Warmup: any concerns?

Library code

```
/** Returns the positive
  * square root of `a`.
  * If the argument is
  * NaN or less than
  * zero, the result is
  * NaN. */
public static double
  sqrt(double a) { … }
```

Client code

```
double a1 = 16.0;
double a2 = 25.0;
double s1 =
  Math.sqrt(a1 – a2);
assert !(s1 < 0);
```

Warmup: any concerns?

Note: NaN is not less than, greater than, or equal to any value (even itself)

A. The client code is buggy – it violates a *precondition*
B. The client code is buggy – it asserts something not guaranteed by the *postcondition*
C. The library's *specification* is incomplete
D. The code shown here is fine

```
/** Returns the positive square
  * root of `a`. If the argument
  * is NaN or less than zero,
  * the result is NaN. */
public static double
  sqrt(double a) { … }
```

```
// Client code
double a1 = 16.0;
double a2 = 25.0;
double s1 =
  Math.sqrt(a1 - a2);
assert !(s1 < 0);
```

CS 2110 Lecture 5

Interfaces, subtyping, polymorphism

Coming up

A2 should be released today

A1 is being graded now

Expect feedback Thurs

iClickers

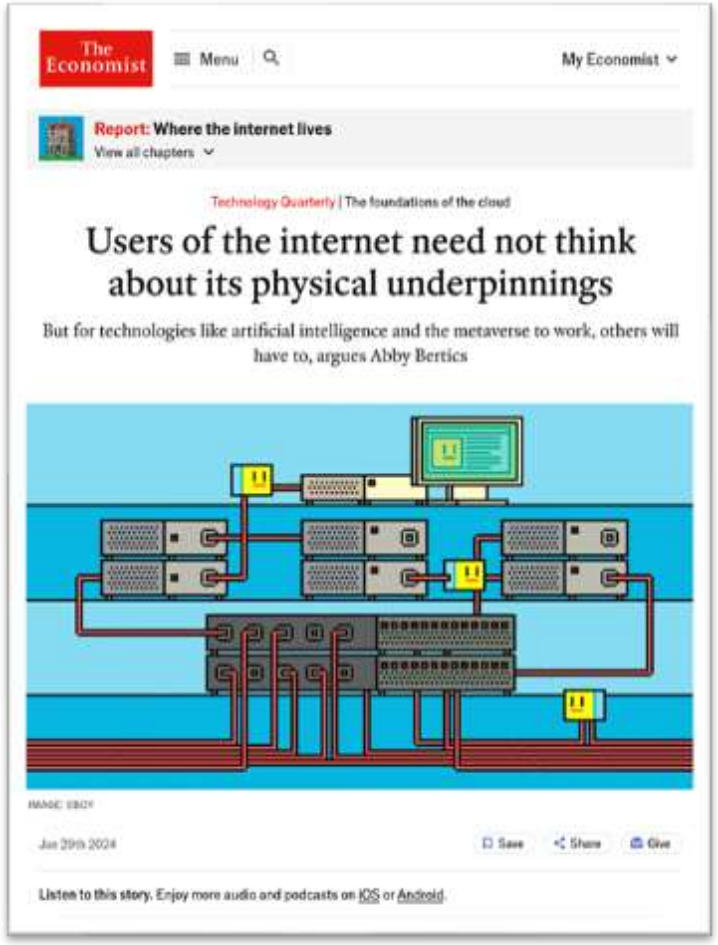
Interfaces

“Interfaces”

- Mechanism by which two parties work together, decided ahead of time
  - HDMI cable: interface between laptop and projector
  - ¼”-20 screw: interface between tripods and camera gear
  - API: interface between client programmers and classes
- Parties don’t need to know each other’s details
  - Forms an “**abstraction barrier**”

- (Client) interface = a class’s `public` methods

Abstraction reminder: separate “what” from “how”



“The power of abstraction… is the secret sauce of the internet. And, indeed, all of computer science.”

- *The Economist*, Jan 29, 2024

Discuss

In a class representing a cafeteria, someone has written a public method to predict the wait time.

