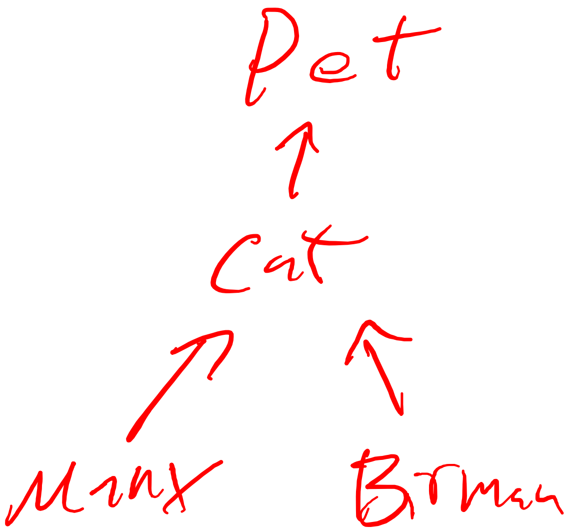
Warmup

Draw an **object diagram** depicting the state of the program's memory after executing the code to the right.

Assume the following subtype relationships:

- `Cat <: Pet`
- `Manx <: Cat`
- `Birman <: Cat`

```
Pet p = new Birman();
Cat c = (Cat)p;
p = new Manx();
Manx m = (Manx)p;
```



Warmup

What is the dynamic type of the object referenced by c?

```
Pet p = new Birman();
Cat c = (Cat)p;
p = new Manx();
Manx m = (Manx)p;
```

A. Pet

B. Cat

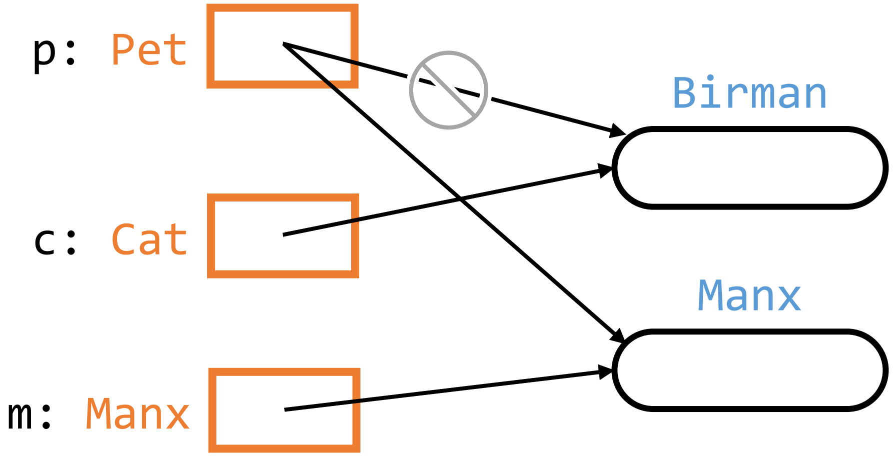
C. Birman

D. Manx

E. A ClassCastException would be thrown

Warmup (solution)

```
Pet p = new Birman();
Cat c = (Cat)p;
p = new Manx();
Manx m = (Manx)p;
```



