Building A Class

- Declarations
  - Objects versus variables
  - Scope of a declaration
- Java statements we know
- How to build a class
  - Price Tickets example
- Introduction to inheritance
  - How to extend classes?

Declarations

- `int i,h;` //sets aside storage for integer valued variables i and h
- `UStime t;`// creates a reference to an UStime object which will be dynamically created later using a `new` command

```
for(h=1;h<13;h++)
{ ...; t=new UStime(h,0);...}
```

//new command sets aside storage for a

// UStime object referred to by `t`

Declaration Scope

Example 1

```
for (int h = 1; h < 13; h++)
sum += h;
System.out.println(“h= “,h);//error
//because h no longer exists
```

scope of h

Example 2

```
int i; int sum=0;
for (i = 1; i <13; i++)
sum += i;
System.out.println(“i = “,i); //ok
```

scope of i

Java Statements - So Far

<statement> → <output-stmt> | <assign-stmt>
| return <expr> | <if-stmt> |
<method_call> | <for-loop>

- **Any of these can be used as the statement in the then or else clause of an if statement**

```
if(x>0||y<-1)System.out.println(
    “first case”); else y += 3;
if (num<15) foo(); else num = 0;
if (x<0) for (int i=0; i<9; i++)
        System.out.println (i);
```

Class Design

- *Coherence* - class should be concerned with one entity in a problem
  - e.g., crew members, planes
- *Separation of concerns* - can use several related classes to describe a complex entity
  - Geometric shapes involve use of Segment, Point, Circle, and Polygon classes
  - Object-oriented programming favors small methods with specific functionality, that interact with each other

Encapsulation

- ***Information hiding*** **- notion that a class only reveals what is necessary to use it**
  - **Methods a user needs to use**
  - **Instance variables whose values are needed**
  - **By convention, all methods and instance variables are `private`, unless designated `public`**
- **Objects should be available to users on a limited basis**
- **Protects against unwitting or intentional changes to objects**

Object-oriented Programming

- Class designer must know how her class will be used to write the necessary methods and define the necessary instance variables
- Class users must know class interface
  - Instance variables and method signatures (i.e., how to call each method and what kind of value it returns)
- *Data via methods* - class designer chooses what to reveal and what to conceal

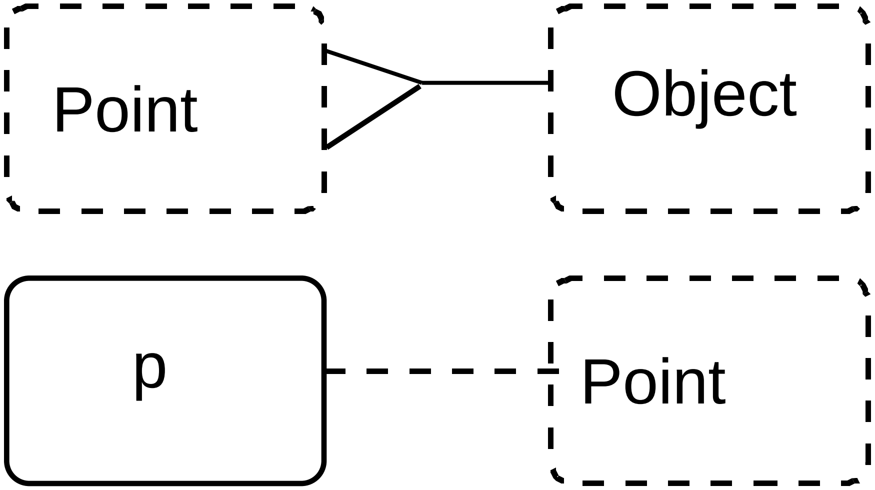
Object-oriented Programming

- Kinds of methods
  - Constructors - create new objects
  - Observers - `getX(), getY()` in Point class
  - Mutators - `setTolerance()` in Point class
  - Other - `distanceTo()` in Point class
- Facilitates building of large programs by many people
  - Protects data values
  - Separates namespaces of different pieces of program

How to test programs?

