List of Slides

17 `Card` code
18 Pros
19 Cons
20 Cons continued
21 **Section 3:** Approach B:Exact abstract model
22 Have the classes `Suit` and `Face`
23 Private constructor
24 *Exactly* four suits
25 Suit ordering based on integers
26 `Suit` code
27 `Suit` code
28 `Suit` code
29 `Suit` code
30 Array of `Faces`
31 Static initialiser
32 `Face` code
33 `Face` code
34 `Face` code

Case study 2

A Pack of Cards
(leading to a game of Snap)

Section 1

Introduction

Introduction

- In this case study we shall implement a pack of playing cards.
- We shall do this 3 times, each time using a different approach.
- We shall also write a simple text-based test program.
- Then we shall go on to provide images for the cards, and develop a simple Human reaction enhancement aid, based on the popular game of Snap. :-)
- Run:  /opt/teaching/bin/Snap

There are many ways to crack an egg . . .

- . . . and also many ways to build a bridge.
- There are usually many solutions to a problem. Part of the engineering process is making decisions about which one to use. These are sometimes subjective.
- The 3 approaches we study here to implementing the pack of cards are:
  - A: The "I model everything as a number because I used to be a C hacker" approach.
  - B: The "I build an exact abstract model of the requirements because I know it is safer than hacking" approach.
  - C: The "I religiously follow the teachings of OO and exploit the power of inheritance whenever I can" approach.
- I hope my prejudice isn't showing too much. :-)

The basis for variation

- A playing card comprises a suit and a face.
- There are 4 suits: **Clubs**, **Diamonds**, **Hearts** and **Spades**.
- There are 13 Faces: **Ace**, **King**, **Queen**, **Jack**, **Ten**, **Nine**, **Eight**, **Seven**, **Six**, **Five**, **Four**, **Three** and **Two**.
- The variation in the 3 approaches is concerned with how we model these two small sets of values.

Section 2

Approach A:

Ex C hacker

Using integers

- You have seen lots of examples of integers being used to model a small set of values. For example:
  - `Direction` and types of `Cell` in the Snake program.
  - Positions of placement in a `BorderLayout`.
- It is in fact quite commonly used.
- It gets its inspiration from how one does similar things in the programming language **C** – there we can use `#defines` to achieve a similar result.
- Many Java programmers are also, or used to be, C programmers.

Card

- Essentially then, an instance of `Card` consists of a pair of integers, one for the suit and one for the face.
- The integer for the suit is in the range 1 to 4 – 1 representing the lowest value suit. (Any true C hacker would use 0 to 3 probably!).
- The integer for the face is in the range 2 to 14 – 14 representing Ace which is the highest value. (Some would here be tempted to use 0 to 12, and add/substract 2 all the time!)
- The class provides 8 `public static final` variables for convenience, such as `ACE` and `SPADES`.
- Observe the `compareTo()` method.

Card code

```
public class Card implements Comparable
{
  public static final int CLUBS    = 1;
  public static final int DIAMONDS = 2;
  public static final int HEARTS   = 3;
  public static final int SPADES   = 4;

  public static final int JACK     = 11;
  public static final int QUEEN    = 12;
  public static final int KING     = 13;
  public static final int ACE      = 14;
```

Card code

```
private final int suit;
private final int face;


public Card(int requiredSuit, int requiredFace)
{
  suit = requiredSuit;
  face = requiredFace;
} // Card
```

Card code

```
public int getSuit()
{
  return suit;
} // getSuit


public int getFace()
{
  return face;
} // getFace
```

Card code

```
// Why is the parameter type Object, rather than Card?
public boolean equals(Object other)
{
  if (other instanceof Card)
    return suit == ((Card) other).suit && face == ((Card) other).face;
  else
    return super.equals(other);
} // equals
```

Card code

```
// Why do we need to cast other to Card?
public int compareTo(Object other)
{
  Card otherCard = (Card) other;
  if (suit == otherCard.suit)
    return face - otherCard.face;
  else
    return suit - otherCard.suit;
} // compareTo
```

Card code

```
public String toString()
{
  return faceToString(face) + " of " + suitToString(suit);
} // toString
```

