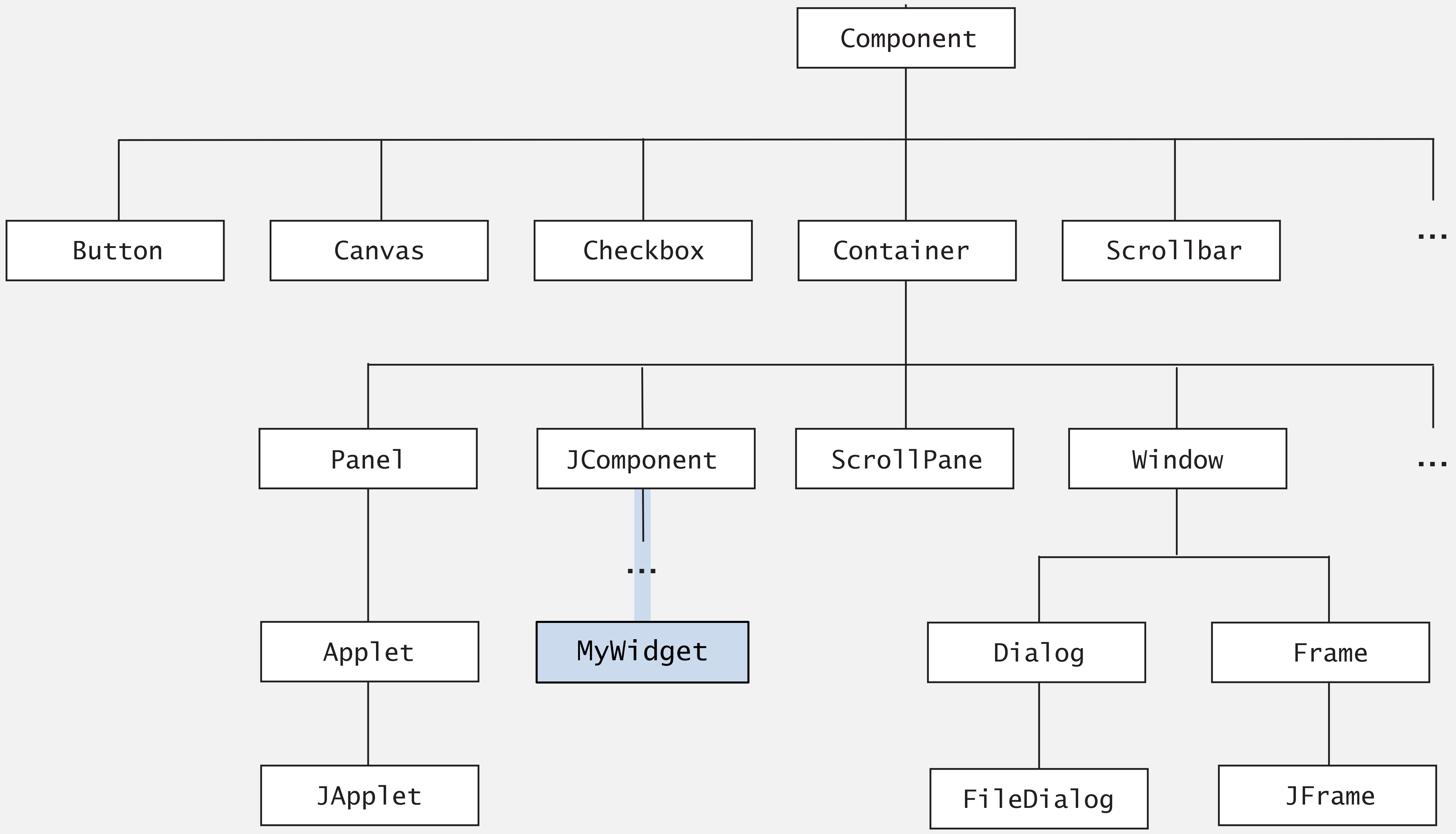
Ex. Quizzera user defines an `InheritanceQuestion` class.



Subclass hierarchy for Java GUI components

Typical use case. Design an extensible library.

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



Component

Button

Canvas

Checkbox

Container

Scrollbar

...

Panel

JComponent

ScrollPane

Window

...

Applet

...