What do you (the client) need to know in order to make use of this method?

1.

2.

3.

4.

Java interfaces

- Guarantee to clients what a type *can do*, without committing to details (i.e. fields)
  - Method signatures
  - Method return types
  - Method specs
- Method declarations are implicitly `public`

```
/** Closed interval on real
  * number line. */
public interface Interval {
  /** Return left endpoint. */
  double left();
  /** Return right endpoint. */
  double right();
  /** Whether x is contained
   *  in this interval. */
  boolean contains(double x);
}
```

Client code 1

```
static boolean intervalsOverlap(Interval i1,
                                Interval i2) {
    return i1.left() <= i2.right() &&
           i2.left() <= i1.right();
}
```

Implementer's code: Option A

```
public class TwoPtInterval
    implements Interval {
  /** Left endpoint. */
  private double left;
  /** Right endpoint. */
  private double right;

  @Override
  double left() {
    return left;
  }

  @Override
  double right() {
    return right;
  }

  @Override
  boolean contains(
      double x) {
    return left <= x &&
           x <= right;
  }
}
```

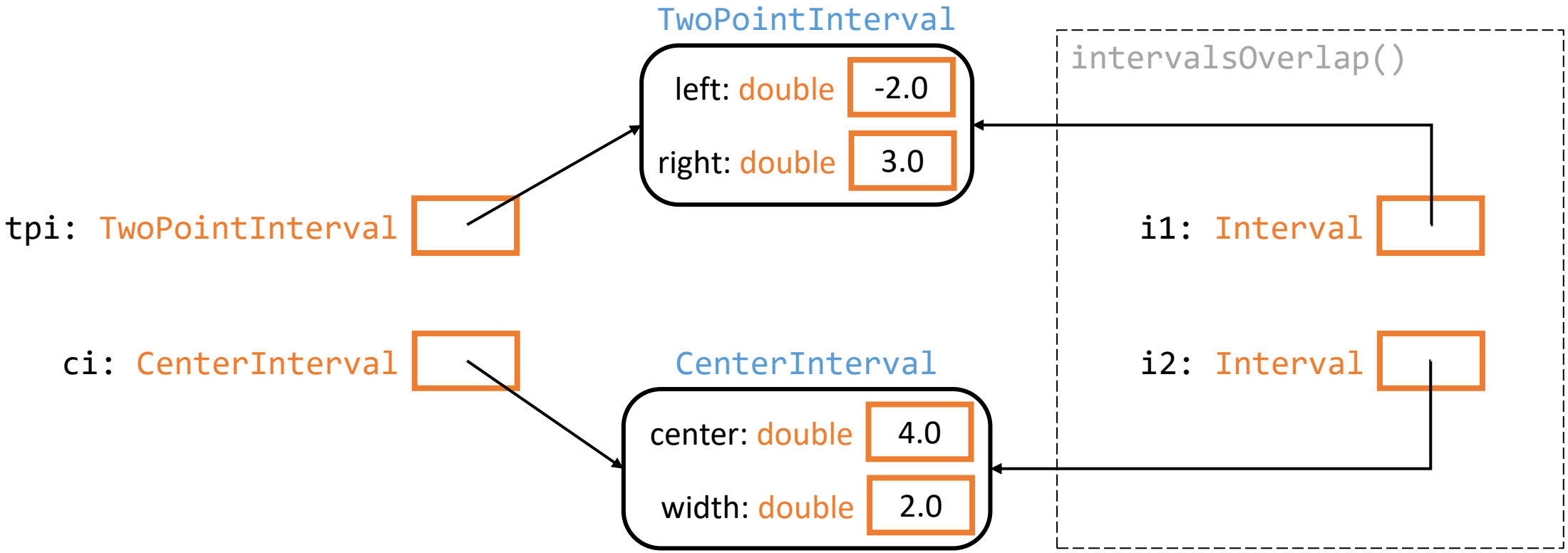
Implementer's code: Option B

```
public class CenterInterval
    implements Interval {
  /** Midpoint. */
  private double center;
  /** Width. */
  private double width;

  @Override
  double left() {
    return center - width/2;
  }

  @Override
  double right() {
    return center + width/2;
  }

  @Override
  boolean contains(
      double x) {
    return abs(x - center)
           <= width/2;
  }
}
```