CS 2110 Lecture 6

Inheritance, equality

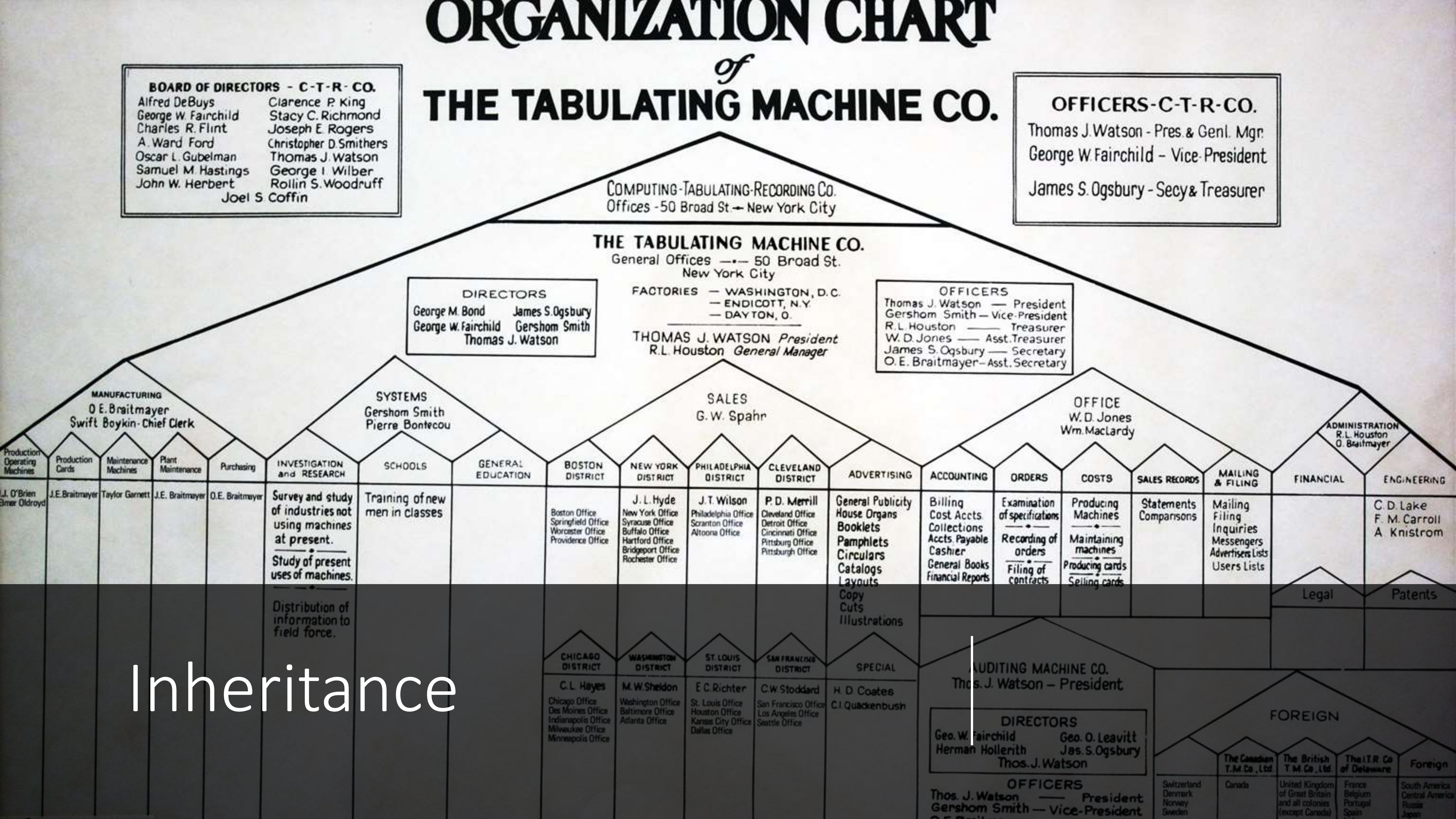
Coming up

A2 released

A1 grades incoming

Quiz 3

Inheritance



ORGANIZATION CHART of THE TABULATING MACHINE CO.

BOARD OF DIRECTORS - C-T-R- CO.

Alfred DeBuys
George W. Fairchild
Charles R. Flint
A. Ward Ford
Oscar L. Gubelman
Samuel M. Hastings
John W. Herbert
Clarence P. King
Stacy C. Richmond
Joseph E. Rogers
Christopher D. Smithers
Thomas J. Watson
George I. Wilber
Rollin S. Woodruff
Joel S. Coffin

OFFICERS-C-T-R-CO.

Thomas J. Watson - Pres. & Genl. Mgr.
George W. Fairchild - Vice-President
James S. Ogsbury - Secy & Treasurer

COMPUTING-TABULATING-RECORDING CO.
Offices - 50 Broad St. - New York City

THE TABULATING MACHINE CO.
General Offices — 50 Broad St.
New York City

FACTORIES — WASHINGTON, D.C.
— ENDICOTT, N.Y.
— DAYTON, O.

THOMAS J. WATSON *President*
R.L. Houston *General Manager*

DIRECTORS

George M. Bond
George W. Fairchild
James S. Ogsbury
Gershom Smith
Thomas J. Watson

OFFICERS

Thomas J. Watson — President
Gershom Smith — Vice-President
R.L. Houston — Treasurer
W. D. Jones — Asst. Treasurer
James S. Ogsbury — Secretary
O. E. Braitmayer — Asst. Secretary

MANUFACTURING
O E. Braitmayer
Swift Boykin - Chief Clerk

Production Operating Machines: J. O'Brien, Elmer Oldroyd
Production Cards: J.E. Braitmayer
Maintenance Machines: Taylor Garnett
Plant Maintenance: J.E. Braitmayer
Purchasing: O.E. Braitmayer

SYSTEMS
Gershom Smith
Pierre Bontecou

INVESTIGATION and RESEARCH: Survey and study of industries not using machines at present. Study of present uses of machines. Distribution of information to field force.
SCHOOLS: Training of new men in classes
GENERAL EDUCATION

SALES
G. W. Spahr

BOSTON DISTRICT: Boston Office, Springfield Office, Worcester Office, Providence Office
NEW YORK DISTRICT: J. L. Hyde; New York Office, Syracuse Office, Buffalo Office, Hartford Office, Bridgeport Office, Rochester Office
PHILADELPHIA DISTRICT: J.T. Wilson; Philadelphia Office, Scranton Office, Altoona Office
CLEVELAND DISTRICT: P. D. Merrill; Cleveland Office, Detroit Office, Cincinnati Office, Pittsburg Office, Pittsburgh Office
ADVERTISING: General Publicity, House Organs, Booklets, Pamphlets, Circulars, Catalogs, Layouts, Copy, Cuts, Illustrations
CHICAGO DISTRICT: C.L. Hayes; Chicago Office, Des Moines Office, Indianapolis Office, Milwaukee Office, Minneapolis Office
WASHINGTON DISTRICT: M.W. Sheldon; Washington Office, Baltimore Office, Atlanta Office
ST. LOUIS DISTRICT: E C. Richter; St. Louis Office, Houston Office, Kansas City Office, Dallas Office
SAN FRANCISCO DISTRICT: C.W. Stoddard; San Francisco Office, Los Angeles Office, Seattle Office
SPECIAL: H. D. Coates, C.I. Quackenbush

