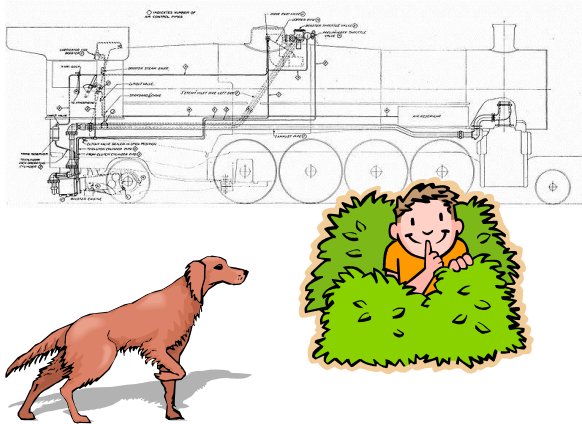


Designing data types



Fundamentals of Computer Science • Keith Vertanen • Copyright © 2013

Object Oriented Programming

- **Procedural programming** [verb-oriented]
 - Tell the computer to do this
 - Tell the computer to do that
- **OOP philosophy**
 - Software **simulation** of real world
 - We know (approximately) how the real world works
 - Design software to model the real world
- **Object oriented programming (OOP)** [noun-oriented]
 - Programming paradigm based on data types
 - **Identify**: objects that are part of problem domain or solution
 - Objects are distinguishable from each other (references)
 - **State**: objects know things (instance variables)
 - **Behavior**: objects do things (methods)

3

Overview

- **Object Oriented Programming (OOP)**
- **Data encapsulation**
 - Important consideration when designing a class
 - Access modifiers
 - Immutability, preventing change to a variable
- **Checking for equality**
 - Not always has simple as you might think!
 - floating-point variables
 - reference variables
 - String variables

2

Alan Kay

- **Alan Kay** [Xerox PARC 1970s]
 - Invented Smalltalk programming language
 - Conceived portable computer
 - Ideas led to: laptop, modern GUI, OOP



Alan Kay
2003 Turing Award



Dynabook: A Personal
Computer for Children of All
Ages, 1968.

“The computer revolution hasn’t started yet.”

“The best way to predict the future is to invent it.”

*“If you don’t fail at least 90 per cent of the time,
you’re not aiming high enough.”*

— Alan Kay

4

Data encapsulation

- **Data type**
 - "Set of values and operations on those values"
 - e.g. int, String, Charge, Picture, Enemy, Player
- **Encapsulated data type**
 - **Hide** internal representation of data type.
- **Separate implementation from design specification**
 - **Class** provides data representation & code for operations
 - **Client** uses data type as black box
 - **API** specifies contract between client and class
- **Bottom line:**
 - You don't need to know how a data type is implemented in order to use it

5

Intuition



Client



API

- volume
- change channel
- adjust picture
- decode NTSC signal



Implementation

- cathode ray tube
- electron gun
- Sony Wega 36XBR250
- 241 pounds

Client needs to know how to use API

Implementation needs to know what API to implement

Implementation and client need to agree on API ahead of time.

6

Intuition



Client



API

- volume
- change channel
- adjust picture
- decode NTSC signal

Client needs to know how to use API



Implementation

- gas plasma monitor
- Samsung FPT-6374
- wall mountable
- 4 inches deep

Implementation needs to know what API to implement

Can substitute better implementation without changing the client.

7

Time Bombs

- **Internal representation changes**
 - **[Y2K]** Two digit years: Jan 1, 2000
 - **[Y2038]** 32-bit seconds since 1970: Jan 19, 2038



<http://xkcd.com/607/>

- **Lesson**

- By exposing data representation to client, may need to sift through millions of lines of code to update

8

Access modifiers

- Access modifier
 - All instance variables and methods have one
 - **public** - everybody can see/use
 - **private** - only class can see/use
 - **protected** - only class and subclasses (stay tuned)
 - **default** - everybody in package, what you get if you don't specify an access modifier
 - Normally:
 - Instance variables are private
 - Methods the world needs are public
 - Helper methods used only inside the class are private

9

Data encapsulation example

- Person class
 - Originally stored first & last name in one instance variable
 - Now we want them separated → change instance vars

```
public class Person
{
    private String name = "";
    private double score = 0.0;

    public String toString()
    {
        return name;
    }
    ...
}
```

Original version, combined names

```
public class Person
{
    private String first = "";
    private String last = "";
    private double score = 0.0;

    public String toString()
    {
        String result = first;
        result += " ";
        result += last;
        return result;
    }
    ...
}
```

New version, names separated.

