Classes

1 Data Abstraction

An abstraction of a concept attempts to capture the essence of the concept – its essential properties and behaviors – by ignoring irrelevant details.

- Process abstraction - group the operations of a process in a subprogram and expose only the essential elements of the process to clients through the subprogram signature (e.g., function/method name and parameters)
- Data abstraction - encapsulation of data with the operations defined on the data
- A particular data abstraction is called an *abstract data type*. Note that ADT's include process abstractions as well

In each case, an abstraction hides details — details of a process or details of a data structure.

> "Abstraction is selective ignorance."
> – Andrew Koenig (C++ Guru)

2 A Complex Number ADT

ADT: Complex
Data:

- **real: double** the real part of a complex number
- **imaginary: double** the imaginary part of a complex number

Operations:

- **new** - construct a new complex number
- **plus** - add one complex number to another, yielding a new complex number

An ADT is *abstract* becuase the data and operations of the ADT are defined independently of how they are implemented. We say that an ADT *encapsulates* the data and the operations on the data.

3 Data Abstractions with Classes

Java provides langauge suppport for defining ADTs in the form of classes.

A class is a blueprint for objects. A class definition contains

- instance variables, a.k.a. member variables or fields – the state, or data of an object
- methods, a.k.a. member functions or messages – the operations defined on objects of the class

We *instantiate* or *construct* an object from a class.

4 Java Implementation of Complex Number ADT

Here's a Java implementation of our complex number ADT [1]:

```
public class Complex {
    // These are the data of the ADT

    private double real;
    private double imaginary;

    // These are the operations of the ADT

    public Complex(double aReal, double anImaginary) {
        real = aReal;
        imaginary = anImaginary;
    }

    public Complex plus(Complex other) {
        double resultReal = this.real + other.real;
        double resultImaginary = this.imaginary + other.imaginary;
        return new Complex(resultReal, resultImaginary);
    }
}
```

5 Reference Variables

Consider the following code:

```
Complex a = new Complex(1.0, 2.0);
Complex b = new Complex(3.0, 4.0);
Complex c = a.plus(b);
```

`a`, `b`, and `c` are *reference* variables of type `Complex`. Reference variables have one of two values:

- the address of an object in memory (in this case an instance of `Complex`), or
- `null`, meaning the variable references nothing.

[1] http://introcs.cs.princeton.edu/java/33design/

6 Invoking Constructors

The line:

```
Complex a = new Complex(1.0, 2.0);
```

invokes the `Complex` constructor, passing arguments `1.0` and `2.0`:

```
public Complex(aReal= 1.0 , anImaginary= 2.0 ) {
    real = 1.0 ;
    imaginary = 2.0 ;
}
```

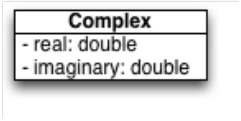
which *instantiates* a `Complex` object and stores its address in the variable `a`:

```
Complex a = new Complex(1.0, 2.0);
```

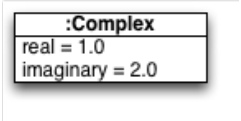
Constructors initialize objects. After the line above, `Complex` object `a`'s instance variables have the values `1.0` and `2.0`.

7 Visualizing Objects and Instantiation

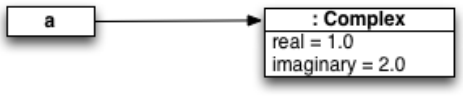
The object creation expression `new Complex(1.0, 2.0)` applies the `Complex` blueprint defined by the class definition from slide



to the constructor arguments `(1.0, 2.0)` to create an instance of `Complex`:



We can assign this object to a reference variable, e.g.,
`Complex a = new Complex(1.0, 2.0)`:



8 Invoking Methods on Objects

The line:

```
Complex c = a.plus(b);
```

invokes the `plus` method on the `a` object, passing the `b` object as an argument, which binds the object referenced by `b` to the parameter `other`:

```
a.plus(other= b ) {
  double resultReal = this.real + b .real; // 1.0 + 3.0
  double resultImaginary = this.imaginary + b .imaginary; // 2.0 + 4.0
  return new Complex(resultReal, resultImaginary);
}
```

which returns a new `Complex` object and assigns its address to the reference variable `c`.

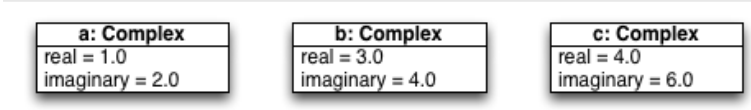
9 Using the `Complex` Class

Users, or *clients* of the `Complex` class can then write code like this:

```
Complex a = new Complex(1.0, 2.0);
Complex b = new Complex(3.0, 4.0);
Complex c = a.plus(b);
```

without being concerned with `Complex`'s implementation (which could use polar form, for example). Clients (i.e., users) of the `Complex` class need only be concerned with its interface, or *API* (application programmer interface) – the public methods of the class.