OFFICE
W. D. Jones
Wm. MacLardy

ACCOUNTING: Billing, Cost Accts., Collections, Accts. Payable, Cashier, General Books, Financial Reports
ORDERS: Examination of specifications, Recording of orders, Filing of contracts
COSTS: Producing Machines, Maintaining machines, Producing cards, Selling cards
SALES RECORDS: Statements, Comparisons
MAILING & FILING: Mailing, Filing, Inquiries, Messengers, Advertisers Lists, Users Lists

ADMINISTRATION
R.L. Houston
O. Braitmayer

FINANCIAL
ENGINEERING: C. D. Lake, F. M. Carroll, A. Knistrom
Legal
Patents

AUDITING MACHINE CO.
Thos. J. Watson — President

DIRECTORS

Geo. W. Fairchild
Herman Hollerith
Geo. O. Leavitt
Jas. S. Ogsbury
Thos. J. Watson

OFFICERS

Thos. J. Watson — President
Gershom Smith — Vice-President

FOREIGN

Switzerland, Denmark, Norway, Sweden
The Canadian T.M. Co., Ltd.: Canada
The British T.M. Co., Ltd.: United Kingdom of Great Britain and all colonies (except Canada)
The I.T.R. Co of Delaware: France, Belgium, Portugal, Spain
Foreign: South America, Central America, Russia, Japan

Chess pieces revisited

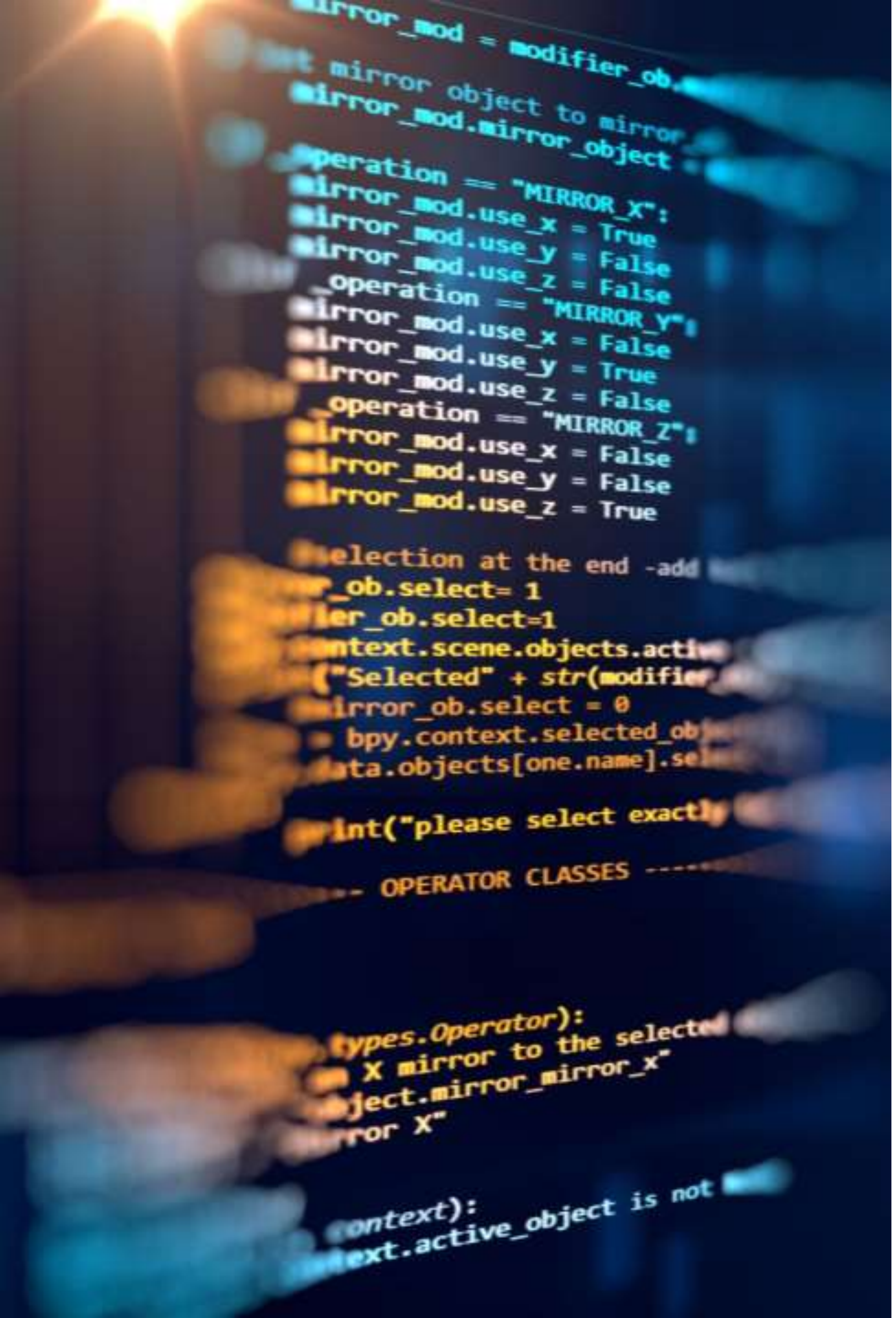
```
public class Knight
    implements Piece {
  private int row;
  private int col;
  private int player;
  public int player() {
    return player;
  }
  // ...
}
```

```
public class King
    implements Piece {
  private int row;
  private int col;
  private int player;
  private boolean hasMoved;
  public int player() {
    return player;
  }
  // ...
}
```