MyWidget

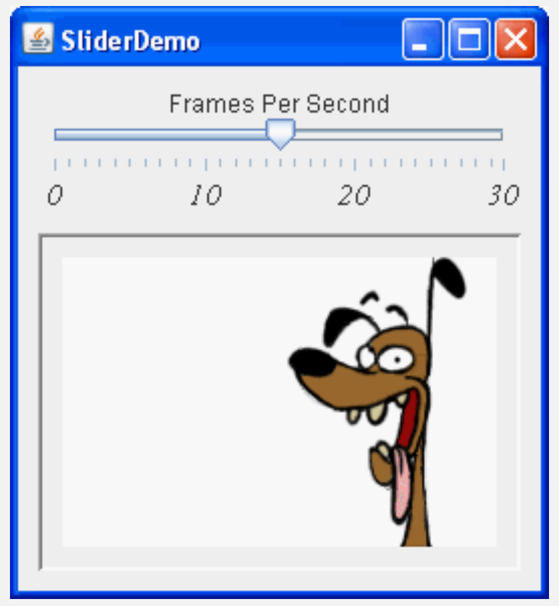
Dialog

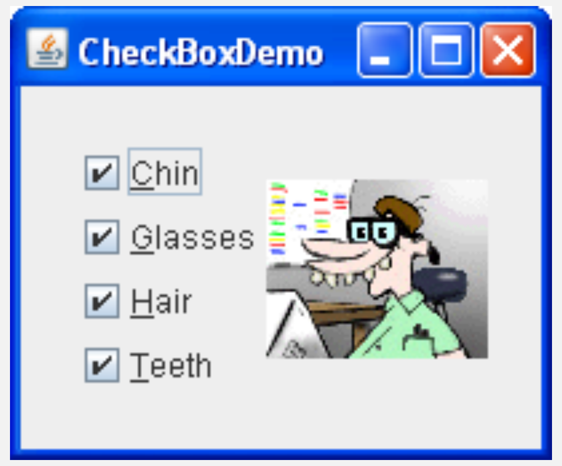
Frame

JApplet

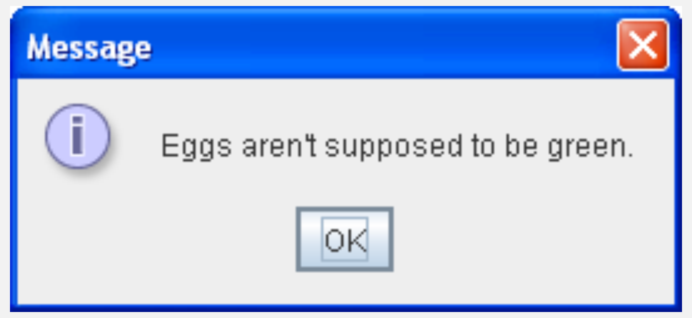
FileDialog

JFrame

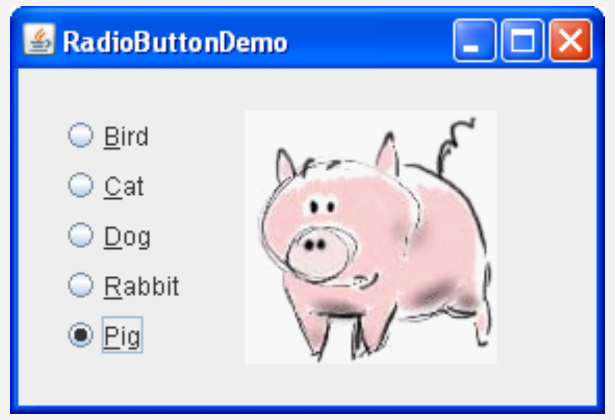




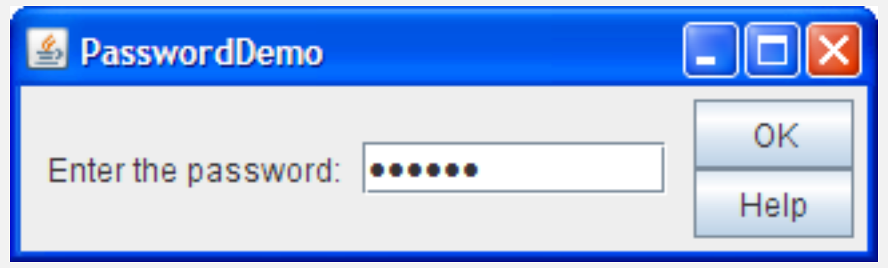
☑ Chin
☑ Glasses
☑ Hair
☑ Teeth



Eggs aren't supposed to be green.