Card code

```
private String suitToString(int suit)
{
  switch (suit)
  {
    case CLUBS:    return "clubs";
    case DIAMONDS: return "diamonds";
    case HEARTS:   return "hearts";
    case SPADES:   return "spades";
    default:       return "error!";
  } // switch
} // suitToString
```

Card code

```
private String faceToString(int face)
{
  switch (face)
  {
    case 2:     return "two";
    case 3:     return "three";
    case 4:     return "four";
    case 5:     return "five";
    case 6:     return "six";
    case 7:     return "seven";
    case 8:     return "eight";
    case 9:     return "nine";
    case 10:    return "ten";
```

Card code

```
      case JACK:  return "jack";
      case QUEEN: return "queen";
      case KING:  return "king";
      case ACE:   return "ace";
      default:    return "error!";
    } // switch
  } // faceToString

} // class Card
```

Pros

- The most attractive feature of this approach is its simplicity – we did not need to make classes for the data types `Suit` and `Face`.
- Of course, this is why it is commonly used.

Cons

- The most repulsive feature of this approach is its dangerous over simplicity – we did not make classes for the data types `Suit` and `Face` and so we have no control over them.
- Every suit has an associated integer, but not vice versa. There is nothing to stop someone writing:

  ```
  Card card = new Card(5, 15);
  ```

  thus making a ‘cheat’ card which has a greater value than any in a real pack! We could tighten that up by making the constructor raise an exception, but we cannot stop people trying to do it (i.e. their code will still compile).

Cons continued

- We also cannot stop someone writing:

```
Card card1 = new Card(Card.CLUBS, 2);
Card card2 = new Card(Card.SPADES, Card.ACE);
int hasNoMeaning = card1.getSuit() * card2.getFace() - 17;
```

- That is almost certainly nonsense, but we have no way of stopping it being done. Because a suit is an integer, someone can do anything to it which can be done to any other integer.
- We would like to make the datatype safer to guard against stupidity like that.
- Afterall, a major motivation for having datatypes (and classes in Java) is to gain safety. (Compare this with having to dangerously use `int` in C for booleans because there is no `boolean` type, etc..)

Section 3

Approach B:
Exact abstract model

Have the classes `Suit` and `Face`

- We can gain control over the suits and faces if we make classes for them.
- At the heart of these classes we will use an integer so that we get an ordering of the values. But this integer will be private, so no-one can abuse it.
- Of course, the approach we use here is not as straightforward as the previous – there is no gain without some pain. Do not let that put you off: in many situations the extra cost is worth it, and it is certainly important that you study the approach so that you are able to choose freely.

Private constructor

- Most (all?) the constructor methods you have seen up to now have had `public` visibility.
- This has enabled instances of the classes to be made from code in other classes.
- However, here we want there to be *exactly* four suits, each being an instance of the class `Suit`. We cannot guarantee this if code in other classes can make more instances.
- So, we simply make the constructor `private`!

Exactly four suits

- We make the four instances of `Suit` within the class and store references to them in `public static final` variables.

Suit ordering based on integers

- When we construct a suit, we give it an **ordinal number** so that suits can be compared to see which is the greatest.
- This integer is stored in a `private` instance variable.

Suit code

```
public class Suit implements Comparable
{
  public static final Suit CLUBS    = new Suit(1);
  public static final Suit DIAMONDS = new Suit(2);
  public static final Suit HEARTS   = new Suit(3);
  public static final Suit SPADES   = new Suit(4);

  private final int ordinalInt;


  private Suit(int requiredOrdinalInt)
  {
    ordinalInt = requiredOrdinalInt;
  } // Suit
```

Suit code

```
// Why is the parameter type Object, rather than Suit?
public boolean equals(Object other)
{
  if (other instanceof Suit)
    return this == other; // Safe because no duplicates.
  else
    return super.equals(other);
} // equals
// Look at the definition of equals() in the superclass --
// was this method needed here?
```

Suit code

```
// Why do we need to cast other to Suit?
public int compareTo(Object other)
{
  return ordinalInt - ((Suit) other).ordinalInt;
} // compareTo
```