Chess pieces revisited

```java
public class Knight
    implements Piece {
  private int row;
  private int col;
  private int player;
  public int player() {
    return player;
  }
  // ...
}
```

```java
public class King
    implements Piece {
  private int row;
  private int col;
  private int player;
  private boolean hasMoved;
  public int player() {
    return player;
  }
  // ...
}
```



DRY principle: Don’t repeat yourself

- Duplicated code is not just tedious to write (or copy-paste) the first time
    - To fix a bug in duplicated code, must find all instances
    - Modifications that aren’t repeated everywhere lead to deviation in “common” behavior
- OOP languages can help you avoid duplication

Commonality beyond interfaces

- Interfaces guarantee *availability* of behaviors
- What if types have similar state? Identical behaviors?
  - Interfaces can't provide fields or method bodies that depend on fields
  - (they can provide method bodies that depend only on other methods)
- **Subclasses** allow a *derived class* to **inherit** fields and method bodies from a *parent class*
  - `class Derived extends Parent {…}`
  - Implies a *subtype* relationship: Derived <: Parent

Piece as a *superclass*

```java
public class Piece {
  private int row;
  private int col;
  private int player;

  public Piece(int row,
      int col, int player) {
    this.row = row;
    this.col = col;
    this.player = player;
}
```

```java
  public int player() {
    return player;
  }

  public boolean legalMove(
      int dstRow, int dstCol,
      Board board) { ??? }
}
```

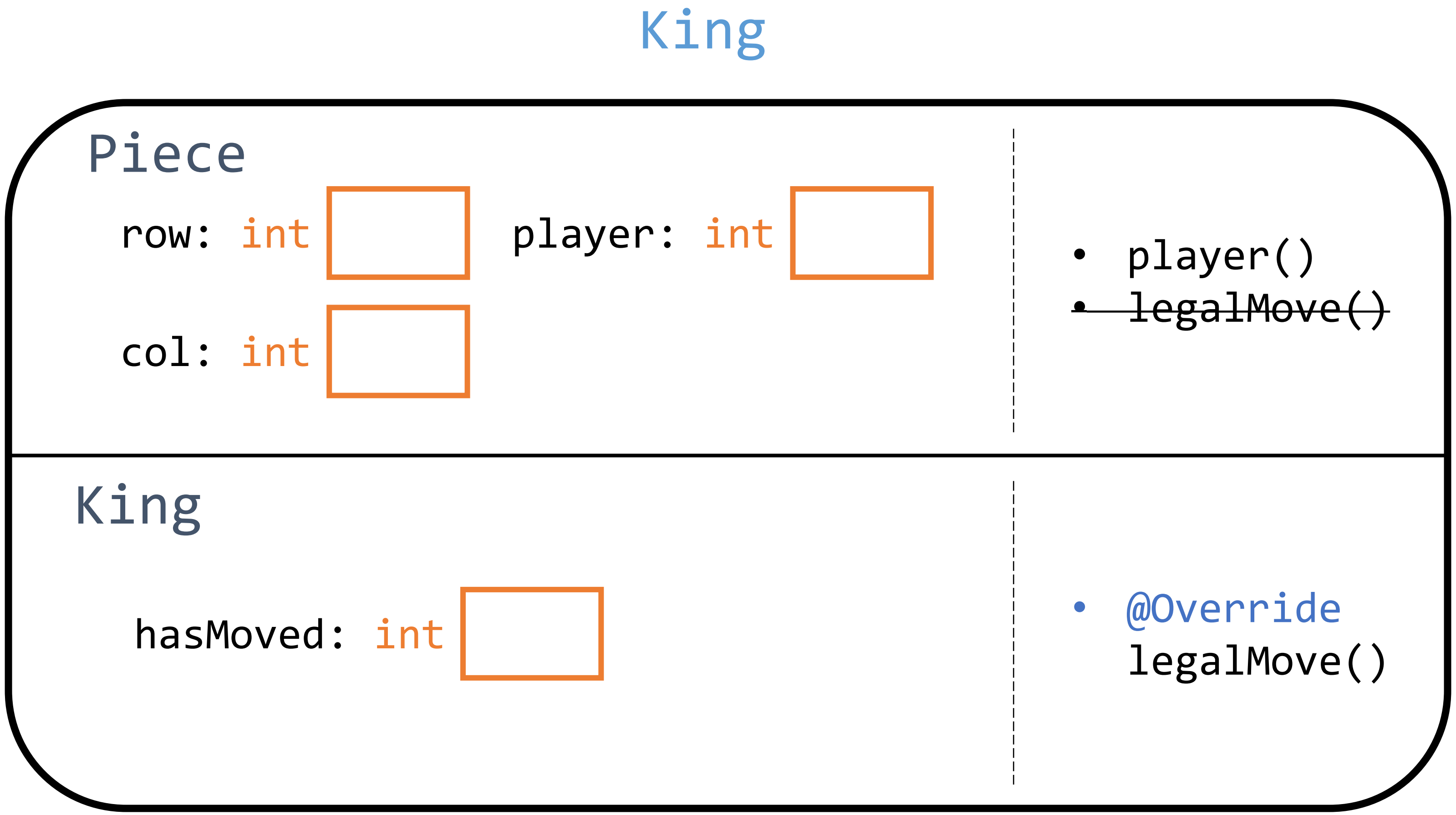
King as a subclass

```
public class King
    extends Piece {
 private boolean hasMoved;


 public King(int player) {
   super((player==1)?0:7,
         3, player);
   hasMoved = false;
 }
```

```
 @Override
  public boolean legalMove(
      int dstRow
      int dstCol,
      Board board) {…}
}
```