After the code above we have the following `Complex` objects in memory:



10 The Anatomy of a Class Definition

`Card.java`

```
import java.util.Arrays;

public class Card {

  public static final String[] VALID_RANKS = {"2",
     ... , "ace"};
  public static final String[] VALID_SUITS =
     {"diamonds", ... };
  private String rank;
  private String suit;

  public Card(String aRank, String aSuit) {
    // ...
  }
  public String toString() {
    return rank + " of " + suit;
  }
  private boolean isValidRank(String someRank) { ... }
}
```

A class definition file contains

- import statements
- class declaration
- static variable definitions
- instance variable definitions
- constructor
- public methods
- private helper methods

11 A Card Class, v0

Consider how to represent a Card ADT:

- rank - the rank of a playing card, e.g., 2, jack, ace
- suit - the suit of a playing card, e.g., spades, diamonds

```
public class Card0 {

    String rank;
    String suit;
}
```

- `rank` and `suit` are instance variables
- Every *instance* of `Card0` has its own copy of instance variables.

12 Using `Card0`

```
public class Card0 {

    String rank;
    String suit;

    public static void main(String[] args) {
        Card0 c = new Card0();
        System.out.println(c);
    }
}
```

Note that we can put a `main` method in any class. This is useful for exploratory testing, like we're doing here.

The `String` representation isn't very appealing. (What does it print?)

13 Card Class, v1

```
public class Card1 {

    String rank;
    String suit;

    public String toString() {
        return rank + " of " + suit;
    }
    public static void main(String[] args) {
        Card1 swedishPop = new Card1();
        swedishPop.rank = "ace";
        swedishPop.suit = "base";
        Card1 handy = new Card1();
        handy.rank = "jack";
        handy.suit = "all trades";
        System.out.println(swedishPop);
        System.out.println(handy);
    }
}
```

Now we have an "ace of base" card and a "jack of all trades" card. But those aren't valid cards.

14 Encapsulation: Card, v2

Let's protect the instance variables by making them private:

```
public class Card2 {

    private String rank;
    private String suit;

    public String toString() {
        return rank + " of " + suit;
    }

    public static void main(String[] args) {
        Card2 c = new Card2();
        c.rank = "ace";
        c.suit = "base";
        System.out.println(c);
    }
}
```

Why does this still compile?

- `main` method still part of `Card2` - has access to private parts

Let's make a dealer class to represent client code.

15 A Dealer Class

2

(We'll synchronize the names of Dealer classes with the names of Card classes, so `Dealer2` is meant to test `Card2`.)

```
public class Dealer2 {

    public static void main(String[] args) {
        Card2 c = new Card2();
        c.rank = "ace";
        c.suit = "base";
        System.out.println(c);
    }
}
```

This won't compile (which is what we want). Why?

16 Mutators: Card, v3

```
public class Card3 {

    private String rank;
    private String suit;

    public void setRank(String rank) {
        rank = rank;
    }
    public void setSuit(String suit) {
        suit = suit;
    }
}
```

- Now client code can set the rank and suit of a card by calling `setRank` and `setSuit`.
- `setX` is the Java convention for a setter method for an instance variable named `x`.

17 Dealing `Card3`

Let's try out our new `Card3` class.

```
public class Dealer3 {

    public static void main(String[] args) {
        Card3 c = new Card3();
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

Oops. Prints "null of null". Why?

[2] Our Dealer class plays the role that a Driver class often plays in your homework.

18 Shadowing Variables

The parameters in the setters "shadowed" the instance variables:

```
public void setRank(String rank) {
    rank = rank;
}

public void setSuit(String suit) {
    suit = suit;
}
```

- `rank` in `setRank` refers to the local `rank` variable, not the instance variable of the same name
- `suit` in `setSuit` refers to the local `suit` variable, not the instance variable of the same name

19 Dealing with `this`: Card, v4

```
public class Card4 {
    private String rank;
    private String suit;

    public void setRank(String rank) {
        this.rank = rank;
    }

    public void setSuit(String suit) {
        this.suit = suit;
    }
}
```

- Every instance of a class has a `this` reference which refers to the instance on which a method is being called.
- `this.rank` refers to the `rank` instance variable for the `Card4` instance on which `setRank` is being called.
- `this.rank` is different from the local `rank` variable that is a parameter to the `setRank` method.

20 Dealing `Card4`

```
public class Dealer4 {

    public static void main(String[] args) {
        Card4 c = new Card4();
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

Now we have encapsulation, but we can still create invalid `Card4`s, e.g., "base" is not a valid suit. How to fix?