○ Bird
○ Cat
○ Dog
○ Rabbit
● Pig



Enter the password: ••••••

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

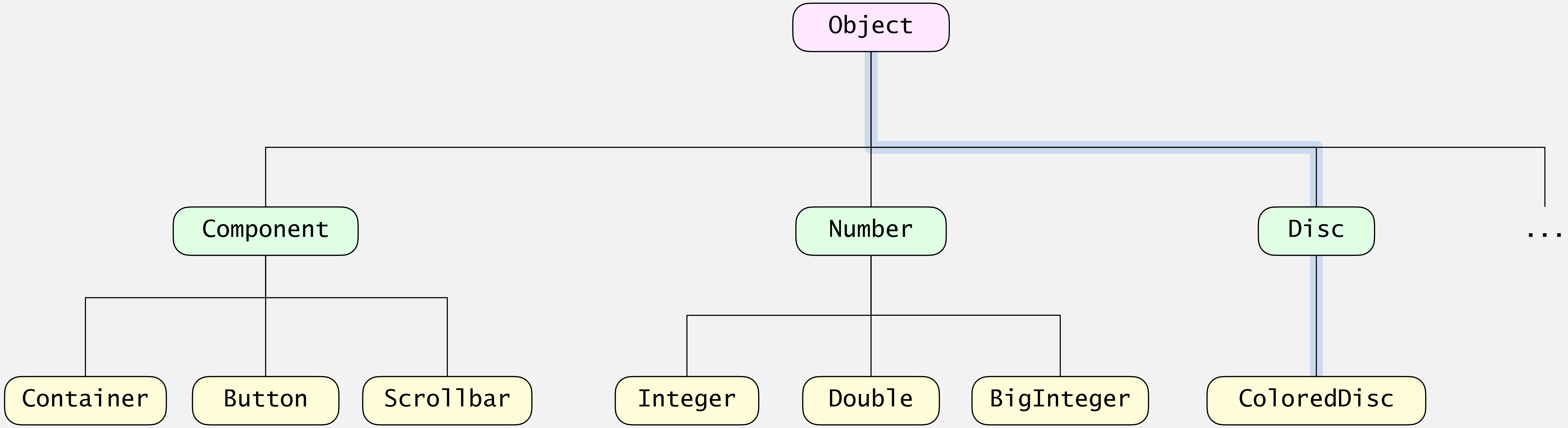
Liskov substitution principle. Subclass objects must always be substitutable for base class objects, without altering desirable properties of program.

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'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.

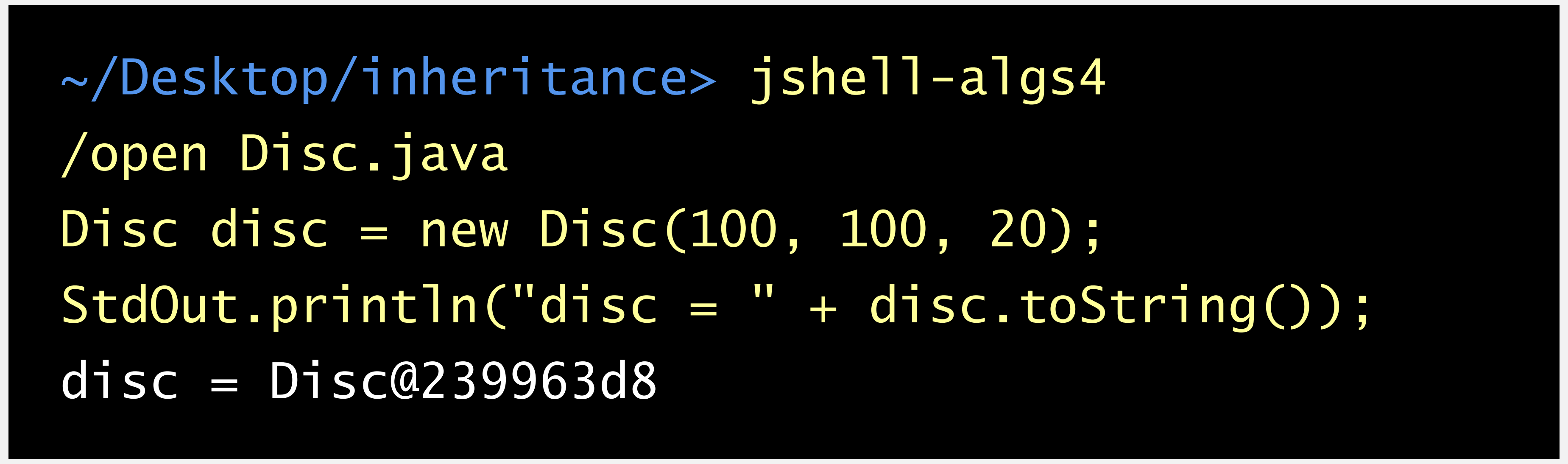
The toString() method

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

```
public class Disc {
   private final int x, y, r;

   ...

   public String toString() {
     return String.format("(%d, %d, %d)", x, y, r);
   }

}
```

