

Building Java Programs

Constructors, Encapsulation, `this`

reading: 8.2 - 8.3, 8.5 - 8.6

Object initialization: constructors

reading: 8.3

Initializing objects

- Currently it takes 3 lines to create a `Point` and initialize it:

```
Point p = new Point();  
p.x = 3;  
p.y = 8;                                // tedious
```

- We'd rather specify the fields' initial values at the start:

```
Point p = new Point(3, 8); // desired; doesn't work (yet)
```

- We are able to this with most types of objects in Java.

Constructors

- **constructor**: Initializes the state of new objects.

```
public type(parameters) {  
    statements;  
}
```

- runs when the client uses the `new` keyword
- no return type is specified;
it implicitly "returns" the new object being created
- equivalent to Python's `__init__`
- If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

Constructor example

```
public class Point {  
    int x;  
    int y;  
  
    // Constructs a Point at the given x/y location.  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public void translate(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
  
    ...  
}
```

Client code, version 3

```
public class PointMain3 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
    }
}
```

OUTPUT:

```
p1: (5, 2)
p2: (4, 3)
p2: (6, 7)
```

Common constructor bugs

1. Re-declaring fields as local variables ("shadowing"):

```
public Point(int initialX, int initialY) {  
    int x = initialX;  
    int y = initialY;  
}
```

- This declares local variables with the same name as the fields, rather than storing values into the fields. The fields remain 0.

2. Accidentally giving the constructor a return type:

```
public void Point(int initialX, int initialY) {  
    x = initialX;  
    y = initialY;  
}
```

- This is actually not a constructor, but a method named `Point`

Multiple constructors

- A class can have multiple constructors.
 - Each one must accept a unique set of parameters.
- *Exercise:* Write a `Point` constructor with no parameters that initializes the point to (0, 0).

```
// Constructs a new point at (0, 0).  
public Point() {  
    x = 0;  
    y = 0;  
}
```


Encapsulation

reading: 8.5 - 8.6

Abstraction

Don't need
to know
this

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND

BECAUSE I WANTED TO SEE A CAT
JUMP INTO A BOX AND FALL OVER.

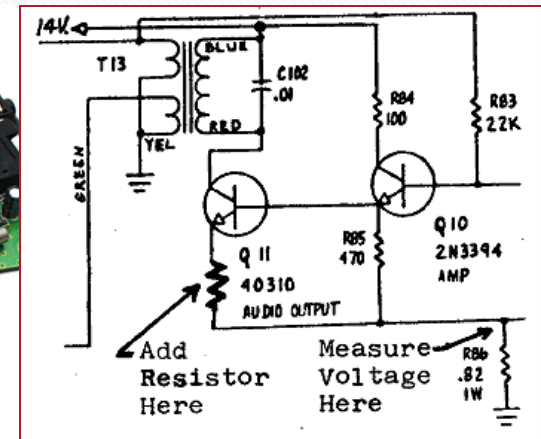
I AM A GOD.



Can focus
on this!!

Encapsulation

- **encapsulation:** Hiding implementation details of an object from its clients.
 - Encapsulation provides *abstraction*.
 - separates external view (behavior) from internal view (state)
 - Encapsulation protects the integrity of an object's data.



Private fields

- A field can be declared *private*.
 - No code outside the class can access or change it.

```
private type name;
```

- Examples:

```
private int id;  
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point  
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
                        ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}
```

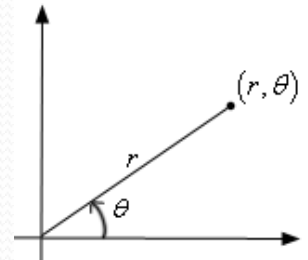
```
// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
 - A bank app forbids a client to change an `Account`'s balance.
- Allows you to change the class implementation.
 - `Point` could be rewritten to use polar coordinates (radius r , angle θ), but with the same methods.
- Allows you to constrain objects' state (**invariants**).
 - Example: Only allow `Points` with non-negative coordinates.



Point class, version 4

// A Point object represents an (x, y) location.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public double distanceFromOrigin() {  
        return Math.sqrt(x * x + y * y);  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setLocation(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
  
    public void translate(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```

Client code, version 4

```
public class PointMain4 {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point(5, 2);  
        Point p2 = new Point(4, 3);  
  
        // print each point  
        System.out.println("p1: + p1);  
        System.out.println("p2: + p2);  
  
        // move p2 and then print it again  
        p2.translate(2, 4);  
        System.out.println("p2: + p2);  
    }  
}
```

OUTPUT:

```
p1 is (5, 2)  
p2 is (4, 3)  
p2 is (6, 7)
```


The keyword `this`

reading: 8.7

this

- **this** : A reference to the implicit parameter.
 - *implicit parameter*: object on which a method is called
 - Equivalent to Python's `self`
- Syntax for using `this`:
 - To refer to a field:
`this.field`
 - To call a method:
`this.method(parameters) ;`
 - To call a constructor from another constructor:
`this(parameters) ;`

Variable names and scope

- Usually it is illegal to have two variables in the same scope with the same name.