Suit code

```
  public String toString()
  {
    if      (this == CLUBS)    return "clubs";
    else if (this == DIAMONDS) return "diamonds";
    else if (this == HEARTS)   return "hearts";
    else if (this == SPADES)   return "spades";
    else                       return "impossible!";
  } // toString


} // class Suit
```

Array of Faces

- `Face` is similar to `Suit`.
- We shall have 4 constants for Ace, King, Queen and Jack.
- Rather than 9 more separate constants, for convenience we shall have a static array to store the references to the 9 faces which are based on the numbers 2 to 10.
- This array will be initialised by a **static initialiser**.

Static initialiser

- A **static initialiser** is a piece of code in a class that is executed once when the program starts (as the class is loaded). It can be used to initialise values of static variables.
- The syntax is:

```
static { /* Put here any code you like,
            but it can only access static variables. */
       } // static initialiser
```

- We have not needed one so far (I think), as the values of static variables have been given via a single assignment.

```
public static int X = 10;
```

Is equivalent to

```
public static int X;
static { X = 10; }
```

Face code

```
public class Face implements Comparable
{
  public static final Face JACK  = new Face(11);
  public static final Face QUEEN = new Face(12);
  public static final Face KING  = new Face(13);
  public static final Face ACE   = new Face(14);
```

Face code

```
// A static array for the numbers mapping int onto Face.
// Positions 0 and 1 are unused.
private static final Face [] numbers = new Face [11];

// A static initialiser for the array.
static
{
  for (int i = 2; i <= 10; i++)
    numbers[i] = new Face(i);
} // static initialiser
```

Face code

```
private final int ordinalInt;


private Face(int requiredOrdinalInt)
{
  ordinalInt = requiredOrdinalInt;
} // Face
```

Face code

```
// An accessor function for the numbers array.
public static Face number(int i)
{
  if (i >= 2 && i <= 10)
    return numbers[i];
  else
    return null;
} // number
```

Face code

```
// Why is the parameter type Object, rather than Suit?
public boolean equals(Object other)
{
  if (other instanceof Face)
    return this == other; // Safe because no duplicates.
  else
    return super.equals(other);
} // equals
// Look at the definition of equals() in the superclass --
// was this method needed here?
```

Face code

```
// Why do we need to cast other to Face?
public int compareTo(Object other)
{
  return ordinalInt - ((Face) other).ordinalInt;
} // compareTo
```

Face code

```
public String toString()
{
  if      (this == numbers[2])  return "two";
  else if (this == numbers[3])  return "three";
  else if (this == numbers[4])  return "four";
  else if (this == numbers[5])  return "five";
  else if (this == numbers[6])  return "six";
  else if (this == numbers[7])  return "seven";
  else if (this == numbers[8])  return "eight";
  else if (this == numbers[9])  return "nine";
  else if (this == numbers[10]) return "ten";
```

Face code

```
    else if (this == ACE)          return "ace";
    else if (this == JACK)         return "jack";
    else if (this == QUEEN)        return "queen";
    else if (this == KING)         return "king";
    else                           return "impossible!";
  } // toString

} // class Face
```

Example Faces

- The 13 instances of `Face` are accessed either by a constant name, or via the `number()` method. For example:

```
Face aFace = Face.ACE;
Face anotherFace = Face.number(7);
```

Card code

```
public class Card implements Comparable
{
  private final Suit suit;
  private final Face face;

  public Card(Suit requiredSuit, Face requiredFace)
  {
    suit = requiredSuit;
    face = requiredFace;
  } // Card
```

Card code

```
public Suit getSuit()
{
  return suit;
} // getSuit


public Face getFace()
{
  return face;
} // getFace
```

Card code

```
public boolean equals(Card other)
{
  // We can use == as there are no duplicates in Suit and Face.
  return suit == other.suit && face == other.face;
} // equals

// Why do we need to cast other to Card?
public int compareTo(Object other)
{
 Card otherCard = (Card) other;
 if (suit == otherCard.suit) // Safe because no duplicates.
   return face.compareTo(otherCard.face);
 else
   return suit.compareTo(otherCard.suit);
} // compareTo
```

Card code

```
  public String toString()
  {
    return face.toString() + " of " + suit.toString();
  } // toString

} // class Card
```

Pros and Cons

- The most attractive feature of this approach is its safety – because we have control over what anyone can do with `Suits` and `Faces`, people are protected from their stupidity.
- The disadvantage is that it is a little more complex than using integers directly.