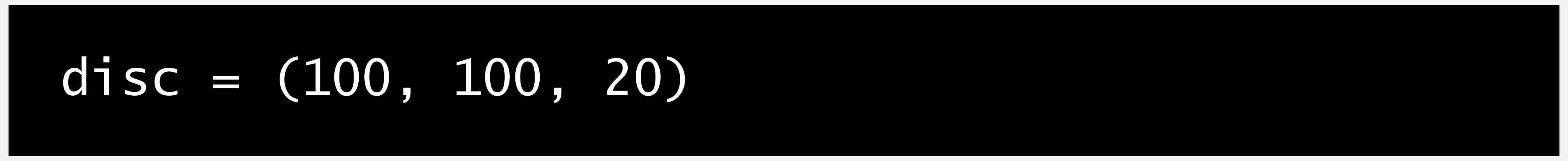
works like `printf()` but returns string
(instead of printing it)

without overriding toString() method



```
~/Desktop/inheritance> jshell-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);
```

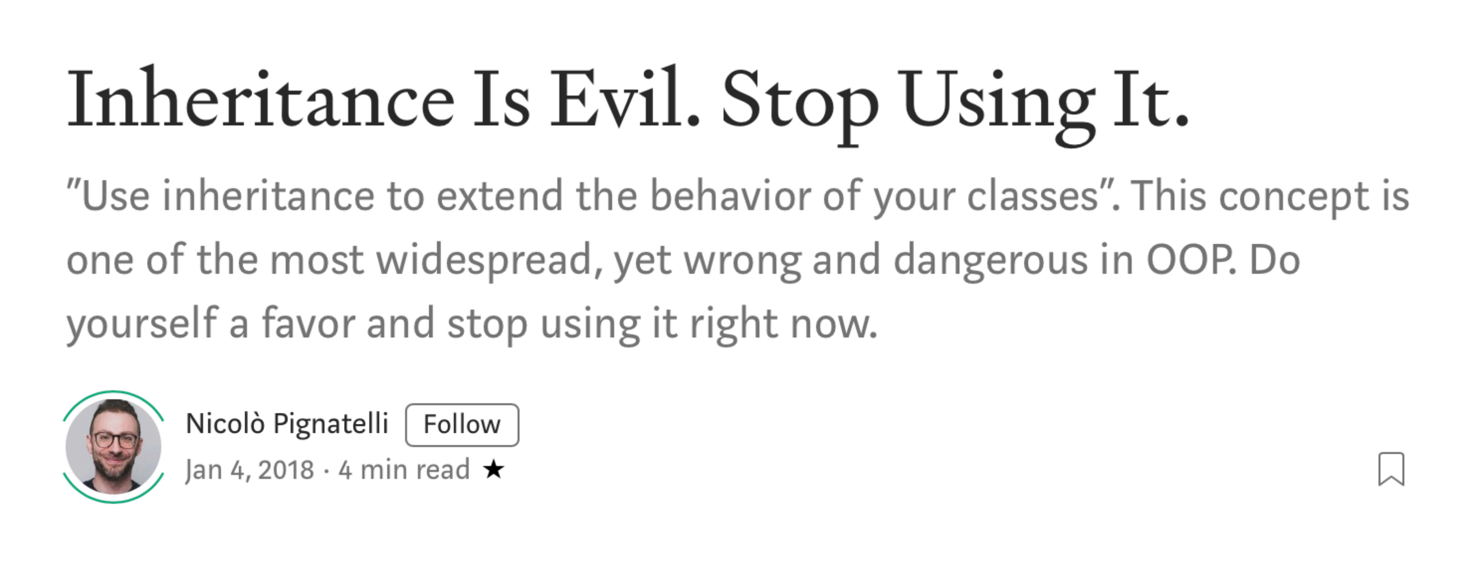
string concatenation operator

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.