21 Class Invariants

Class invariant: a condition that must hold for all instances of a class in order for instances of the class to be considered valid.

Invariants for Card class:

- `rank` must be one of {`"2"`, `"3"`, `"4"`, `"5"`, `"6"`, `"7"`, `"8"`, `"9"`, `"10"`, `"jack"`, `"queen"`, `"king"`, `"ace"`}
- `suit` must be one of {`"diamonds"`, `"clubs"`, `"hearts"`,`"spades"`}

22 Maintaining Class Invariants via Input Validation

`rank` invariant can be maintained by adding:

```
public class Card5 {
    private final String[] VALID_RANKS =
        {"2", "3", "4", "5", "6", "7", "8", "9",
         "10", "jack", "queen", "king", "ace"};
    public void setRank(String rank) {
        if (!isValidRank(rank)) {
            System.out.println(rank + " is not a valid rank.");
            System.exit(0);
        }
        this.rank = rank;
    }
    private boolean isValidRank(String someRank) {
        return contains(VALID_RANKS, someRank);
    }
    private boolean contains(String[] array, String item) {
        for (String element: array) {
            if (element.equals(item)) {
                return true;
            }
        }
        return false;
    }
    // ...
}
```

23 Class Invariants Ensure Consistent Objects

Now we can't write code that instantiates an invalid `Card5` object:

```
public class Dealer5 {

    public static void main(String[] args) {
        Card5 c = new Card5();
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

yields:

```
$ java Dealer5
base is not a valid suit.
```

24 Progress Check

Let's review our progress with our Card class design:

- We have a nice string representation of Card objects (`Card1`).
- We have encapsulated the rank and suit in private instance variables (`Card2`) with mutator methods (`Card4`) to set their values.
- We validate the rank and suit in the mutator methods so we can't set invalid ranks and suits in Card objects (`Card5`).

25 Classes and Objects

`Card5` now ensures that we don't create card objects with invalid ranks or suits. But consider this slight modification to `Dealer5`:

```
public class Dealer5 {

    public static void main(String[] args) {
        Card5 c = new Card5();
        System.out.println(c);
        c.setRank("ace");
        c.setSuit("base");
        System.out.println(c);
    }
}
```

What if we printed our `Card5` instance, `c`, before we called the setters?

26 Object Initialization

There are two ways to initialize the instance variables of an object:

- Declaration point initialization:

```
public class Card {
  private String rank = "2";
  // ...
}
```

- Constructors

```
public class Card {
  public Card() {
```

```
    rank = "2";
  }
  // ...
}
```

A constructor is what's being called when you invoke operator `new`.

27 Initializing Objects

Since we didn't write our own constructor, Java provided a default no-arg constructor that simply sets instance variables (that don't have their own declaration-point intializations) to their default values. That's why `Card5` objects are `null of null` after they're instantiated. We have to call the setters on a `Card5` instance before we have a valid object.

In general, it's poor style to require multi-step initialization.

- After `new Card5()` is called, instance variables have useless defaults.
- Client programmer must remember to call setter methods.
- Often there can be order dependencies that we don't want to burden client programmers with.

The way to fix this is by writing our own constructor.

28 A Card Constructor

If we write a constructor, Java won't provide a default no-arg constructor. (We may choose to provide one.)

```
public class Card6 {
    // ...
    public Card6(String rank, String suit) {
        setRank(rank);
        setSuit(suit);
    }
    // ...
}
```

Now we have a safer, more consistent way to initialize objects:

```
public class Dealer6 {

    public static void main(String[] args) {
        Card6 c = new Card6("queen", "hearts");
        System.out.println(c);
    }
}
```

29 Static Members

Do we need a separate instance of `VALID_RANKS` and `VALID_SUITS` for each instance of our Card class?

`static` members are shared with all instances of a class:

```
public static final String[] VALID_RANKS =
    {"2", "3", "4", "5", "6", "7", "8", "9",
     "10", "jack", "queen", "king", "ace"};
public static final String[] VALID_SUITS =
    {"diamonds", "clubs", "hearts","spades"};
```

Given the declarations above:

- Each instance shares a single copy of `VALID_RANKS` and a single copy of `VALID_SUITS`
- Since they're `final`, we can safely make them `public` so clients of our Card class can use them

30 One Final Enhancement

`Card6` is pretty good, but we can write code like this:

