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);</pre>
```

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);</pre>
```



CS 2110 Lecture 5

Interfaces, subtyping, polymorphism



A2 should be released today

Coming up

A1 is being graded now

Expect feedback Thurs

iClickers

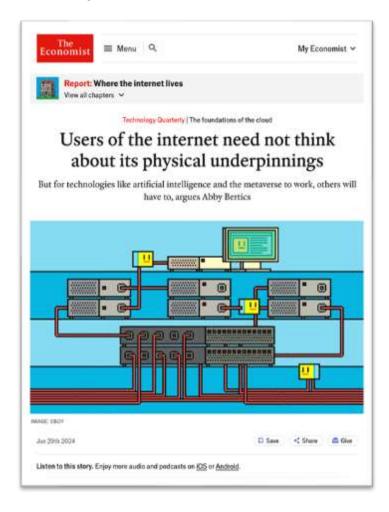


Interfaces

"Interfaces"

- Mechanism by which two parties work together, decided ahead of time
 - HDMI cable: interface between laptop and projector
 - 1/4"-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?

2.

J.

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

Implementer's code: Option A

```
@Override
public class TwoPtInterval
    implements Interval {
                                      double right() {
  /** Left endpoint. */
                                         return right;
  private double left;
  /** Right endpoint. */
  private double right;
                                      @Override
                                      boolean contains(
  @Override
                                           double x) {
  double left() {
                                         return left <= x &&
                                                    x <= right;
    return left;
```

Implementer's code: Option B

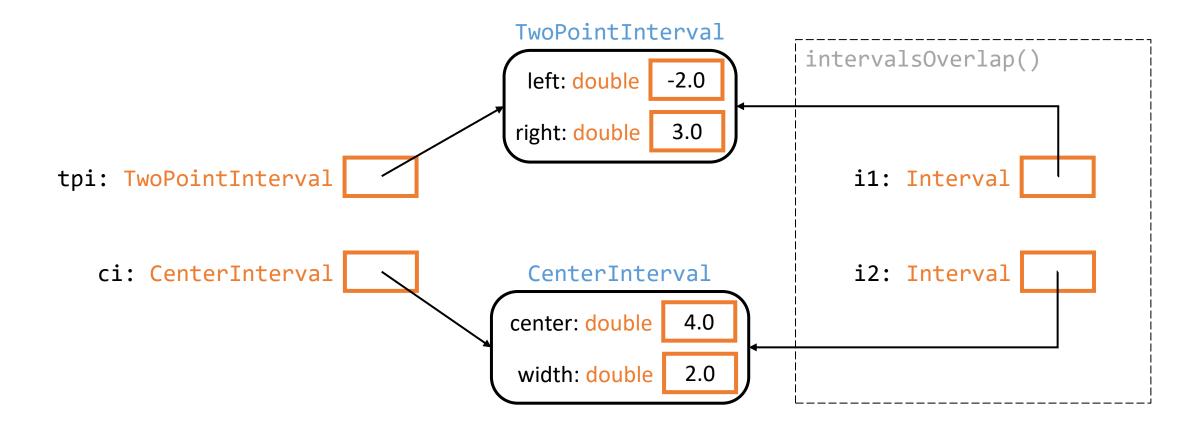
```
@Override
public class CenterInterval
    implements Interval {
                                      double right() {
  /** Midpoint. */
                                        return center + width/2;
  private double center;
  /** Width. */
  private double width;
                                      @Override
                                      boolean contains(
  @Override
                                           double x) {
                                        return abs(x - center)
  double left() {
                                                 <= width/2;
    return 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
$$b = f()$$
;

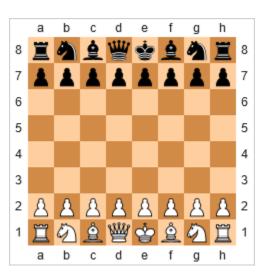
C. Neither





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

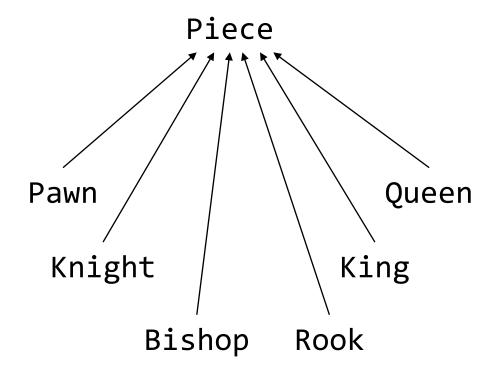
```
public interface Piece {
  /** Return whether this piece is able to move to
     location (`dstRow`, `dstCol`) from its current
      position, given board config. `board`.
     Requires dstRow, dstCol in [0..7]. */
  boolean legalMove(int dstRow, int dstCol,
                    Board board);
```

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);
}
```