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 subclass hierarchies.

https://algs4.cs.princeton.edu

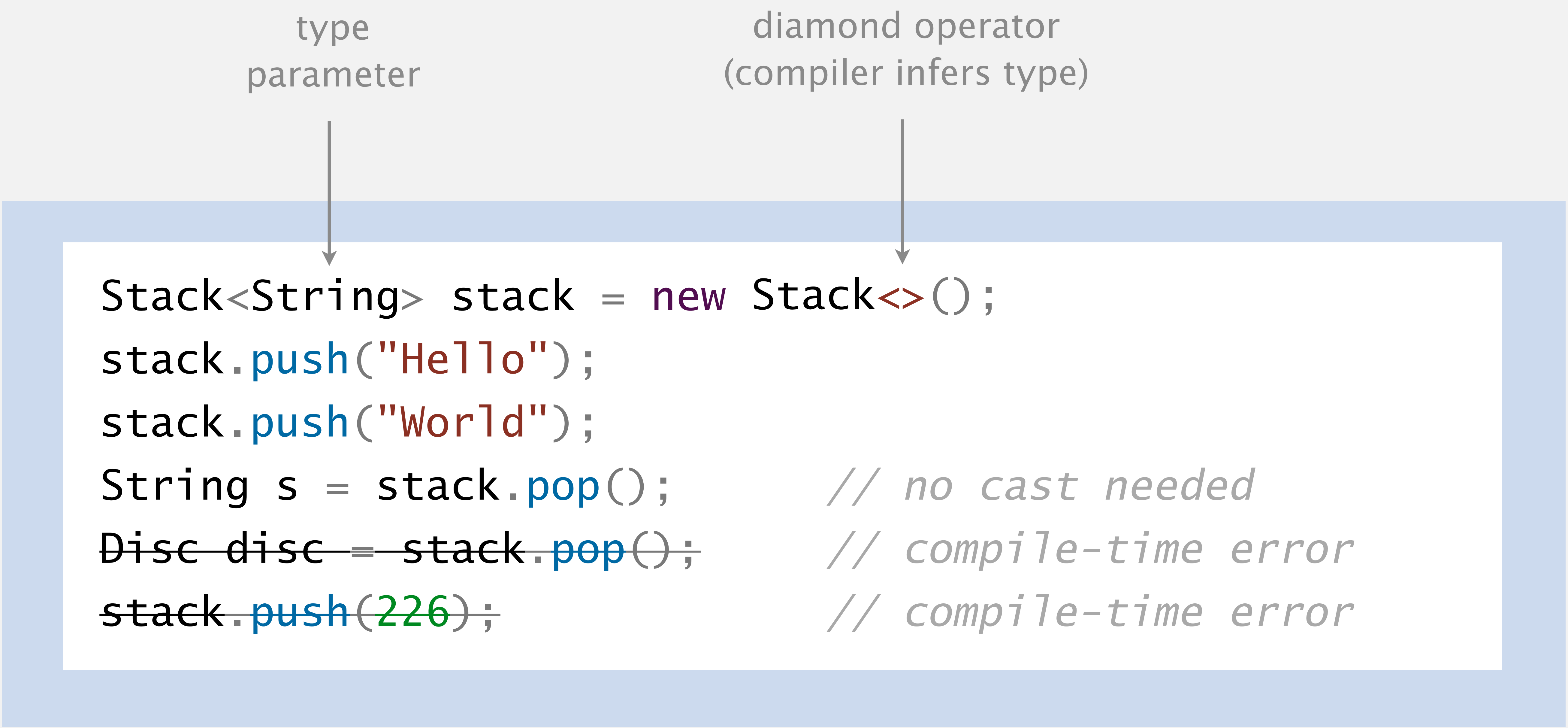
Advanced Java

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

Motivation

Q. How to create a data type that can store collections of a user-specified type?

A. Java generics.



```
Stack<String> stack = new Stack<>();
stack.push("Hello");
stack.push("World");
String s = stack.pop();      // no cast needed
~~Disc disc = stack.pop();~~     // compile-time error
~~stack.push(226);~~             // compile-time error
```

How are generic implemented in Java?

Inheritance + type erasure. Compiler checks generic types at compile time, but then erases that generic type information for generated bytecode.

programmer defines a generic type

```
public class Stack<Item> {
   private Node first;

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

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

compiler checks generic types at compile time

```
Stack<String> x = new Stack<String>();
stack.push("Hello");
String result = stack.pop();
```

How are generic implemented in Java?

Inheritance + type erasure. Compiler checks generic types at compile time, but then erases that generic type information for generated bytecode.

Java compiler removes the generic types

```
public class Stack {
   private Node first;

   public class Node {
      private Object item;
      private Node next;
   }

   public Object pop() {
      Item item = first.item;
      first = first.next;
      return item;
   }
   ...
}
```

and adds casts, as needed