- Use `println`'s liberally while debugging
- Always test both the true and false branches of an if statement
- Pick data that will exercise different paths through a nested if statement
- Test boundary values
  - in Summation, test with `limit==0`
  - in NimState, test with `cnt==0`

Class Diagrams in Bishop, p75



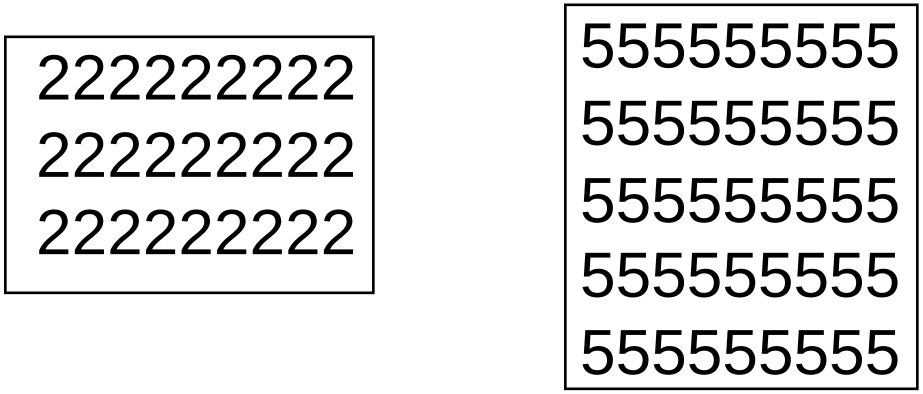
Point extends Object

p is a Point object

Gives a graphical depiction of relationship between classes, derivation of objects, and interaction of methods in classes.

Price Tickets Program, Bishop p 79ff

- **Problem: to produce tickets for an event on the computer**
  - **Need 1,2,5,10 denominations**
  - **Want easily distinguishable tickets**
- **Design idea: have tickets state 1, 2, 5, or 10 on their face and be of different sizes**