Object diagram showing inheritance



King

Piece

row: int

player: int

col: int

- player()
- ~~legalMove()~~

King

hasMoved: int

- @Override legalMove()

Parent section (above)

Derived section (below)

Accessibility

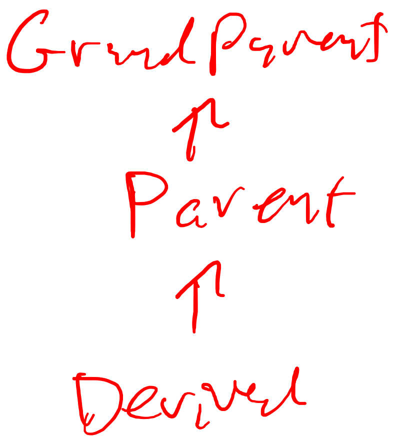
- Subclasses cannot see private members of parent class
  - Is this a concern?
- "Specialization interface": in what ways can subclasses tweak the behavior of a parent?
  - Another layer of **encapsulation**

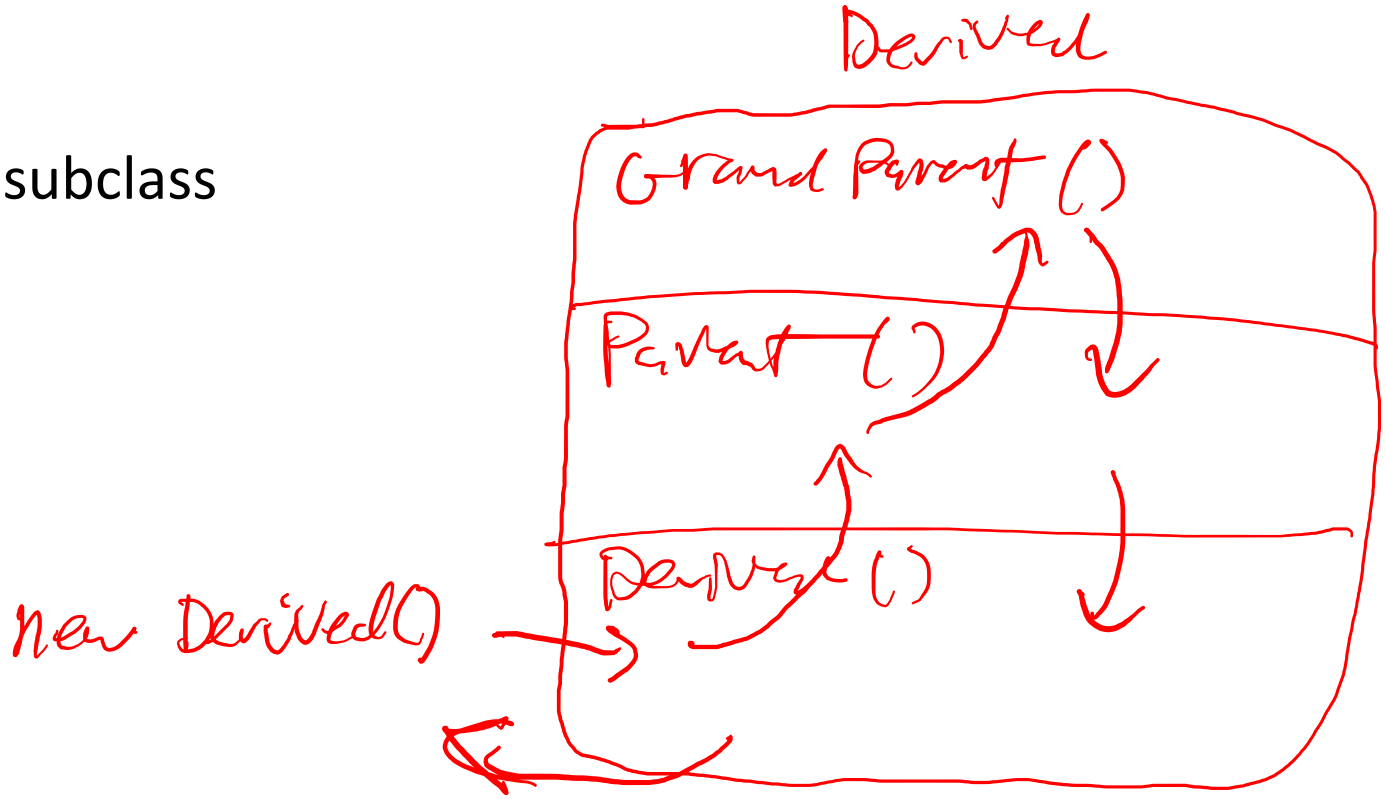
- `private` ("don't mess with my invariants")
  - Parent class has exclusive responsibility
- `protected` ("I'm trusting you")
  - Derived classes have rights and responsibilities
- `public`
  - The "client interface" is also usable by derived classes

Constructors

- Since some state *could* be private, subclass *must* call a parent class constructor
  - Invoked using `super()`
  - Must be *first* statement in subclass constructor