Section 4

Approach C:
Using Inheritance

Have the classes `Suit` and `Club` (etc.)

- In this third approach we consider that `Club` is not an *instance* of `Suit` but a *subclass* of it. Obviously we think the same of the other 3 suits.
- This means we do not need to use a conditional statement when we need variations in the properties of the different suits. Instead each subclass overrides a method in the `Suit` class. For example each suit subclass overrides `toString()`.
- We do not want anyone to make instances of `Suit` directly so we shall declare that class as `abstract`.
- We want exactly one instance of each of the four suits, so their constructors will be `private`, and we shall have one `public static final` variable in each referring to the one instance. We shall call that variable `SUIT`. So the four suits will be accessed as

```
Clubs.SUIT  Diamonds.SUIT  Hearts.SUIT  Spades.SUIT
```

Suit code

```
public abstract class Suit implements Comparable
{
  // This is now a method so it can be overridden / implemented.
  // Making it abstract forces it to be implemented.
  protected abstract int ordinalInt();

  // Making it protected means it can be accessed by classes in the same
  // package, plus subclasses.

  // This is the same as public if we have only the default package!

  // Java could do with another, tighter, visibility,
  // for access in subclasses only.
```

Suit code

```
// Needed?
public boolean equals(Object other)
{
  if (other instanceof Suit)
    return this == other; // Safe because no duplicates.
  else
    return super.equals(other);
} // equals

public int compareTo(Object other)
{
  return ordinalInt() - ((Suit) other).ordinalInt();
} // compareTo
```

Suit code

```
  // This also must be implemented in subclasses.
  // Each subclass provides a method that returns the right string.
  public abstract String toString();

} // class Suit
```

Clubs code

```
public class Clubs extends Suit
{
  public final static Clubs SUIT = new Clubs();


  private Clubs() { } // No one else can make instances.


  protected int ordinalInt()
    { return 1; }


  public String toString()
    { return "clubs"; }

} // class Clubs
```

Diamonds code

```
public class Diamonds extends Suit
{
  public final static Diamonds SUIT = new Diamonds();

  private Diamonds() { } // No one else can make instances.

  protected int ordinalInt()
    { return 2; }

  public String toString()
    { return "diamonds"; }
} // class Diamonds
```

Faces

- We treat faces in the same way.
- We have one abstract class called `Face`.
- We have 13 subclasses called `Ace`, `King`, `Queen`, `Jack`, `Ten`, `Nine`, `Eight`, `Seven`, `Six`, `Five`, `Four`, `Three` and `Two`.
- These each have `private` constructors, and make one instance referenced from a `public static final` variable called `FACE`. That is, they will be accessed as

```
Ace.FACE  King.FACE  Queen.FACE  Jack.FACE  Ten.FACE ...
```

Face code

```
public abstract class Face implements Comparable
{
  // This is now a method so it can be overridden / implemented.
  // Making it abstract forces it to be implemented.
  protected abstract int ordinalInt();
```

Face code

```
// Needed?
public boolean equals(Object other)
{
  if (other instanceof Face)
    return this == other; // Safe because no duplicates.
  else
    return super.equals(other);
} // equals

public int compareTo(Object other)
{
  return ordinalInt() - ((Face) other).ordinalInt();
} // compareTo
```

Face code

```
  // This also must be implemented in subclasses.
  public abstract String toString();

} // class Face
```

Ace code

```
public class Ace extends Face
{
  public final static Ace FACE = new Ace();

  private Ace() { } // No one else can make instances.

  protected int ordinalInt()
    { return 14; }

  public String toString()
    { return "ace"; }
} // class Ace
```

Two code

```
public class Two extends Face
{
  public final static Two FACE = new Two();


  private Two() { } // No one else can make instances.


  protected int ordinalInt()
    { return 2; }


  public String toString()
    { return "two"; }

} // class Two
```

`Card` code

- The `Card` class in approach C is identical to that in approach B.