222222222
222222222
222222222

555555555
555555555
555555555
555555555
555555555

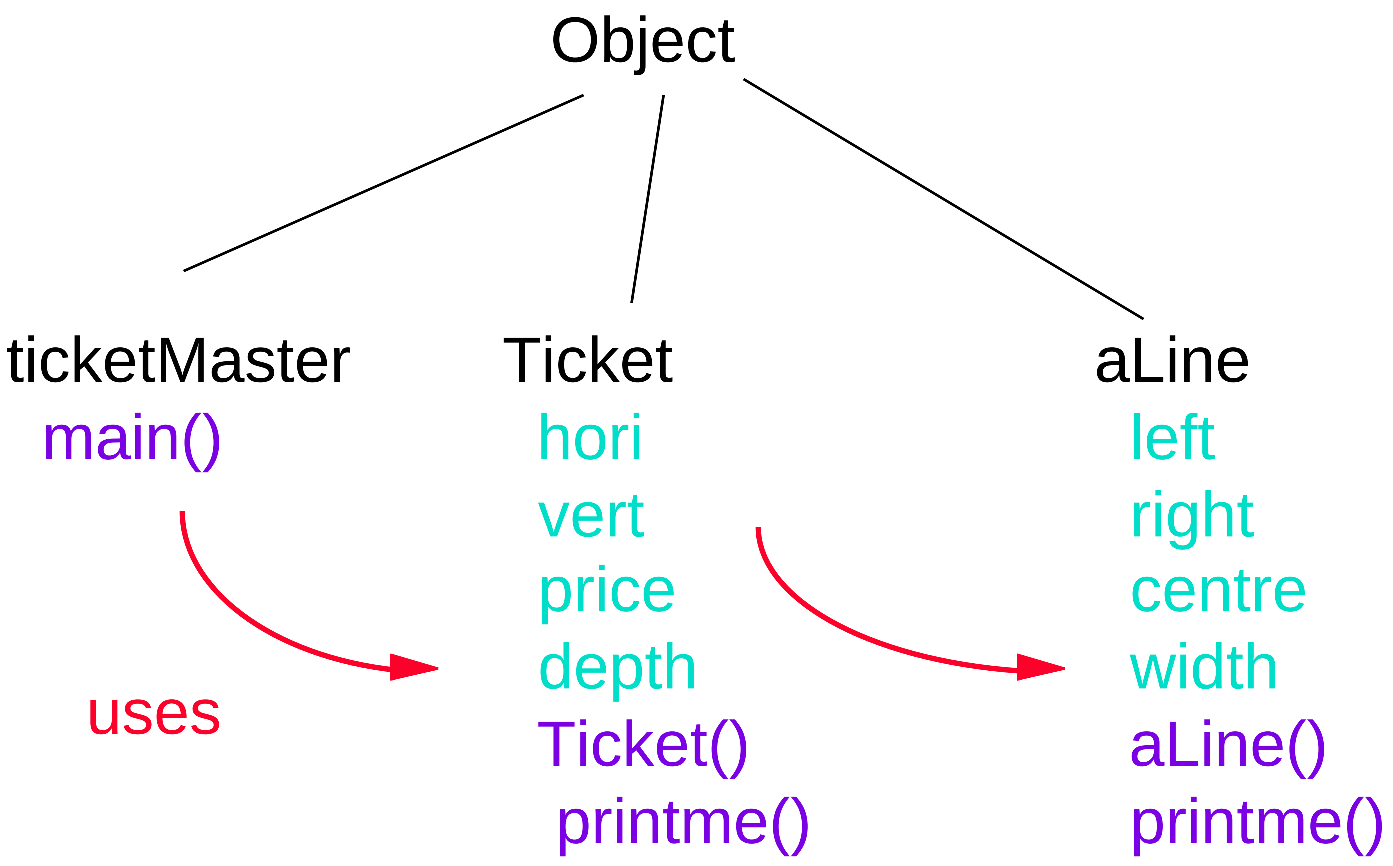
Ticket Class Design

- **Decompose problem into pieces**
- **Each ticket composed of 2 kinds of rows:**
  - **Top or bottom row**
  - **Middle row**
  - **Define `aLine` object to correspond to a row**
    - **Each `aLine` will have a left, center, right character**
    - **Each `aLine` will have a `printme()` method**
  - **Printing a ticket will consist of (possibly repeated) printing of the consituent `aLine` objects**

Ticket Class Design

- Decide to use 3 classes:
  - `ticketMaster` to print the tickets
  - `Ticket` to form the ticket
  - `aLine` to correspond to each row of a ticket
- Ticket construction
  - Decide to set width of ticket and vary the height
  - Need filler characters, top/bottom and sides characters

Class Structure



Object

ticketMaster
main()

uses

Ticket
hori
vert
price
depth
Ticket()
printme()

aLine
left
right
centre
width
aLine()
printme()

aLine Class

```
class aLine extends Object
    private String left,right,centre;
    private int width = 20;
    public aLine(String l,String c,
        String r){//constructor
        left = l;
        right = r;
        centre = c;
        }
    public void printme()
```

printme in aLine class

```
//prints a line of the ticket
public void printme(){
     System.out.print(left);
     for (int w=2; w < width; w++)
     System.out.print (centre);
     System.out.println (right);
}
```

Ticket class

```
class Ticket extends Object{
   private String hori, vert, price;
   private int depth;
   public Ticket(String h, String v,
     int d, String p){
     hori = h;//always use a length 1 string as h
     vert = v;//always use a length 1 string as v
     depth = d;
     price = p;//always use a length 1 string as p
}
```

Ticket class

```
void printme(){
 aLine topbot = new aLine(hori,hori,hori);
 aLine mid = new aLine (vert, price, vert);
 //code to print the ticket
 topbot.printme();
 int d;
 for (d=2; d<depth; d++)
 {mid.printme();}
 topbot.printme();
 System.out.println();//leave a blank line
  //between tickets to ease cutting apart
}
```

ticketMaster class

```
class ticketMaster extends Object{
  public static void main (String [] args){
  System.out.println();// skip a line
  Ticket t1 = new Ticket("+","!",10,"1");
  t1.printme();
  Ticket t2 = new Ticket("+","!",10,"2");
  t2.printme();
  Ticket t5 = new Ticket("+","!",15,"5");
  t5.printme();
  Ticket t10 = new Ticket("+","!",15,"0");
  t10.printme();
  System.out.println();// skip a line
}
}
```