Client code 2

```
TwoPtInterval tpi = new TwoPtInterval(-2, 3);
CenterInterval ci = new CenterInterval(4, 2);

// Is this allowed?
boolean overlap = intervalsOverlap(tpi, ci);
```

Object diagram



Subtypes

- A `TwoPtInterval` can do anything an `Interval` can do
  - Behavior bound by same specifications
- Therefore, a `TwoPtInterval` can be used anywhere an `Interval` is expected

- Implementing an `interface` establishes a **subtype** relationship
  - `TwoPtInterval <: Interval`
  - `CenterInterval <: Interval`

Subtype compatibility

- Assignment
  - `T x = expr;` is allowed if the type of *expr* is a subtype of `T`
- Argument passing
  - `void foo(T x);`
    `foo(expr)`
    is allowed if the type of *expr* is a subtype of `T`
- Returning
  - `T bar() { return expr; }` is allowed if the type of *expr* is a subtype of `T`

- "If x can store an animal, it can store a cat."
- "If foo() expects a bird, it can work with a robin."
- "If bar() says it will return an insect, it's allowed to give me a cricket."

Quick check

Suppose `Foo <: Bar`.

Consider these method declarations:

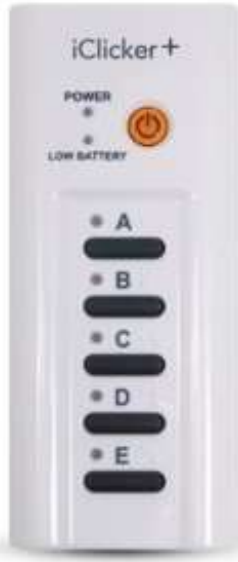
- `Foo f();`
- `Bar g();`

Which of these is allowed?

A. `Foo f = g();`

B. `Bar b = f();`

C. Neither



(subtype) Polymorphism

Variations in behavior

- The Interval interface abstracted over state, but both implementations behaved identically
- Sometimes, behavior specifications leave room for variation
- Example: chess pieces

Chess piece interface

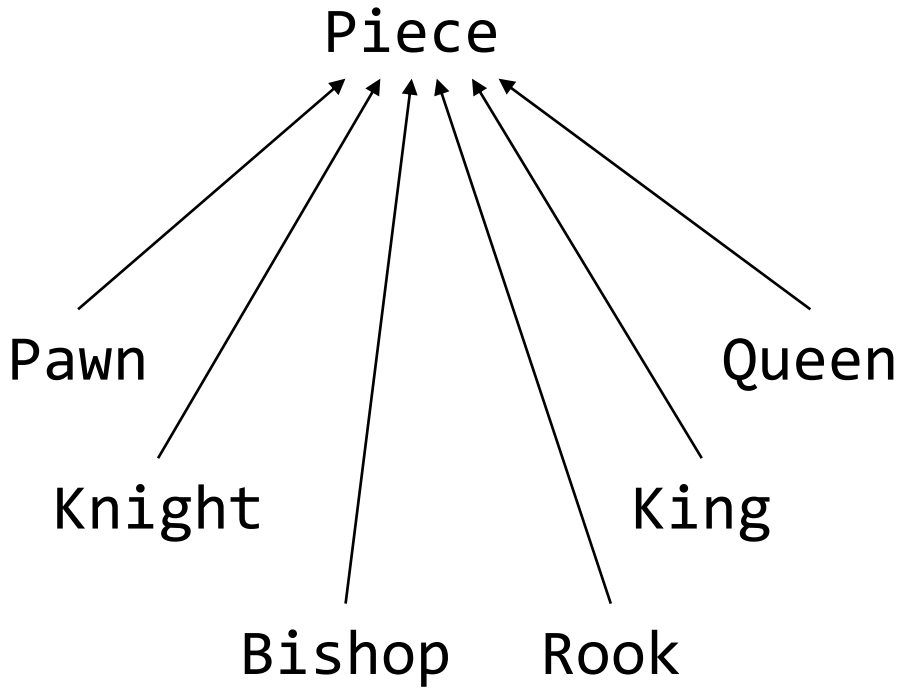
```
public interface Piece {
  /** Return whether this piece is able to move to
   *  location (`dstRow`, `dstCol`) from its current
   *  position, given board config. `board`.
   *  Requires dstRow, dstCol in [0..7]. */
  boolean legalMove(int dstRow, int dstCol,
                    Board board);
}
```