```
Stack x = new Stack();
stack.push("Hello");
String result = (String) x.pop();
```

cast added by compiler

How are generic implemented in Java?

Inheritance + type erasure. Compiler checks generic types at compile time, but then erases that generic type information for generated bytecode.

Consequence 1. No generic type information available at runtime.

Consequence 2. No generic array creation.

Q. Why erase the type information for the generated bytecode?

A. Backward compatibility with old JVM bytecode.

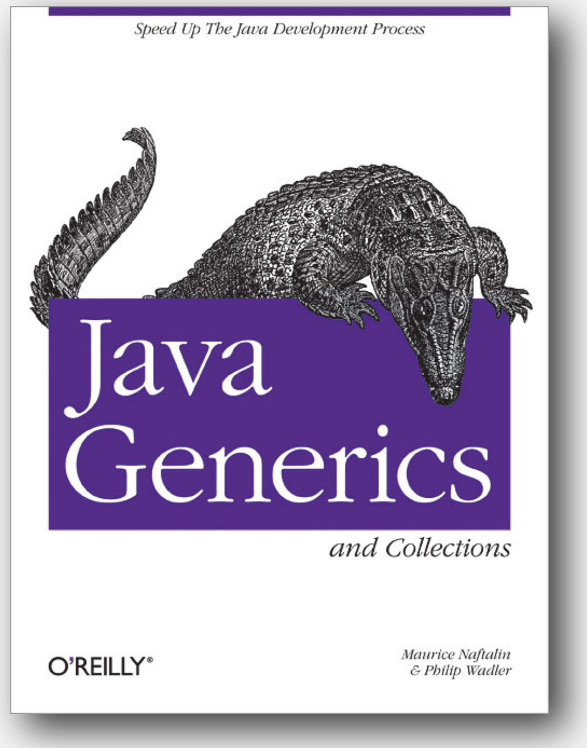
Java generics summary

Advantages.

- Code reuse (“generic programming”).
- Clear and concise code.
- Type checking (by compiler and IDE).

Disadvantages.

- Performance overhead with primitive types.
- Awkward implementation (e.g., no generic array creation).
- Complex when combined with inheritance.


Speed Up The Java Development Process
Java Generics
and Collections
O'REILLY
Maurice Naftalin & Philip Wadler

Lesson. Hard to evolve language while maintaining backward compatibility.

This course. Embrace generics in client code; define collections using generic type parameters.

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

Advanced Java

- *inheritance*
- *generics*
- ***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.

```
public interface Shape2D {
   double area();
   boolean contains(int x0, int y0);
}
```

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

class promises to
honor the contract

```
public class Disc implements Shape2D  {
   private final int x, y, r;

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

   public double area() {
      return Math.PI * r * 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

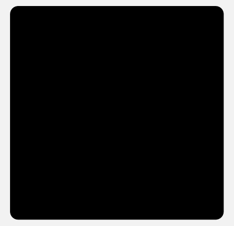
Interface. A set of methods that define some behavior (partial API) for a class.

```
public interface Shape2D {
   double area();
   boolean contains(int x0, int y0);
}
```

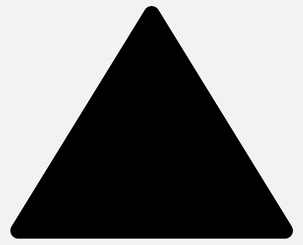
the contract: methods with these signatures
(and prescribed behaviors)

Many classes can implement the same interface.