Pros and Cons

- It has the safety of approach B.
- It is, possibly, more efficient: if/else statments are replaced by **dynamic method binding**.
- On the other hand, it is much more code than approach B!

Section 5

A Pack of Cards

Card collections

- Having only single cards would not be much fun!
- We want to have collections of cards.
- For example one obvious collection is a Pack of Cards.
- However, in a game of cards we might have collections like a hand, a discard pile, a take up pile, etc..
- We shall start by creating a super-class of card collections with some general features.
- As it happens, the code for this class is the same in all three of the approaches to modelling cards.

CardCollection

- A `CardCollection` is based on an `ArrayList` of `Cards`.
- It has two obvious places where we can add cards – the low end (index 0) and the high end. The high end is more efficient as it does not require the `ArrayList` to shift cards up to make room when we insert a card or down to close a gap when we remove one.
- So subclasses of `CardCollection` which only need to use one end for adding and removing cards should use the high end. For example, a stack of cards.
- Some subclasses of `CardCollection` will need to use both ends, For example a queue of cards.
- In addition to adding and removing cards, other operations on a `CardCollection` include shuffling and sorting.

CardCollection code

```
import java.util.ArrayList;
import java.util.Collections;


public class CardCollection
{
  private ArrayList cards = new ArrayList();

  public CardCollection()
  {
  } // CardCollection
```

CardCollection code

```
public boolean isEmpty()
{
  return cards.size() == 0;
} // isEmpty


public int size()
{
  return cards.size();
} // size
```

CardCollection code

```
// Could throw an exception for illegal index?
public Card lookAt(int index)
{
  if (index < 0 || index >= cards.size())
    return null;
  else
    return (Card) cards.get(index);
} // lookAt
```

CardCollection code

```
public void addHigh(Card card)
{
  cards.add(card);
} // addHigh

public Card removeHigh()
{ if (cards.size() == 0)
    return null;
  else
  { int highestIndex = cards.size() - 1;
    Card result = (Card) cards.get(highestIndex);
    cards.remove(highestIndex);
    return result;
  } // else
} // removeHigh
```

CardCollection code

```
public void addLow(Card card)
{
  cards.add(0, card);
} // addLow

public Card removeLow()
{
  if (cards.size() == 0)
    return null;
  else
  { Card result = (Card) cards.get(0);
    cards.remove(0);
    return result;
  } // else
 } // removeLow
```

CardCollection code

```
public void shuffle()
{
  ArrayList shuffledCards = new ArrayList();
  while (cards.size() > 0)
  {
    int randomIndex = (int) (Math.random() * cards.size());
    shuffledCards.add(cards.get(randomIndex));
    cards.remove(randomIndex);
  } // while
  cards = shuffledCards;
} // shuffle
```

CardCollection code

```
public void sort()
{
  Collections.sort(cards);
} // sort
```

CardCollection code

```
  public CardCollection removeAll()
  {
    CardCollection result = new CardCollection();
    result.cards = cards;
    cards = new ArrayList();  // Sharing issue?
    return result;
  } // removeAll


} // class CardCollection
```

A pack of cards

- A pack of cards is one type of `CardCollection` so we define it as a subclass.
- A `CardPack` always starts off with one copy of all 52 cards.
- The code for `CardPack` is different for each of the three approaches to modelling cards, because of the different way we build actual cards. Here we show the version for approach B.