Chess board interface

```
public interface Board {
  /** Return 0 if position (`row`, `col`) is empty,
   *  1 if occupied by a white piece, 2 if occupied
   *  by a black piece.  Requires row, col in
   *  [0..7]. */
  int playerAt(int row, int col);
}
```

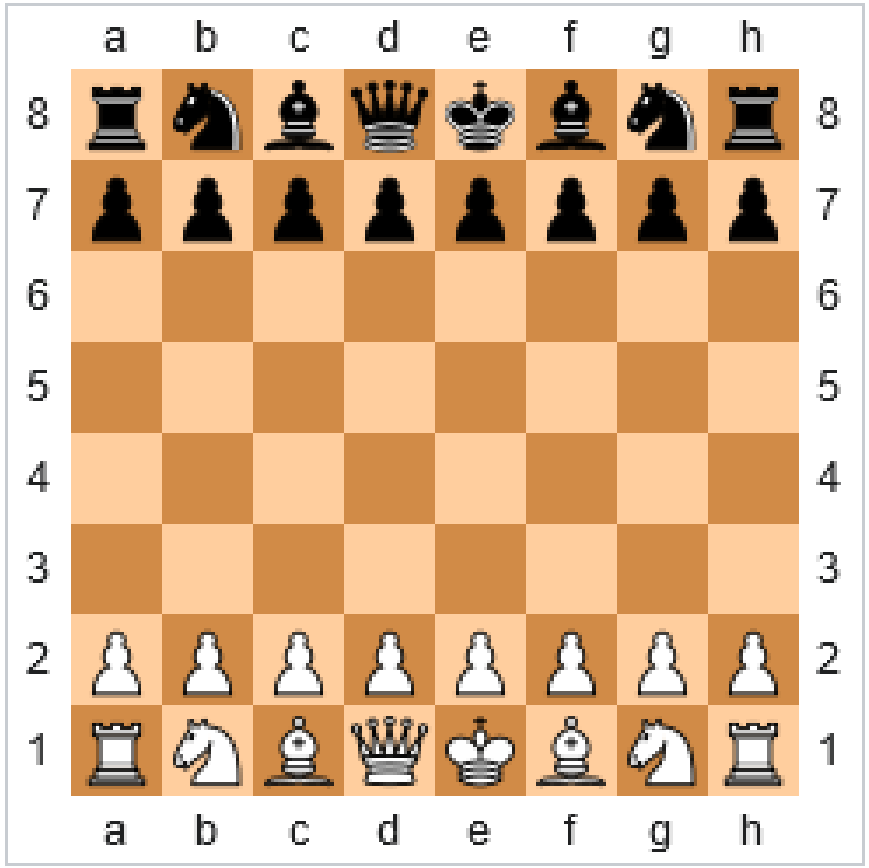
Type hierarchy

- `Pawn   <: Piece`
- `Knight <: Piece`
- `Bishop <: Piece`
- `Rook   <: Piece`
- `Queen  <: Piece`
- `King   <: Piece`



Knight

```
public class Knight
    implements Piece {
  private int row;
  private int col;
  private int player;
  @Override
  public boolean legalMove(
      int dstRow,
      int dstCol,
      Board board) {
    int dx = abs(row-dstRow);
    int dy = abs(col-dstCol);
    return board.playerAt(
        dstRow, dstCol)!=player
        && ((dx==1 && dy==2) ||
            (dx==2 && dy==1));
  }}
```