```
public class Dealer6 {

    public static void main(String[] args) {
        Card6 c = new Card6("queen", "hearts");
        System.out.println(c);
        c.setRank("jack"); // modifying c
        System.out.println(c);
    }
}
```

Does this make sense? Should Card objects be mutable?

31 Immutable Objects

Card objects don't change. We can model this behavior by removing the setters and putting the initialization code in the constructor (or making the setters private and calling them from the constructor):

```
public Card(String aRank, String aSuit) { // constructor
  if (!isValidRank(rank)) {
    System.out.println(aRank + " is not a valid rank.");
    System.exit(0);
  }
  rank = aRank;
  if (!isValidSuit(aSuit)) {
    System.out.println(aSuit + " is not a valid suit.");
    System.exit(0);
  }
  suit = aSuit;
}
```

Note the use of another idiom for disambiguating constructor paramters from instance variables (as opposed to using `this`).

32 Designing Immutable Classes

An immutable class is a class whose instances cannot be modified. To make a class immutable:

- Don't provide mutator methods ("setters")
- Make the class `final` so it can't be extended (there's another way to accomplish this, but making the class `final` is good enough for now)
- Make all fields `final`
- Make all fields `private`
- For fields of mutable class types, return defensive copies in accessor methods (we'll discuss this later)

33 Prefer Immutable Classes

In general, make your classes immutable unless you have a good reason to make them mutable. Why? Because immutable objects

- are simpler to design becuase you don't have to worry about enforcing class invariants in multiple places,
- are easier to reason about because the state of an object never changes after instantiation,
- are inherently thread-safe becuase access to mutable data need not be syncronized, and
- enable safe instance sharing, so redundant copies need not be created.

34 A Few Final Bits of Polish

Take a look at the final evolution of our Card class, Card.java. It contains a few more enhancements:

- Instead of simply terminating the program, the constructor throws `IllegalArgumentException` on invalid input so that client code can choose to deal with the exception at run-time.
- Input is normalized to lower case and spaces trimmed to make the Card object robust to oddly formatted input.
- It has an `equals()` method.

35 Equality

- == means identity equality (aliasing) for reference types (objects).
- The `equals(Object)` tests value equality for objects.

Given our finished Card.java class with a properly implemented `equals(Object)` method, this code:

```
  Card c1 = new Card("ace", "spades");
  Card c2 = new Card("ace", "spades");
  Card c3 = c1;
  System.out.println("c1 == c2 returns " + (c1 == c2));
  System.out.println("c1.equals(c2) returns " + c1.equals(c2));
  System.out.println("c1 == c3 returns " + (c1 == c3));
  System.out.println("c1.equals(c3) returns " + c1.equals(c3));
```

produces this output:

```
  c1 == c2 returns false
  c1.equals(c2) returns true
  c1 == c3 returns true
  c1.equals(c3) returns true
```

By the way, what if we left off the parentheses around `c1 == c2` in `System.out.println("c1 == c2 returns " + (c1 == c2))`?

36 Review Question 1

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten fiona = new Kitten("Fiona");
        Kitten fiona2 = new Kitten("Fiona");
```

What is the value of `maggie`?

- ?

37 Review Question 1 Answer

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten fiona = new Kitten("Fiona");
        Kitten fiona2 = new Kitten("Fiona");
```

What is the value of `maggie`?

- the address of a `Kitten` object

38 Review Question 2

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten fiona = new Kitten("Fiona");
        Kitten fiona2 = new Kitten("Fiona");
```

What does `maggie.toString()` return?

- ?

39 Review Question 2 Answer

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten fiona = new Kitten("Fiona");
        Kitten fiona2 = new Kitten("Fiona");
```

What does `maggie.toString()` return?

- `"Kitten:  null"`

40 Review Question 3

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten fiona = new Kitten("Fiona");
        Kitten fiona2 = new Kitten("Fiona");
```

What is the value of the expression `fiona == fiona2`?

- ?

41 Review Question 3 Answer

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten fiona = new Kitten("Fiona");
        Kitten fiona2 = new Kitten("Fiona");
```

What is the value of the expression `fiona == fiona2`?

- `false`

42 Review Question 4

```
public class Kitten {
```

```
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten[] kittens = new Kitten[5];
```

What is the value of `kittens[0]` ?

- ?

43 Review Question 4 Answer

```
public class Kitten {
    private String name;

    public Kitten(String name) {
        name = name;
    }
    public String toString() {
        return "Kitten: " + name;
    }
}
```

Assume the following statements have been executed:

```
        Kitten maggie = new Kitten("Maggie");
        Kitten[] kittens = new Kitten[5];
```

What is the value of `kittens[0]` ?

- `null`