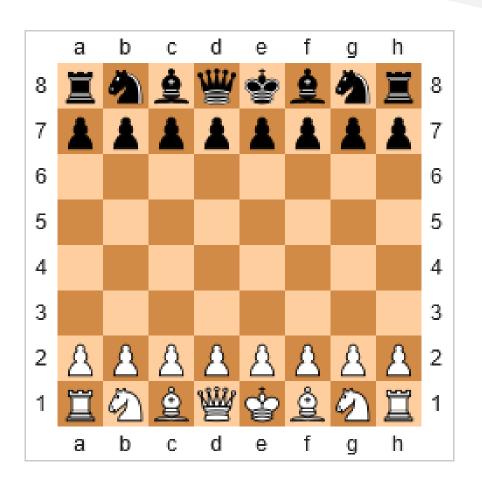
Type hierarchy

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



Knight

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



King

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

Object diagram

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

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
                                   Why declare with less-specific static
  * type specified by `code`
                                   types?
  * (algebraic notation). */

    Might not know which constructor will

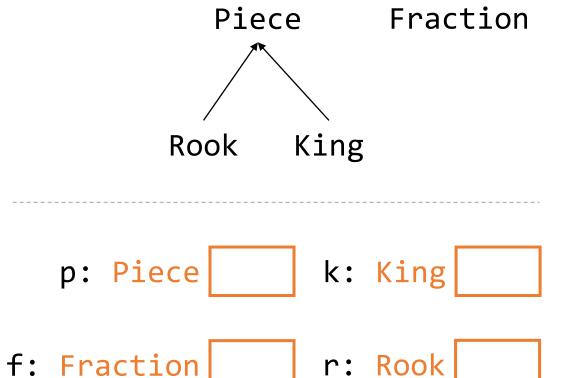
static Piece makePiece(
                                     be called!
    char code) {
  if (code == 'K') {
                                   Piece p1 = makePiece('K');
    return new King(...);
                                   // Static type of p1?
  else if (code == 'N') {
                                   // Dynamic type of p1?
    return new Knight(...);
  } else // ...
                                   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
- Runtime type queries are useful in some circumstances, but are usually a sign of poor OO design

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

 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();
```



ORGANIZATION CHART THE TABULATING MACHINE CO. BOARD OF DIRECTORS - C-T-R-CO. OFFICERS-C-T-R-CO. Alfred DeBuys Clarence P. King George W. Fairchild Stacy C. Richmond Thomas J. Watson - Pres. & Genl. Mgr. Charles R. Flint Joseph E. Rogers A. Ward Ford Christopher D. Smithers George W. Fairchild - Vice-President Oscar L. Gubelman Thomas J. Watson Samuel M. Hastings George I Wilber Rollin S. Woodruff John W. Herbert COMPUTING-TABULATING-RECORDING CO James S. Ogsbury - Secy & Treasurer Joel S Coffin Offices - 50 Broad St - New York City THE TABULATING MACHINE CO. General Offices --- 50 Broad St. New York City FACTORIES - WASHINGTON, D.C. OFFICERS DIRECTORS - ENDICOTT, N.Y. Thomas J. Watson - President George M. Bond James S. Ogsbury - DAYTON, O. Gershom Smith - Vice President George W. Fairchild Gershom Smith R.L. Houston _____ Treasurer THOMAS J. WATSON President W. D. Jones - Asst. Treasurer Thomas J. Watson R.L. Houston General Manager James S. Ogsbury - Secretary O. E. Braitmayer-Asst. Secretary SYSTEMS SALES OFFICE 0 E. Braitmayer Gershom Smith G. W. Spahr W.D. Jones Swift Boykin-Chief Clerk Pierre Bontecou DMINISTRATION Wm. MacLardy R.L. Houston O. Braitmayer Production Cards Maintenance Machines INVESTIGATION GENERAL BOSTON SCHOOLS NEW YORK Purchasing PHILADELPHIA CLEVELAND and RESEARCH MAILING & FILING ADVERTISING EDUCATION DISTRICT ACCOUNTING ORDERS DISTRICT DISTRICT DISTRICT COSTS SALES RECORDS FINANCIAL ENGINEERING J.E.Braitmayer Taylor Garnett J.E. Braitmayer O.E. Braitmaye Survey and study Training of new J. L. Hyde J.T. Wilson General Publicity P. D. Merril Billing Examination Statements Producing Mailing C.D. Lake of industries not men in classes New York Office Boston Office Philadelphia Office Cleveland Office House Organs Cost Accts of specifications Machines Comparisons Filing F. M. Carroll Springfield Office Syracuse Office Scranton Office Detroit Office using machines Booklets Collections Inquiries Worcester Office **Buffalo Office** Altoona Office Cincinnati Office A Knistrom at present. Accts. Payable Recording of Maintaining **Pamphlets** Messengers Hartford Office Pittsburg Office machines Bridgeport Office Rochester Office Cashier orders Pittsburgh Office Advertisers Lists Circulars Study of present Producing cards General Books Filing of Users Lists Catalogs uses of machines Financial Reports avouts **Patents** Distribution of information to field force. Illustrations ST. LOUIS AN FRANCIS Inheritance UDITING MACHINE CO. DISTRICT SPECIAL DISTRICT J. Watson – President. C.L. Hayes M. W. Sheldon E C.Richter C.W. Stoddard H. D. Coates San Francisco Off Los Angeles Offici C.I. Quackenbush FOREIGN DIRECTORS Geo. W. Geo. O. Leavitt Inneapolis Office Herman Hollerith Thos. J. Watson Thos. J. Watson - President Gershom Smith - Vice-President