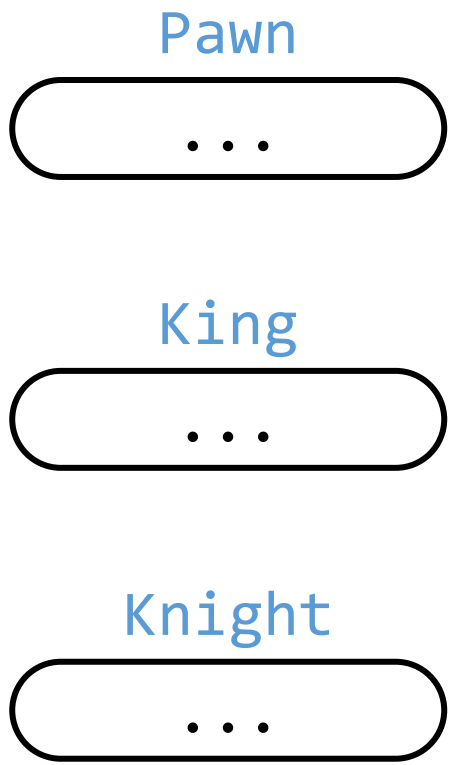

a b c d e f g h
8 7 6 5 4 3 2 1

King

```
public class King
    implements Piece {
  private int row;
  private int col;
  private int player;
  private boolean hasMoved;
  @Override
  public boolean legalMove(
      int dstRow
      int dstCol,
      Board board) {
int dx = abs(row-dstRow);
int dy = abs(col-dstCol);
return board.playerAt(
    dstRow, dstCol)!=player
    && (dx <= 1 && dy <= 1
     || !hasMoved &&
        canCastle(board));
}
public boolean canCastle(
    Board board) { …
}}
```

Object diagram

```
Piece pickNextPiece() {…}
// ...
Piece p;
while (!gameOver) {
   p = pickNextPiece();
   // assign r, c
   if (p.legalMove(r, c)) {
      // ...
   }
}
```



Static vs. dynamic type

- While the program is running, the type of the object referenced by `p` could change, but it will always be a subtype of `Piece`

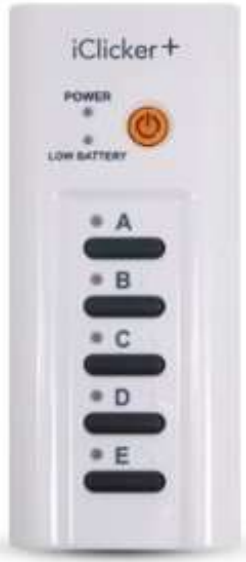
- Static type: types declared for variables & return values, derived for expressions (compile-time)
- Dynamic type: the type of an object being referenced (runtime)

- Behavior is determined by *dynamic* type
  - "**Dynamic dispatch**"

Poll

Should a client be able to call `p.canCastle()` when the *dynamic type* of the object referenced by `` `Piece p` `` is a King?

A. Yes
B. No
C. Only if they know more than the compiler



Compile-time reference rule

- Client can only request behavior supported by the target's static type
  - Guarantees that requested method will exist (unless target is null)
  - Compiler does not reason about dynamic types, even if "obvious"
    - `Piece p = new King(…);  // Static type of p is Piece`

Most important rule in the course!

Factories

```
/** Create a Piece of the
  * type specified by `code`
  * (algebraic notation). */
static Piece makePiece(
    char code) {
  if (code == 'K') {
    return new King(…);
  else if (code == 'N') {
    return new Knight(…);
  } else // …
}
```

Why declare with less-specific static types?

- Might not know which constructor will be called!