- Delegation order: fully construct superclass, then specialize



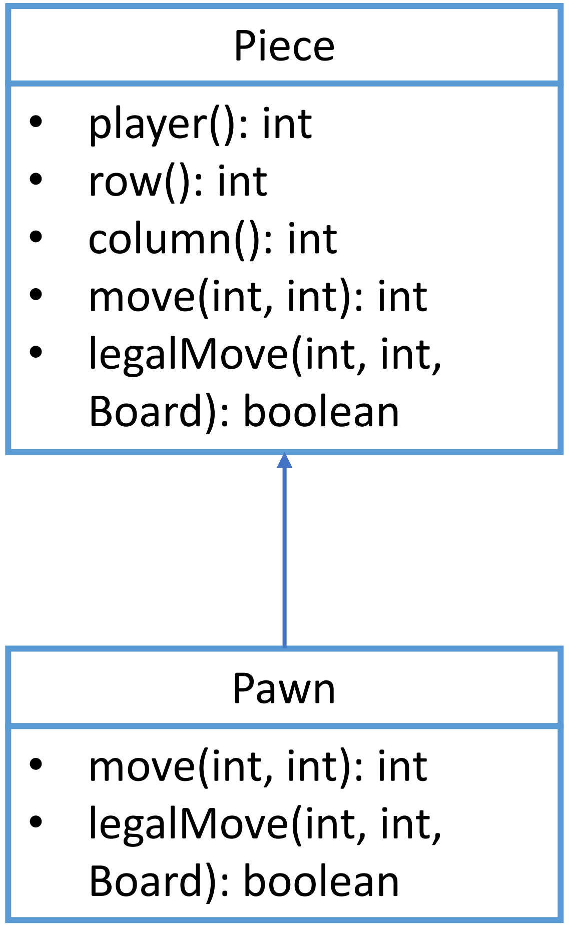


Overriding

- A subclass method with the same signature as a parent class method will **override** it
  - Whenever that method is invoked on the object, the *subclass* version will be executed
  - Consequence of **dynamic dispatch**

```
class Piece {
  boolean legalMove(
    int r, int c, Board b) {
    // IDK?
  }
}
class Pawn {
  @Override
  boolean legalMove(
    int r, int c, Board b) {
    // Pawn logic
  }
}
```

Dynamic dispatch illustrated



Piece

- player(): int
- row(): int
- column(): int
- move(int, int): int
- legalMove(int, int, Board): boolean

Pawn

- move(int, int): int
- legalMove(int, int, Board): boolean

```
Piece p = selectPiece();
// user selects a Pawn
p.move(newRow, newCol);
// which class’s move()
// code is run?
```

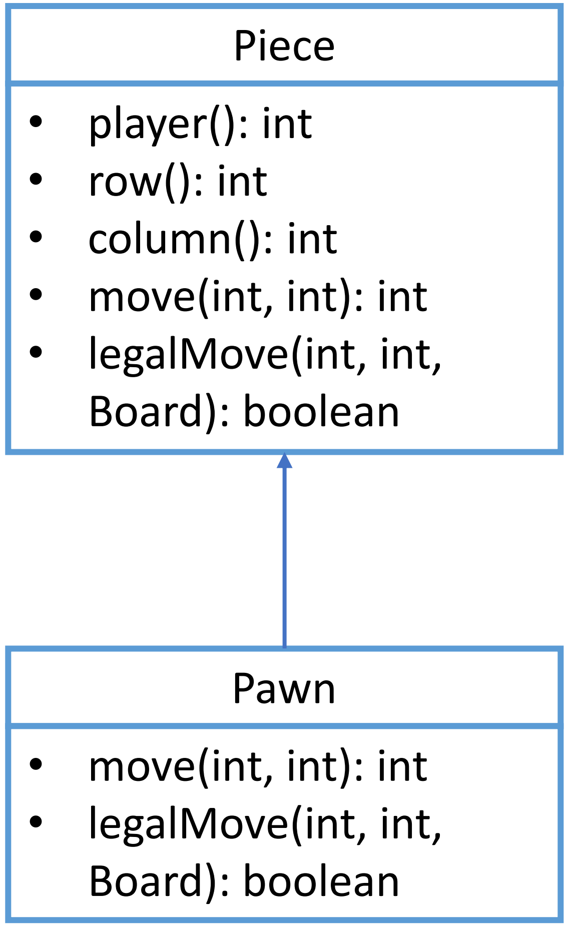
Overriding

- A subclass method with the same signature as a parent class method will **override** it
  - Whenever that method is invoked on the object, the *subclass* version will be executed
  - Consequence of **dynamic dispatch**
- Impossible for *client* to request a parent implementation
  - Only subclass impl could know about all the relevant invariants

- Subclass may delegate to its parent’s implementation

  -
    ```
    @Override
    public void move(int r,
        int c) {
      super.move(r, c);
      checkPromotion();
    }
    ```
- No way to prefer “grandparent’s” implementation

Bottom-up rule



Piece

- player(): int
- row(): int
- column(): int
- move(int, int): int
- legalMove(int, int, Board): boolean

Pawn

- move(int, int): int
- legalMove(int, int, Board): boolean

```
class Piece {
  boolean legalMove(…) {}
  void move(…) {
    if (legalMove(…)) {}
  }
}
class Pawn {
  @Override boolean legalMove(…) {}
  @Override void move(…) {
    super.move(…);
    checkPromotion();
  }
}
```

Checkpoint

```
class Parent {
  int foo(int x) { /* Parent impl */ }
  int bar(int x) {
    return foo(2*x + 1);
  }
}
class Derived extends Parent {
  int foo(int x) { /* Derived impl */ }
}
class Child extends Parent {
  int bar(int x) {
    return super.foo(x - 1);
  }
}
```

Which blocks of code are executed when the following client code is run?