10

Non-encapsulated example

- If instance variables were public:
 - Client program might use instead of methods

```
public class Person
{
    public String first = "";
    public String last = "";
    public double score = 0.0;

    public String toString()
    {
        String result = first;
        result += " ";
        result += last;
        return result;
    }
    ...
}
```

Non-encapsulated version, instance variables are public.

```
...
Person p = new Person("Bob Dole");
system.out.println(p.name +
    " " +
    p.score);
...
```

Client program. Changing instance variables causes compile error. Client should have been using toString() but used instance variable because they were publically available.

11

Getters and setters

- Encapsulation does have a price
 - If clients need access to instance var, must create:
 - **getter methods** - "get" value of an instance var
 - **setter methods** - "set" value of an instance var

```
public double getPosX()
{
    return posX;
}
```

Getter method.
Also know as an **accessor** method.

```
public void setPosX(double x)
{
    posX = x;
}
```

Setter method.
Also know as a **mutator** method.



12

Immutability

- Immutable data type

- Object's value cannot change once constructed

<i>mutable</i>	<i>immutable</i>
Picture	Charge
Histogram	Color
Turtle	Stopwatch
StockAccount	Complex
Counter	String
Java arrays	primitive types

13

Immutability: Pros and Cons

- Immutable data type

- Object's value cannot change once constructed

- Advantages

- Avoid aliasing bugs
- Makes program easier to debug
- Limits scope of code that can change values
- Pass objects around without worrying about modification

- Disadvantage

- New object must be created for every value

14

String immutability: consequences

```
String s = "Hello world!";
System.out.println("before : " + s);
s.toUpperCase();
System.out.println("after : " + s);
```

Since String is immutable, this method call *cannot* change the variable!

```
before : Hello world!
after  : Hello world!
```

```
String s = "Hello world!";
System.out.println("before : " + s);
s = s.toUpperCase();
System.out.println("after : " + s);
```

```
before : Hello world!
after  : HELLO WORLD!
```

15

Final access modifier

- Final

- Declaring variable **final** means that you can assign value only once, in initializer or constructor

```
public class Counter
{
    private final String name;
    private int count;
    ...
}
```

This value doesn't change once the object is constructed

This value can change in instance methods

- Advantages

- Helps enforce immutability
- Prevents accidental changes
- Makes program easier to debug
- Documents that the value cannot not change

16

Equality: integer primitives

- Boolean operator ==
 - See if two variables are exactly equal
 - i.e. they have identical bit patterns
- Boolean operator !=
 - See if two variables are NOT equal
 - i.e. they have different bit patterns

```
int a = 5;
if (a == 5)
    System.out.println("yep it's 5!");
while (a != 0)
    a--;
```

This is a safe comparison since we are using an integer type.

17

Equality: floating-point primitives

- Floating-point primitives
 - i.e. double and float
 - Only an approximation of the number
 - Use == and != at your own peril

```
double a = 0.1 + 0.1 + 0.1;
double b = 0.1 + 0.1;
double c = 0.0;

if (a == 0.3)
    System.out.println("a is 0.3!");

if (b == 0.2)
    System.out.println("b is 0.2!");

if (c == 0.0)
    System.out.println("c is 0.0!");
```

b is 0.2!
c is 0.0!