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



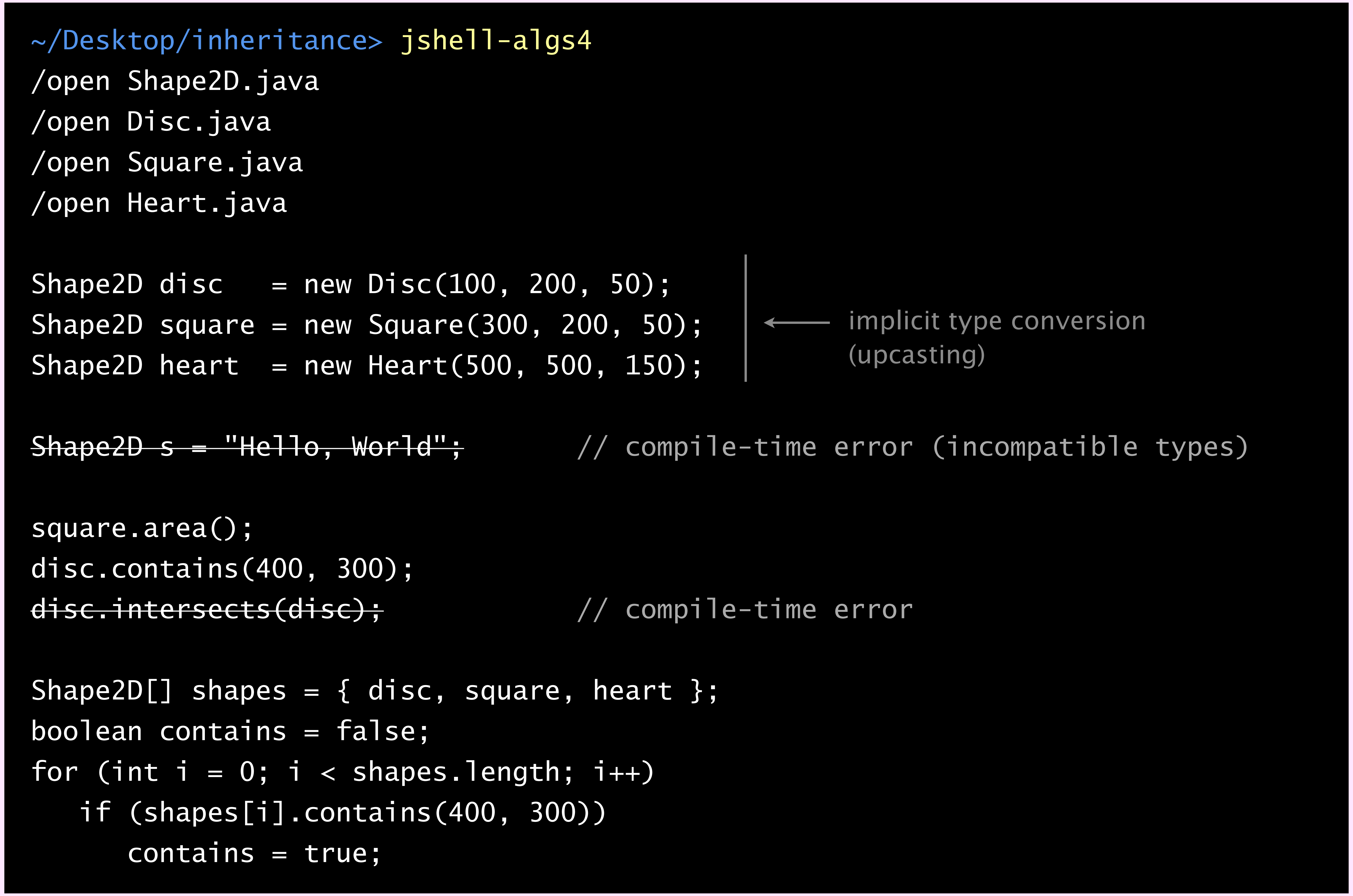
```
public class Triangle implements Shape2D {
   ...
}
```



```
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(100, 200, 50);
Shape2D square = new Square(300, 200, 50);
Shape2D heart  = new Heart(500, 500, 150);

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

square.area();
disc.contains(400, 300);
~~disc.intersects(disc);~~            // compile-time error

Shape2D[] shapes = { disc, square, heart };
boolean contains = false;
for (int i = 0; i < shapes.length; i++)
   if (shapes[i].contains(400, 300))
      contains = true;
```

← implicit type conversion
(upcasting)

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, ...

Multiple interfaces. A class can implement many interfaces.

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

Advanced Java: quiz 2

Which of the following statements lead to compile-time errors?

A. `Shape2D x = new Shape2D();`

B. `Shape2D[] a = 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.

This course.

