ADVANCED JAVA

- inheritance
- generics
- interfaces
- iterators
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.
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() •
get Autoscrolls() • 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

```java
import java.awt.Color;

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);
    }
}

public class ColoredDisc extends Disc {
    private final Color color;

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

    public Color getColor() {
        return color;
    }

    public void draw() {
        StdDraw.setPenColor(color);
        super.draw();
    }
}
```

**base class**

**subclass**
Inheritance demo (in JShell)

```java
~/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()
```
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. 💣

Advanced Java: quiz 1
Subtype polymorphism. A subclass is a subtype of its superclass: objects of the subtype can be used anywhere objects of the superclass are allowed.

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

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

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

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

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
Typical use case. Design an extensible library.

Ex. Quizzera user defines an InheritanceQuestion class.
Typical use case. Design an extensible library.
Ex. Android developer design a new GUI widget for their app.
Is-A relationship

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

<table>
<thead>
<tr>
<th>subclass</th>
<th>base class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColoredDisc</td>
<td>Disc</td>
</tr>
<tr>
<td>ArithmeticException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>JPasswordField</td>
<td>JTextField</td>
</tr>
<tr>
<td>Jeans</td>
<td>Clothing</td>
</tr>
<tr>
<td>SamsungGalaxyS10</td>
<td>Smartphone</td>
</tr>
</tbody>
</table>

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

Barbara Liskov
Turing Award 2008
Object data type. Every class has Object as a (direct or indirect) superclass.

```java
public class Disc extends Object {
    ...
}
```
Java’s Object superclass

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

```java
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.

```java
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)*

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

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

**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)
```
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.
ADVANCED JAVA

- inheritance
- generics
- interfaces
- iterators

https://algs4.cs.princeton.edu
Motivation

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

A. Java generics.

```java
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**

```java
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**

```java
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

```java
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

```java
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.

Lesson. Hard to evolve language while maintaining backward compatibility.

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

```java
String[] a = { "Apple", "Orange", "Banana" };
Arrays.sort(a);

Integer[] b = { 3, 1, 2 }
Arrays.sort(b);
```

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

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

sort arrays

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

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

```java
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 promises to honor the contract

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.

```java
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.

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

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

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

```java
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)

disc.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;
```
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.

Multiple interfaces. A class can implement many interfaces.

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

<table>
<thead>
<tr>
<th>purpose</th>
<th>built-in interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorting</td>
<td>java.lang.Comparable</td>
</tr>
<tr>
<td></td>
<td>java.util.Comparator</td>
</tr>
<tr>
<td>iteration</td>
<td>java.lang.Iterable</td>
</tr>
<tr>
<td></td>
<td>java.util.Iterator</td>
</tr>
<tr>
<td>collections</td>
<td>java.util.List</td>
</tr>
<tr>
<td></td>
<td>java.util.Map</td>
</tr>
<tr>
<td></td>
<td>java.util.Set</td>
</tr>
<tr>
<td>GUI events</td>
<td>java.awt.event.MouseListener</td>
</tr>
<tr>
<td></td>
<td>java.awt.eventKeyListener</td>
</tr>
<tr>
<td></td>
<td>java.awt.event.ActionListener</td>
</tr>
<tr>
<td>lambda</td>
<td>java.util.function.Consumer</td>
</tr>
<tr>
<td>expressions</td>
<td>java.util.function.Supplier</td>
</tr>
<tr>
<td></td>
<td>java.util.function.BinaryOperator</td>
</tr>
<tr>
<td>concurrency</td>
<td>java.lang.Runnable</td>
</tr>
<tr>
<td></td>
<td>java.lang.Callable</td>
</tr>
</tbody>
</table>
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.
ADVANCED JAVA

- inheritance
- generics
- interfaces
- iterators

https://algs4.cs.princeton.edu
**Iteration**

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

### resizing-array representation

```
0 1 2 3 4 5 6 7 8 9
have a dream today ! null null null null
```

### linked-list representation

```
first
```

```
! -> today -> dream -> a -> have -> I -> null
```

**Java solution.** Use a *foreach* loop.
Foreach loop

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

**“foreach” loop (shorthand)**

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

**equivalent code (longhand)**

```java
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 methods:
  - the `hasNext()` method 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
// 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

```java
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
```
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
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?

```java
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 A-C A-B A-A
E. Run-time error (two iterators at same time).
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?

```java
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`.

```java
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).