18

Equality: floating-point primitives

- Floating-point primitives
 - i.e. double and float
 - Only an approximation of the number
 - Use == and != at your own peril

```
double a = 0.1 + 0.1 + 0.1;
double b = 0.1 + 0.1;
double c = 0.0;
final double EPSILON = 1e-9;

if (Math.abs(a - 0.3) < EPSILON)
    System.out.println("a is 0.3!");

if (Math.abs(b - 0.2) < EPSILON)
    System.out.println("b is 0.2!");

if (Math.abs(c) < EPSILON)
    System.out.println("c is 0.0!");
```

a is 0.3!
b is 0.2!
c is 0.0!

19

Equality: reference variables

- Boolean operator ==, !=
 - Compares bit values of remote control
 - Not the values stored in object's instance variables
 - Usually not what you want

```
Ball b = new Ball(0.0, 0.0, 0.5);
Ball b2 = new Ball(0.0, 0.0, 0.5);

if (b == b2)
    System.out.println("balls equal!");

b = b2;
if (b == b2)
    System.out.println("balls now equal!");
```

20

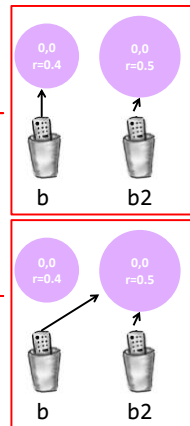
Equality: reference variables

```
Ball b = new Ball(0.0, 0.0, 0.4);
Ball b2 = new Ball(0.0, 0.0, 0.5);
```

```
if (b == b2)
    System.out.println("balls equal!");
```

```
b = b2;
if (b == b2)
    System.out.println("balls now equal!");
```

balls now equal



21

Object equality

- Implement equals() instance method
 - Up to class designer exactly how it works
 - Client needs to call equals(), not == or !=

```
public class Ball
{
    // See if this Ball is at the same location and radius
    // as some other Ball (within a tolerance of 1e-10).
    // Ignores the color.
    public boolean equals(Ball other)
    {
        final double EPSILON = 1e-9;
        return ((Math.abs(posX - other.posX) < EPSILON) &&
            (Math.abs(posY - other.posY) < EPSILON) &&
            (Math.abs(radius - other.radius) < EPSILON));
    }
    ...
}
```

22

Equality: String variables

- Boolean operator ==, !=
 - Compares bit values of remote control
 - A String is a reference variable
 - Does *not* compare text stored in the String objects

– Usually *not* what you want

```
String a = "hello";
String b = "hello";
String c = "hell" + "o";
String d = "hell";
d = d + "o";

if (a == b) System.out.println("a equals b!");
if (b == c) System.out.println("b equals c!");
if (c == d) System.out.println("c equals d!");
```

a equals b!
b equals c!

23

Handy String methods

- String is an object with lots of methods:

Method	
int length()	How many characters in this string
char charAt(int index)	char value at specified index
String substring(int start, int end)	Substring [start, end - 1] inclusive
boolean equals(String other)	Is this string the same as another?
boolean equalsIgnoreCase(String other)	Is this string the same as another ignoring case?
String trim()	Remove whitespace from start/end
String toLowerCase()	Return new string in all lowercase
String toUpperCase()	Return new string in all uppercase
int indexOf(String str)	Index of first occurrence of specified substring, -1 if not found
int indexOf(String str, int from)	Index of next occurrence of substring starting from index from, -1 if not found

24

Equality: String variables

- Check equality with `equals()` method
 - Each letter must be the same (including case)

```
String a = "hello";
String b = "hello";
String c = "hell" + "o";
String d = "hell";
d = d + "o";

if (a.equals(b)) System.out.println("a equals b!");
if (b.equals(c)) System.out.println("b equals c!");
if (c.equals(d)) System.out.println("c equals d!");
```

```
a equals b!
b equals c!
c equals d!
```

25

Summary

- Object oriented programming
- Data encapsulation
 - Important consideration when designing a class
 - Access modifiers decide who can see what
 - Immutability, preventing change to a variable
- Equality
 - Usually avoid `==` or `!=` with floating-point types
 - Usually avoid `==` or `!=` with reference types
 - Including `String`
 - Implement or use the `equals()` method

26