```
Parent p = new Derived();
int ans = p.bar(42);
```

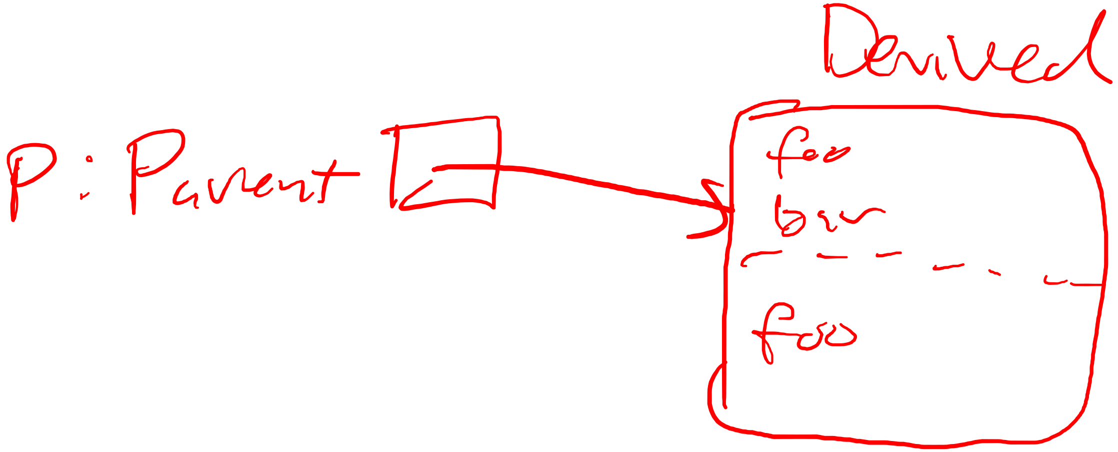
- A: `/* Parent impl */`
- B: `/* Derived impl */`
- C: Both Parent impl and Derived impl
- D: None of these
- E: Compile-time error

Checkpoint

```
class Parent {
  int foo(int x) { /* Parent impl */ }
  int bar(int x) {
    return foo(2*x + 1);
  }
}
class Derived extends Parent {
  int foo(int x) { /* Derived impl */ }
}
class Child extends Parent {
  int bar(int x) {
    return super.foo(x - 1);
  }
}
```

Which blocks of code are executed when the following client code is run?

```
Parent p = new Derived();
int ans = p.bar(42);
```



Abstract classes

- How should `Piece` implement `legalMove()` itself?
- Who should be allowed to construct a `Piece`?
    - Not a concern with `interface` – no method bodies, no constructor

- Abstract classes
    - Cannot be constructed on their own (must construct a subclass)
        - Even though they may define a constructor
    - May have abstract methods
        - Declarations only (like interfaces)
- Subclass must override all abstract methods, or be abstract itself

Piece as an abstract superclass

```
public abstract class Piece {
  protected int row;
  protected int col;
  private int player;

  protected Piece(int row,
      int col, int player) {
    this.row = row;
    this.col = col;
    this.player = player;
  }
```

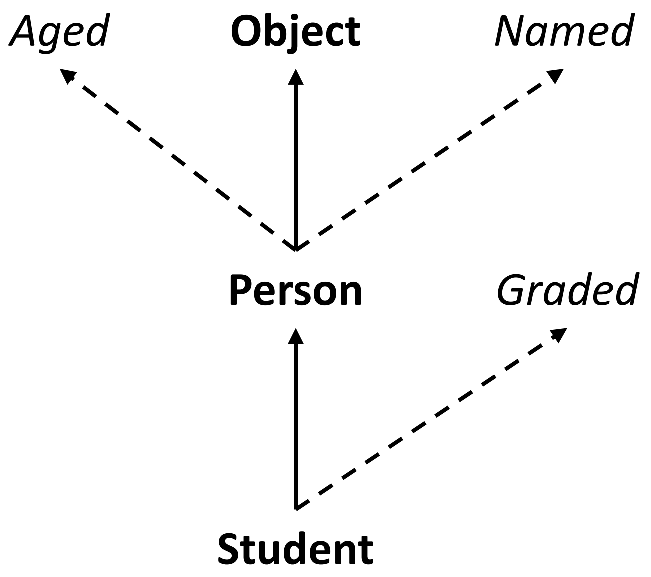
```
  public int player() {
    return player;
  }

  public abstract boolean
      legalMove(int dstRow,
      int dstCol, Board board);
}
```

Class `Object`

Relationships

- Java only supports *single inheritance*
  - Only one superclass
  - Reserve for “is-a” relationship
- Classes may implement multiple interfaces
  - “Can-do” relationship
- Interfaces may extend (multiple) other interfaces



Relationships

- Don’t forget about **composition**
  - Reference an instance of another class in a field
  - “Has a” relationship
  - Often most flexible, maintainable kind of relationship

Object

- All classes are a subtype of Object
  - If no `extends` clause, then `Object` is the superclass
  - Interfaces implicitly must be implemented by an `Object`

- Object provides useful universal methods that you may want to override
  - `toString()`
  - `equals()`
  - `hashCode()`

toString() example: Point