CardPack code

```
public class CardPack extends CardCollection
{
  public CardPack()
  {
    addNewSuitOfCards(Suit.SPADES);
    addNewSuitOfCards(Suit.HEARTS);
    addNewSuitOfCards(Suit.CLUBS);
    addNewSuitOfCards(Suit.DIAMONDS);
  } // CardPack
```

CardPack code

```
  private void addNewSuitOfCards(Suit suit)
  {
    addHigh(new Card(suit, Face.ACE));
    addHigh(new Card(suit, Face.JACK));
    addHigh(new Card(suit, Face.QUEEN));
    addHigh(new Card(suit, Face.KING));
    for (int i = 2; i <= 10; i++)
      addHigh(new Card(suit, Face.number(i)));
  } // addNewSuitOfCards


  public Card dealCard()
  {
    return removeHigh();
  } // dealCard

} // class CardPack
```

A test program

- We want to have a test program to check that our implementation works.
- The test program presented here proceeds as follows.
  - Take two packs of cards, pack1 and pack2.
  - Shuffle pack1 and deal seven cards from it – showing these cards on the standard output.
  - Next sort both packs.
  - Now deal from each, and see where they are different. Show the similar and different cards on the standard output.
  - The result should show seven random cards, followed by a sorted pack of cards indicating where the seven random cards would be in the sequence.

TestPack code

```
public class TestPack
{
  public static void main(String args [])
  {
     CardPack pack1 = new CardPack();
     CardPack pack2 = new CardPack();
     pack1.shuffle();
     for (int i = 1; i <= 7; i++)
     {
       Card card = pack1.dealCard();
       System.out.println(card);
     } // for
     System.out.println("----------------------------");
     pack1.sort();
     pack2.sort();
```

TestPack code

```
// Go through both packs and compare cards.
// Pack1 will run out no later than pack2.
while (! pack1.isEmpty())
{
  Card card1 = pack1.dealCard();
  Card card2;
  do
  {
    card2 = pack2.dealCard();
    if (! card2.equals(card1))
      System.out.println("------ " + card2);
  } while (! card2.equals(card1));
  System.out.println(card1);
} // while
```

TestPack code

```
    // Now finish of pack2 if needed.
    while (! pack2.isEmpty())
    {
      Card card2 = pack2.dealCard();
      System.out.println("------ " + card2);
    } // while

  } // main

} // class TestPack
```

TestPack sample output

```
two of hearts
three of clubs
ten of hearts
six of clubs
six of hearts
five of hearts
jack of clubs
------------------------------
ace of spades
king of spades
queen of spades
jack of spades
ten of spades
nine of spades
eight of spades
```

TestPack sample output

```
seven of spades
six of spades
five of spades
four of spades
three of spades
two of spades
ace of hearts
king of hearts
queen of hearts
jack of hearts
------ ten of hearts
nine of hearts
eight of hearts
seven of hearts
------ six of hearts
```

TestPack sample output

```
------ five of hearts
four of hearts
three of hearts
------ two of hearts
ace of diamonds
king of diamonds
queen of diamonds
jack of diamonds
ten of diamonds
nine of diamonds
eight of diamonds
seven of diamonds
six of diamonds
five of diamonds
four of diamonds
```

TestPack sample output

```
three of diamonds
two of diamonds
ace of clubs
king of clubs
queen of clubs
------ jack of clubs
ten of clubs
nine of clubs
eight of clubs
seven of clubs
------ six of clubs
five of clubs
four of clubs
------ three of clubs
two of clubs
```

Section 6

A Game of Snap

A game of Snap

- To put our cards to ‘good’ use, we next develop a simple card game.
- Actually, it is a Human reaction enhancement program, based on the game of Snap.
- The user can either play against the computer or another Human.
- The two players are dealt cards into a playing queue, face down. They then take it in turns to play cards onto a discard stack, face up. The card playing is automatic – no user interaction is required.
- When the top two cards of the stack match, the players can claim a **Snap** by moving their hand (image) onto the stack. The first player to get his or her hand on the stack wins the contents of it – those cards are added to the back of his or her playing queue.

The game speed

- The whole point of the game is to play as fast possible.
- The speed is regulated by the program.
- If a Human player wins the stack, the game speeds up. Otherwise, whenever there is a match which is not won by a Human player, the game slows down.
- When two Humans are playing, this means neither was fast enough to claim the stack in the time allowed.
- When playing against the computer, then either the Human or the computer will always win the stack whenever there is a match. The computer waits a period of time before moving its hand, depending on the speed of the game.