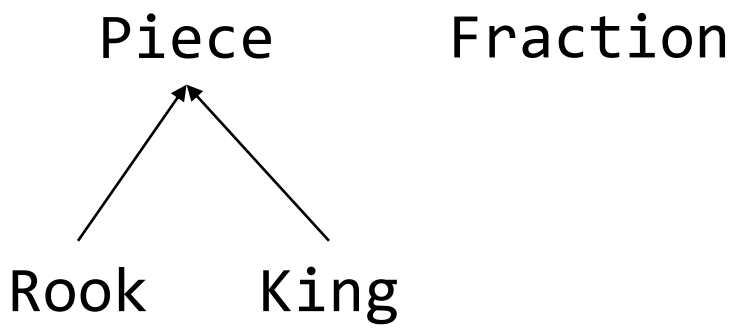
```
Piece p1 = makePiece('K');
// Static type of p1?
// Dynamic type of p1?

char code = in.readChar();
Piece p2 = makePiece(code);
```

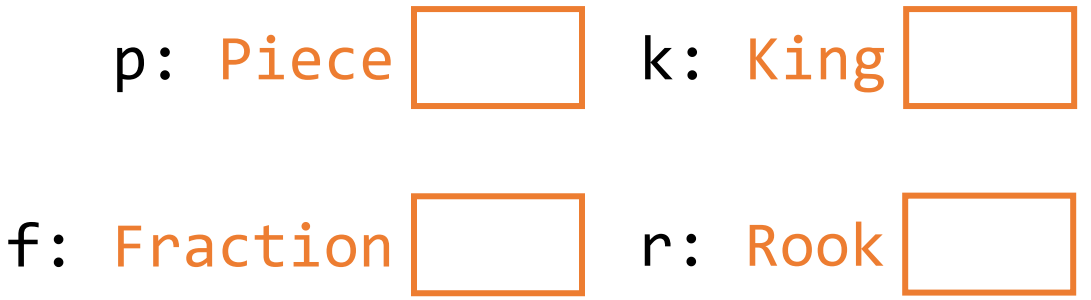
Casting between reference types

- **Dynamic dispatch** means an object will behave according to its dynamic type at runtime
- What if you think you know an object's dynamic type more specifically than its static type?
  - Can **cast** an expression to recover access to additional behavior
    - `King k = (King) p;`
- Casting is *checked* at runtime, but otherwise has no effect on the object
  - Just used to massage static types
  - If you're wrong, will get a runtime error (`ClassCastException`)

Casting examples

- `k = (King) p;`
  - Allowed; could fail at runtime
- `p = (Piece) k;`
  - Allowed but unnecessary; will always succeed
  - `p = k;` allowed by subtype substitution rule
- `k = (King) r; k = (King)f;`
  - Not allowed (impossible)
- `k = (King)(Piece) r;`
  - Allowed; will fail at runtime





Checking dynamic type before casting

- `instanceof` operator
  - `` `expr instanceof T` `` is true if the dynamic type of *expr* is a subtype of T

```
if (p instanceof King) {
  King k = (King) p;
  if (k.canCastle()) {…}
}
```

- Runtime type queries are useful in *some* circumstances, but are usually a sign of poor OO design
- When possible, specify common behavior in a supertype, then leverage dynamic dispatch for polymorphism

Checkpoint

- Compile-time errors
  - Red underlines in IntelliJ
  - Red icons in “Problems” panel
  - Compilation errors from smoketester
- Runtime errors
  - Exception backtrace when program is run
- Spec violations are *bugs* but might not prevent compilation or produce an exception
  - Defensive programming: turn spec violations into runtime errors

Any problems?

A: Compile-time error
B: Runtime error
C: Spec violated
D: Okay

Implementer

```
public class DynArray {
  /** Double capacity */
  private void incCap() {
    ...
  }
}
```

Client

```
DynArray a =
     new DynArray();
a.incCap();
```

Any problems?

