Interfaces, subtyping, polymorphism
A2 is Released

A1 will be graded soon

Test 1 is Tomorrow (I’ll end with what to prep for Test1)
A2 Logistics and Test1 Prep

- Please start A2, Also if you’re working alone and haven’t talked to me about working with a partner. Please find a partner ASAP
- Test1 Prep and Expectations
  - Exactly the same style of questions as the prelims in the previous years
  - Topics are obviously adapted to what we covered here last week
  - Format is hard to predict exactly
JUnit assertions != Java `assert` statements
  • `assertEquals()`
  • `assertTrue()` / `assertFalse()`

• Argument order: *expected*, then *actual*

• Floating-point is tricky (see comment in A1Test)
Terms So Far

- Java syntax specific
  - this
  - final
  - static
  - main
  - class
  - void
  - public/private/protected
  - Junit Test related terms
  - extends

- Concepts
  - Value/Reference Semantics
  - Primitive/Class Types
  - Specifications/Invariants
  - Constructors/Getters/Setters
  - Scope
  - Casting
  - Testing- Black Box/Glass Box
  - OOP - Inheritance and Polymorphism
Inheritance

- Inheritance in Java is the method to create a hierarchy between classes by inheriting from other classes.
- It is basically a method to establish relationships between classes.

```java
public class A {
    // A's fields
    // A's methods
}

public class B extends A {
    // B's fields + A's (public and protected fields)
    // B's methods and A's (public and protected fields)
}
```
Relationships

- Java only supports *single inheritance*
  - Only one superclass
  - Reserve for “is-a” relationship
- Classes may implement multiple interfaces
  - “Can-do” relationship
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
Consequences of this

- Avoid code reduplication
- Subtype Polymorphism, Interface Polymorphism
- Allows for the expression of variations in behaviour

  - Defining inheritance hierarchies is basically a modelling problem
Interface Polymorphism

Interfaces allow us to define polymorphism in a declarative way, unrelated to implementation.

What is an interface?

```java
public interface Box {
    2 implementations
    public void shift(int dx, int dy);
    2 implementations
    public float area();
    2 implementations
    public boolean isInsideBox();
}
```
What this looks like in Java

Interfaces are basically like contracts

```java
public class Box1 implements Box {

    /**
     * Location of the lower-left corner of this box (point with minimum x-coordinate and minimum
     * y-coordinate). Non-null.
     */
    private final Point lower;

    /**
     * Location of the upper-right corner of this box (point with maximum x-coordinate and maximum
     * y-coordinate). Non-null. Invariant: \{@code upper.x >= lower.x AND upper.y >= lower.y\}.
     */
    private final Point upper;

    public void shift(int dx, int dy) { /* ..... */ }
    public float area() { /* ..... */ }
    public boolean isInsideBox() { /* ..... */ }
}
```
What this looks like in Java

Here’s another way of doing the exact same thing

```java
import java.awt.Point;
import java.awt.geom.Rectangle2D;

public class Box2 implements Box {

    /**
     * Location of box's centroid. Non-null.
     */
    private final Point center;

    /**
     * Width of box (in coordinate system units). Finite and non-negative.
     */
    private final double width;

    /**
     * Height of box (in coordinate system units). Finite and non-negative.
     */
    private final double height;

    public void shift(int dx, int dy) { /* .... */ }
    public float ares() { /* .... */ }
    public boolean isInsideBox() { /* .... */ }
}
```
Polymorphism (SubTyping)
Variations in behavior

• The Interval interface abstracted over state, but both implementations behaved identically. We just saw an example of this.
• Sometimes, behavior specifications leave room for variation
• Example: chess pieces
Chess piece interface

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

```java
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. */
    int playerAt(int row, int col);
}
```
Type hierarchy

- Pawn <: Piece
- Knight <: Piece
- Bishop <: Piece
- Rook <: Piece
- Queen <: Piece
- King <: Piece
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));
    }
}
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));
    }
}
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”
Should client be able to call `p.canCastle()` when the dynamic type of the object referenced by Piece p is a King?

- Yes: 0%
- No: 0%
- Only if they know more than the compiler: 0%
Compile-time reference rule

• Client can only request behavior supported by the static type

• It is possible to ask about the dynamic type of an object and cast the reference so that additional behavior is available, but this is usually not good OOP practice
  • instanceof
  • Example next time: equals()
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

• **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;
    }

    public int player() {
        return player;
    }

    public boolean legalMove(int dstRow, int dstCol, Board board) {...}
}
```
King as a subclass

```java
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) {
        ...
    }
}
```
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**
- 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
OOP terms chart

- extends
- interface / implements
- @override
- public/private/protected
- super

- Interface
- Encapsulation
- Interface/Subtype
- Polymorphism
- Inheritance
- Compile time reference rule
- Dynamic dispatch
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()
Equality

Referential equality (identity)
• Are two objects the same object?
• Test using ==

Logical equality (state)
• Should two objects be considered equivalent (substitutable)?
• Override equals() to define separately from identity
• 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()

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