The end of the game

- The game never ends.
- When a player runs out of playing queue, the other takes all the turns.
- Whenever two of the three card piles are empty, both players are dealt an additional new pack of cards to add to their playing queues.

Example play

- The game is available in /opt/teaching/bin/Snap
- Run:  /opt/teaching/bin/Snap
- When there is a match, you can move your hand to the centre by pressing space.
- For a slower game, you can also use the mouse to drag your hand image to the centre – good for practising use of the mouse.
- The two person game is selected with the command line option ”2”. Mouse dragging is not available (obviously) – instead the left player uses the ‘A’ key to move, and the other uses the return key.

Classes of the implementation

- These classes are based on approach B to the model of cards.

Class list for Snap program

Class	Description
`Suit`	The suits using approach B.
`SuitImage`	Provides images for suits, i.e. a single club, diamond, heart and spade image. A custom subclass of `Component`.
`Face`	The faces using approach B.

Classes of the implementation

Class list for Snap program

Class	Description
`Card`	The cards using approach B.
`CardImage`	Images for cards. These produce 13 small suit images for Kings, etc. rather than a proper picture. This is a custom subclass of `Component`.
`CardCollection`	Super class of the card collection classes, provides shuffle and sort.

Classes of the implementation

Class list for Snap program

Class	Description
`CardPack`	A pack of cards. A subclass of `CardCollection`.
`CardQueue`	A queue of cards – first in first out. Used for the playing queues of the two players. A subclass of `CardCollection`.
`CardQueueImage`	Images for `CardQueue`. These show a pile of cards face down and indicate depth proportional to the size of the queue.

Classes of the implementation

Class list for Snap program

Class	Description
`CardStack`	A stack of cards – last in first out. Used for the discard pile between the two players. A subclass of `CardCollection`.
`CardStackImage`	Images for `CardStack`. These show the cards face up, and randomly disarrayed for effect.
`Hand`	The model of a hand – position, home positions etc..
`HandImage`	The image of a `Hand` – colour, size etc..

Classes of the implementation

Class list for Snap program

Class	Description
`SpeedController`	Speed controller for the game, offering a number of speeds and speed up / down methods. A particular feature is the 'binary chop' mode which helps to quickly find a settled speed to suit the player at the start of the game.
`SpeedControllerImage`	An image and interface for the speed controller.

Classes of the implementation

Class list for Snap program

Class	Description
`SnapGame`	The game play part of the program – an extension of `Panel`.
`Snap`	The whole `Frame` with buttons etc.. It also contains the main method.

Section 7

Code management issues

Installing Snap as a command

- We wish to install the Snap program into `/opt/teaching/bin` so that people can run it by simply typing `Snap`.
- You are used to running a Java program by explicitly invoking `java` with the name of the main class whilst in the directory containing the class files.
- So how do we achieve our aim? Here is the way I often do it.

Installing Snap as a command

- We put all the class files into a **jar** file called `Snap.jar`.
- A jar file is similar to a zip or tar file: it is simply a file which contains many other files in such a way that the right bit of software can retrieve them. Jar files are Java's format of such 'archive' files.
- Then we copy `Snap.jar` to `/opt/teaching/bin` and also make a symbolic link called `/opt/teaching/bin/Snap` to a shell script called `/opt/teaching/bin/run-jar`.
- This shell script figures out the name of the main class from the basename of `$0`. It also knows the name of the jar file, by appending `.jar` onto `$0`. So it can then start `java` to run the program.

Installing Snap as a command

- This can all be done as follows.

```
jar -cf Snap.jar *.class
cp Snap.jar /opt/teaching/bin/Snap.jar
cd /opt/teaching/bin
ln -s run-jar Snap
```

- The file `/opt/teaching/bin/run-jar` contains

```
Basename=‘basename $0‘
export CLASSPATH="$0.jar"
exec java $Basename "$@"
```

- You can use this approach for your programs.

Makefiles

- Once you get past a handful of Java classes you wish to have some systematic way to recompile those files which have been altered, without recpmpiling those which have not.
- A makefile (or some similar thing) is what you need. This is used with the program `make`.