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

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

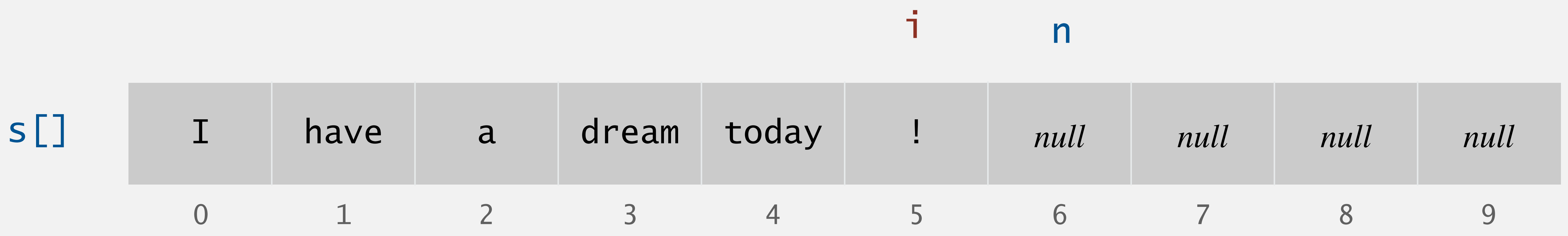
Advanced Java

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

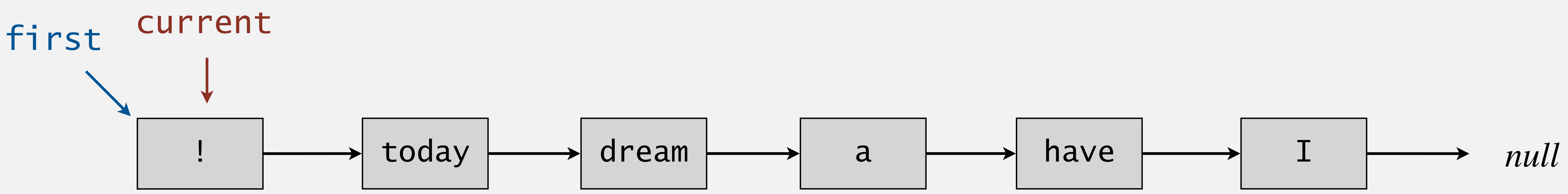
Iteration

Design challenge. Allow client to iterate over items in a collection (e.g., a stack), without exposing its internal representation.

resizing-array representation



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"
- Both should be used with generics.

java.lang.Iterable interface

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

java.util.Iterator interface

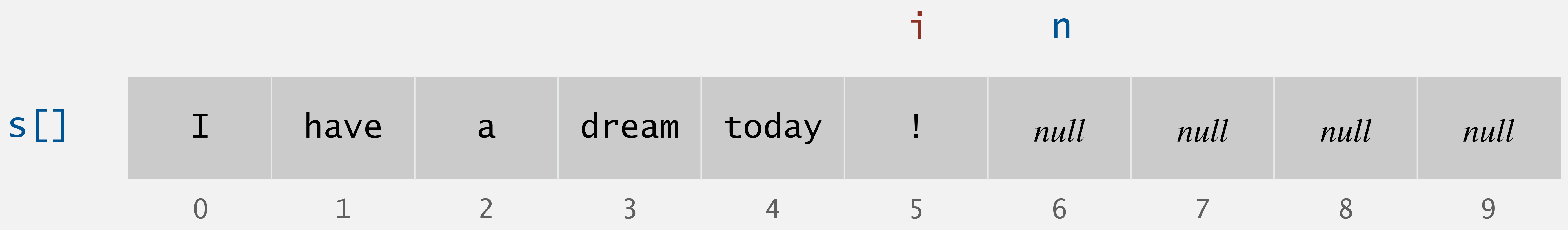
```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
}
```

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--];    }
   }

}
```

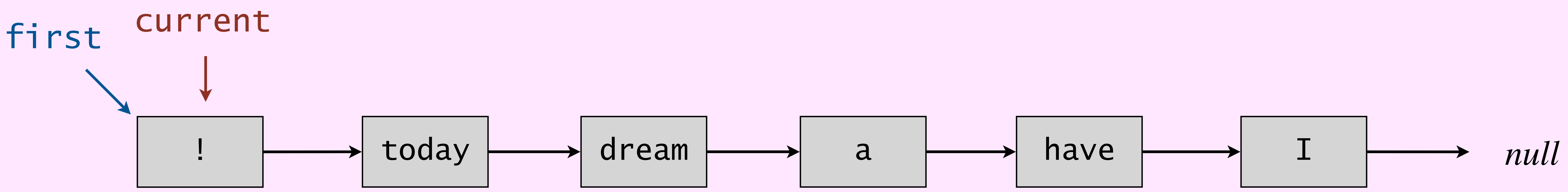
Note: next() must also throw a NoSuchElementException if no more items in iteration



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 also throw a NoSuchElementException if 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. Run-time error (two iterators at same time).

Advanced Java: quiz 4

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

What does the following code fragment do?

```
for (String s : queue)
{
   StdOut.println(s);
   StdOut.println(queue.dequeue());
   queue.enqueue(s);
}
```

A. Prints A A B B C C

B. Prints A A C B C C

C. Prints A A B B C C A A B B C C A A B B C C . . .

D. Run-time error.

E. Depends on implementation.

ITERATION: CONCURRENT MODIFICATION

Q. What should happen if a client modifies a collection while iterating over 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.

General-purpose debugging strategies

1. Trace code by hand.
2. Add `printf()` statements to trace code by computer.
3. Test, test, test, as you code.
4. Look for off-by-one and cut-and paste errors.
5. Pay attention to corner cases.
6. Recompile early and often.
7. Run static code analysis tools (Checkstyle, SpotBugs, IntelliJ inspections).
8. Read all errors and warnings; fix as you code.
9. Use a REPL (read–evaluate–print–loop), such as JShell.
10. Use a debugger (IntelliJ debugging workshop TBA).