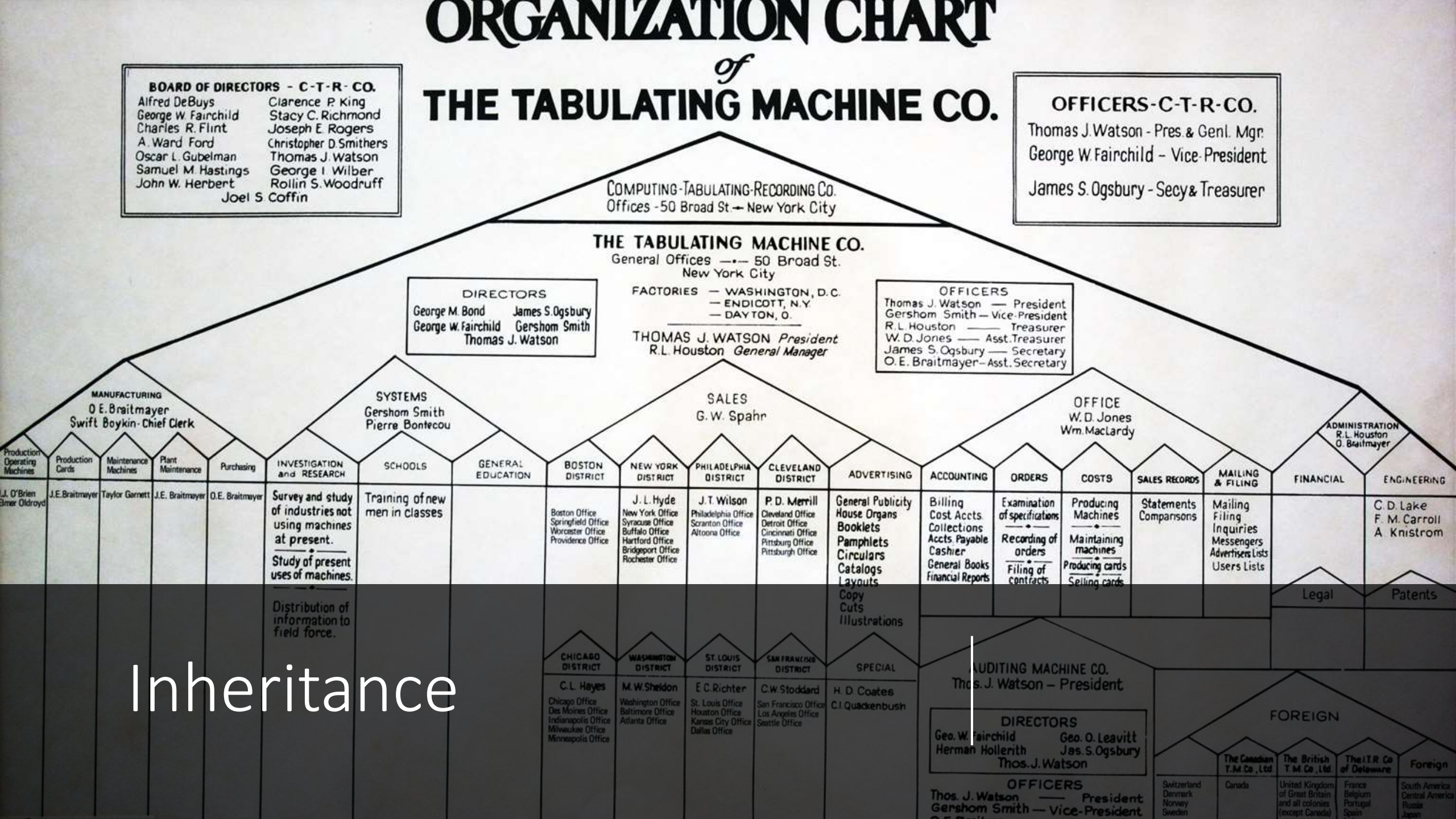
A: Compile-time error
B: Runtime error
C: Spec violated
D: Okay

Implementer

```
interface Phone {
  void call(int[] num);
}
class IPhone
    implements Phone {
  public void call(
    int[] num) {…}
  public void takePic() {…}
}
```

Client

```
Phone p;
p = new IPhone();
p.takePic();
```



Inheritance