Makefile for Snap

- The Snap program makefile. (It is quite advanced for a beginner.)
- Note that lines which are offset to the right are started with a tab, not multiple spaces: this is important (unfortunately).
- You can find out more about makefiles and `make` by running `man make`.

```
JAVAC=javac
JARNAME=Snap.jar
INSTALLPATH=/opt/teaching/bin
JAVA := $(wildcard *.java)
CLASS := $(JAVA:%.java=%.class)

all : ${CLASS} ${JARNAME}

clean:
        rm *.class ${JARNAME}
```

Makefile for Snap

```
%.class : %.java
        unset CLASSPATH; ${JAVAC} $<

${JARNAME}: ${CLASS}
        jar -cf ${JARNAME} *.class

install: all
        (test ! -d ${INSTALLPATH} \
           && (mkdir -p ${INSTALLPATH}; \
               chmod a+rx ${INSTALLPATH}); \
         true)
        jar -cf ${INSTALLPATH}/${JARNAME} *.class
        chmod a+r ${INSTALLPATH}/${JARNAME}
```

Software tools

- The exercises you have done in laboratories have mostly required you to work with only a small number of classes at once.
- Some of you have discovered various IDE tools to ‘help’. Our view is that these can actually *hinder* in the early stages of learning, because they hide the important details.
- However, some of the examples you have seen in the course begin to challenge the limits of what can comfortably be managed without additional tools.
- Next year you will be introduced to an IDE which you can use for Java code development, and also for making UML diagrams.

Make your own software tools

- You have all heard the expression “A poor workman blames his tools.” I always like to add the corollary “But a good workman can make his own.”
- I invite you to consider the view that you will need software tools to help you build software. But you should (perhaps ultimately) consider the possibility of building *some* of these yourself, rather than always soak up what some other body has decided is good for you.
- Especially if you are working in a Unix environment, then some really useful tools can be simple to build. (Arguably, Unix is the best IDE in the world, if you are prepared to harness its power.)
- Recall the graphs of textual dependence between classes which we have seen. The data for those graphs was obtained using a simple shell script.

Make your own software tools

- Here are the requirements for a simple software tool.

  "I want a program that takes a string as an argument (e.g. a variable name) and looks in the current directory for any Java source files containing that string. For each source file found, I want it to create an xterm running the program `less`, with the source file being listed, and the search string already highlighted."

- Challenge: could you write this shell script?
- How many characters of code would it have?

Make your own software tools

- Here is the script I use for this.

```
#!/bin/sh
for i in ‘grep -il $1 *.java‘
do
   xterm -T "$i" -e less -I -p $1 "$i" &
done
```

- I once set this as a challenge at the end of a lecture, just before Christmas. The first person to meet the challenge had emailed me an answer within an hour of the lecture ending!

Make your own software tools

I also have a number of useful aliases defined in my `$HOME/.my_bashrc.all` file, as follows. As a parting challenge, see if you can work out what they do!

```
function do_suffix ()
{ local command="$1"
  local suffix="$2"
  shift 2
  local i
  local args
  local opts
```

Make your own software tools

```
  for i in "$@"
  do
    case "$i" in
    -*) opts=$opts" ${i}" ;;
    *\.$suffix) args=$args" ${i}" ;;
    *\.) args=$args" ${i}$suffix" ;;
    *) args=$args" ${i}.$suffix" ;;
    esac
  done
  test "$args" == "" && args="*.$suffix"
  $command $opts $args
}
```

Make your own software tools

```
function do_nosuffix ()
{ local command="$1"
  shift 1
  local i
  local args
  for i in "$@"
  do
    case "$i" in
    *\.) i=‘echo $i | sed "s,\.$,,g"‘
         args=$args" ${i}" ;;
    *) args=$args" ${i}" ;;
    esac
  done
  $command $args
}
```

Make your own software tools

```
alias ej="do_suffix $EDITOR java"
alias mj="do_suffix less java"
alias lj="do_suffix ls java"
alias llj="do_suffix \"ls -l\" java"
alias cj="do_suffix javac java"
alias wj="do_suffix wc java"
alias rj="do_nosuffix java"
```