```
public class Point {
  private double x;
  private double y;

  @Override
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

Equality

Referential equality (identity)

- Are two objects the same object?
- Test using ==
- Best avoided in most client code
  - Let classes define their own equivalence relations

Logical equality (state)

- Should two objects be considered equivalent (substitutable)?
- Test using `equals()`
  - Defaults to referential equality
  - May override `equals()` to define value equality
    - Danger if class is mutable

Equivalence relations

- Reflexive
  - You equal yourself
- Symmetric
  - If you equal someone, they equal you
- Transitive
  - If you equal someone and they equal someone else, you also equal that someone else

Overriding .equals()

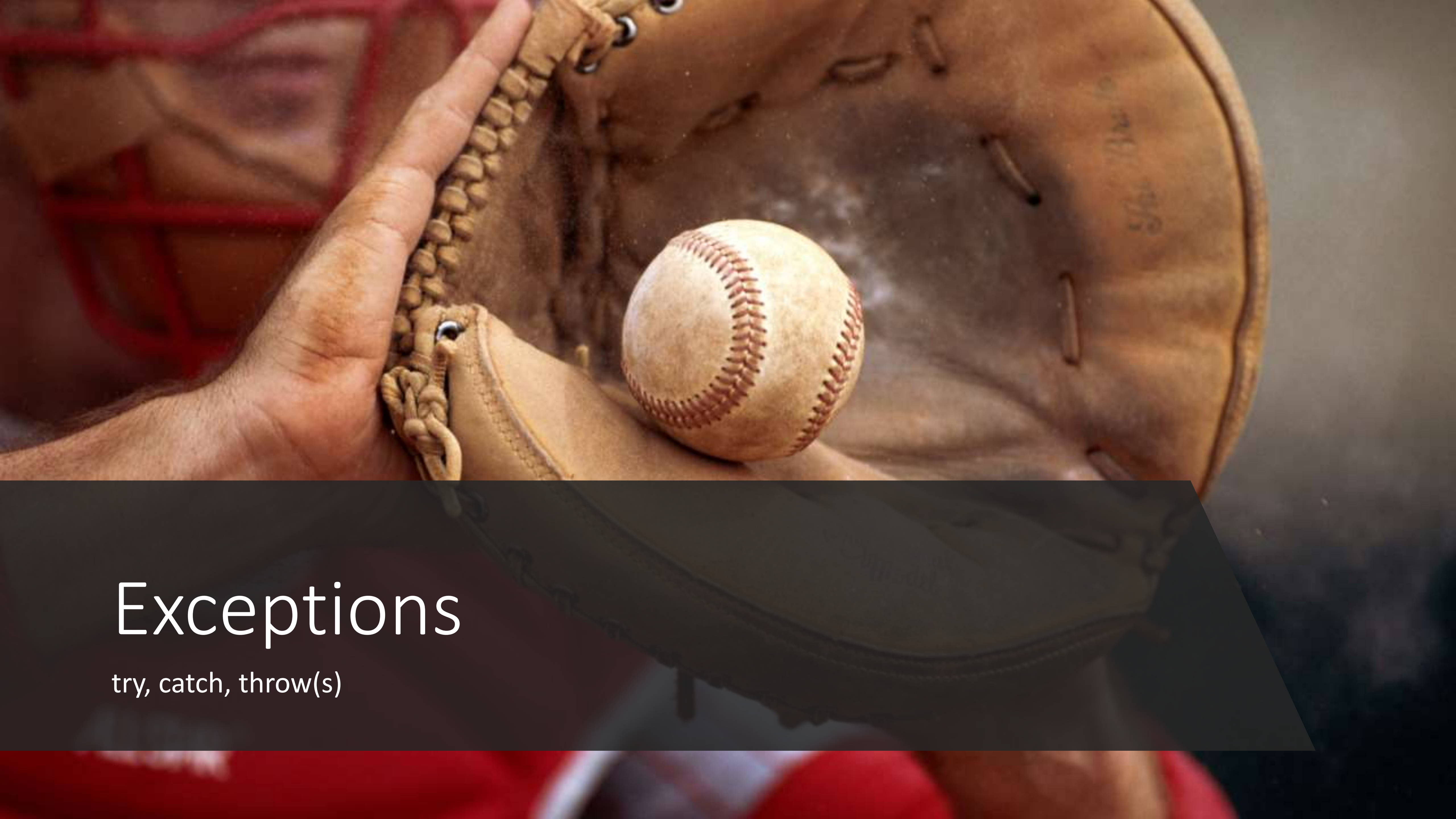
```
@Override
public boolean equals(Object other) {
    if (!(other instanceof Point)) {
        return false;
    }
    Point p = (Point) other;
    return x == p.x && y == p.y;
}
```

Warning: not **symmetric** if subclasses might be compared with superclasses

getClass()

- To look at the exact dynamic type of an object (rather than an upper bound), use getClass() (inherited from Object)
  - Reserve for special cases
- Stronger equals() template:

```
@Override
public boolean equals(Object obj) {
    if ((obj == null) || (getClass() != obj.getClass())) {
        return false;
    }
    MyClass objAsMC = (MyClass) obj;
    // Compare fields
}
```



Exceptions

try, catch, throw(s)