Sample Output

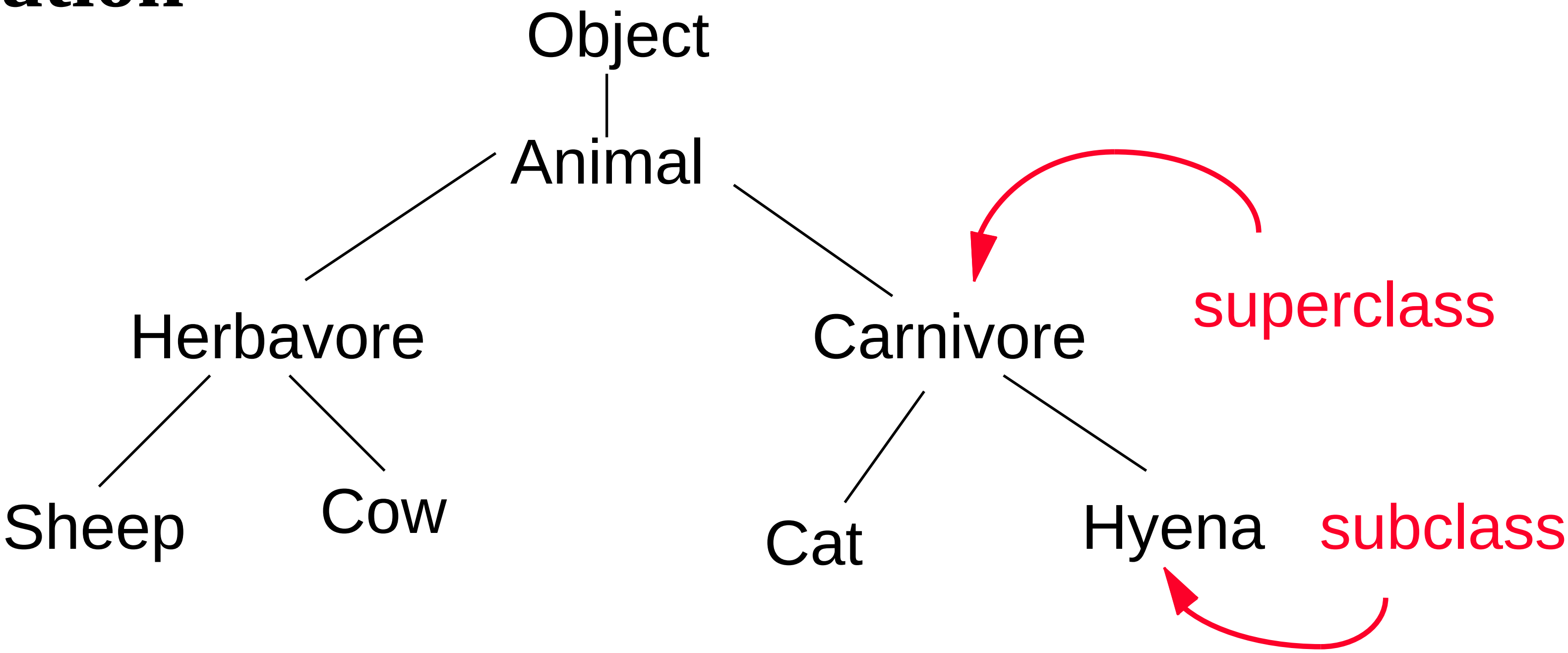
```
++++++++++++++++++++
!111111111111111111!
!111111111111111111!
!111111111111111111!
!111111111111111111!
!111111111111111111!
!111111111111111111!
!111111111111111111!
!111111111111111111!
++++++++++++++++++++
```

Possible Changes to Consider

- You decide you want to print the tickets, 3 across on each page
  - How to change the program?
  - Is this an easy change?
- You decide to change the design of the tickets themselves to incorporate the date of the event
  - How to change the program?
  - Is this an easy change?

Inheritance: Extending Classes

- Every class extends another (topmost class is Object)
- Often class hierarchy expresses an "is-a" relation



Object

Animal

Herbavore

Carnivore

superclass

Sheep

Cow

Cat

Hyena

subclass

Why Extend Classes?

- To share common attributes and methods
  - i.e., to share code
- To create collections of useful classes which divide the work of problem solution between them
  - Easier to maintain and test
- To create useful packages (Java word for libraries) which others can extend and specialize for their own needs

Method Placement

- Where to define method or instance variable(s) to be shared by instances of subclasses?
- needsWater() in Animal class
- forageAmount() in Herbavore class
- range() in Carnivore class
- livesLeft() in Cat class
- kitsInLitter with instances of Cat class

Method Lookup

- `chelsea` is a Cat object
- We want `chelsea.needsWater()`
  - First lookup `needsWater()` in Cat class
  - If not found, then lookup `needsWater()` in parent class to Cat, Carnivore
  - If not found, then lookup `needsWater()` in parent class of Carnivore, Animal.
  - Apply found method to receiver `chelsea`
- Lookup proceeds up the tree from class of object until a same-named method is found.