Inheritance

COMPI400 - Week 12
Uno Game

Consider the card game Uno:


There are 6 kinds of cards:

• number cards
• draw two
• skip
• reverse
• wild
• wild draw four
We have a **Game** class which defines the rules of the game:

- Players take turns playing a card
- A valid play must be the same number or symbol or a wild card
- If you cannot play, draw a card.
- First player with no cards wins.
Card interface

Each kind of card has different properties and effects so belongs in a different class.

However they share a common interface.

They all support the methods of:

• can I play this card now?
• play this card.
public interface Card {

    // get the colour
    public int getColour();

    // get the symbol
    public char getSymbol();

    // test if it can be played
    public boolean canPlayOn(Card card);

    // implement any effects
    public void play(Game g);
}

There are six classes that implement the Card interface:

- NumberCard
- DrawCard
- SkipCard
- ReverseCard
- WildCard
- WildDrawCard
Card classes
Looking at the card classes we notice a lot of common data and code, e.g.:

```java
private int myColour;

public boolean canPlayOn(Card c) {
    return c.getSymbol() == 'S'
    || myColour == c.getColour();
}
```
Abstraction

The design principles of abstraction and encapsulation prompt us to ask:

Is there a way to factor out this common data and code into a single, reusable chunk?

The techniques we've seen so far do not apply very well.

We need a new idea: inheritance.
Inheritance

A Java class can extend another class:

// based class
class Parent {
}

// derived class
class Child extends Parent {
}
Inheritance

A derived class (child) inherits:

- All the fields of its base class (parent)
- All the methods of its base class

However it cannot access the private fields or methods on its base.
Extending

An extended child class may:

• Add new fields
• Add new methods
• Override old methods on its parent.
public class Turtle {
    private Point myPos;
    public Point getPos() {
        return myPos;
    }
    public void move(int dist) {
        // move forward
    }
}
public class ColourTurtle extends Turtle {

    // add a field
    private Color myColour;

    // add a method
    public Color getColour() {
        return myColour;
    }
}

public class TurningTurtle
extends Turtle {

private double myTurnSpeed;

// override method
public void move(int dist) {
    // move forward
    // while turning
}
}

Example

Turtle
myPos
getPos()
mov()
Calling methods

When you call a method on a derived class Java searches up the *inheritance hierarchy* until it finds a class that implements it.
Example

ColouredTurtle ct =
    new ColouredTurtle();

tc.getColour();
    // on ColouredTurtle

tc.move(100);
    // on Turtle
Example

TurningTurtle tt =
    new TurningTurtle();

tt.getPos();
// on Turtle

tt.move(100);
// on TurningTurtle
A method on a subclass can use the keyword `super` to refer to its parent class.

```java
public class TurningTurtle extends Turtle {
    public void move(int dist) {
        // move forward
        super.move();
        // while turning
    }
}
```
public class TurningTurtle extends Turtle {

    public void move(int dist) {
        // call parent
        // to move forward
        super.move();
        // now turn...
        myAngle += turnSpeed;
    }
}
Constructors

When we constructor a derived class we must first construct its parent.

We use the notation:

```
super()
```

Or if the super-constructor has parameters:

```
super(value1, value2, ...)
```

The super-constructor must always come first.
Constructor

```java
public class Turtle {
    private Point myPos;
    public Turtle() {
        myPos = new Point(0,0);
    }
}
```
public class ColourTurtle {

    private Color myColour;

    public ColourTurtle(Color colour) {
        super(); // call parent's
        // constructor first

        myColour = colour;
    }
}
Abstract

Sometimes several classes are based on the same parent, but the parent is incomplete or does not make sense as a usable object on its own.

In these cases it is appropriate to make the parent class **abstract**.
public abstract class AbstractCard {

    public boolean canPlayOn(Card c) {
        return mySymbol == c.getSymbol() || myColour == c.getColour();
    }

    // method not implemented:
    abstract public void play(Game g);
}
Abstract classes

An abstract class cannot be instantiated. It only exists to provide a base for other classes:

```java
AbstractCard card =
    new AbstractCard('X', Card.COLOUR_BLUE);

// ERROR!
```
Advantages

The advantages of inheritance:

- abstraction: common code is chunked
- encapsulation: parent code is hidden from children
- extendability: extra features can be added to classes
- polymorphism: child classes all inherit the same interface