```
public class Point {  
    int x;  
    int y;  
    ...  
  
    public void setLocation(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
}
```

- The parameters to `setLocation` are named `newX` and `newY` to be distinct from the object's fields `x` and `y`.

Variable shadowing

- An instance method parameter can have the same name as one of the object's fields:

```
// this is legal
public void setLocation(int x, int y) {
    ...
}
```

- Fields `x` and `y` are *shadowed* by parameters with same names.
- Any `setLocation` code that refers to `x` or `y` will use the parameter, not the field.

Avoiding shadowing w/ `this`

```
public class Point {  
    private int x;  
    private int y;  
  
    ...  
  
    public void setLocation(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

- Inside the `setLocation` method,
 - When `this.x` is seen, the *field* `x` is used.
 - When `x` is seen, the *parameter* `x` is used.

Inheritance

reading: 9.1

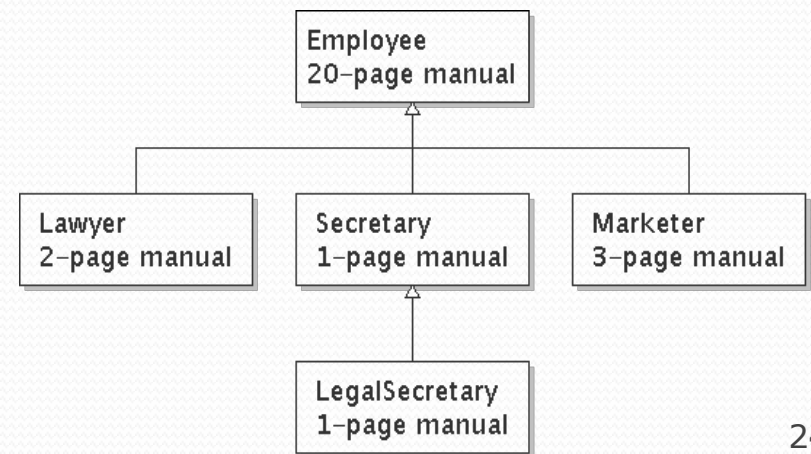
The software crisis

- **software engineering:** The practice of developing, designing, documenting, testing large computer programs.
- Large-scale projects face many issues:
 - programmers working together
 - getting code finished on time
 - avoiding redundant code
 - finding and fixing bugs
 - maintaining, reusing existing code
- **code reuse:** The practice of writing program code once and using it in many contexts.



Law firm employee analogy

- common rules: hours, vacation, benefits, regulations ...
 - all employees attend a common orientation to learn general company rules
 - each employee receives a 20-page manual of common rules
- each subdivision also has specific rules:
 - employee receives a smaller (1-3 page) manual of these rules
 - smaller manual adds some new rules and also changes some rules from the large manual

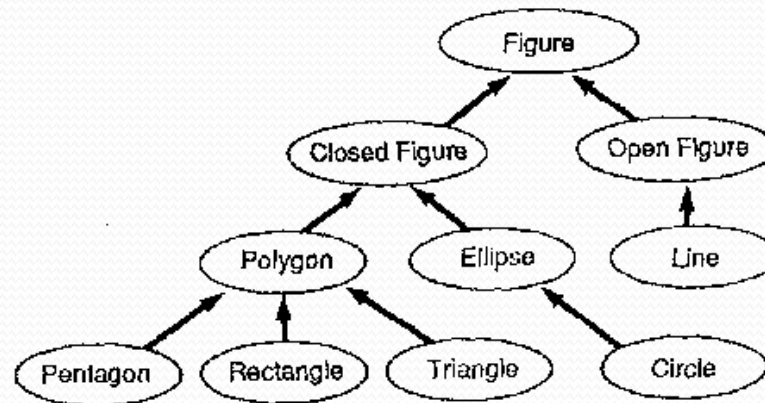


Separating behavior

- Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?
- Some advantages of the separate manuals:
 - maintenance: Only one update if a common rule changes.
 - locality: Quick discovery of all rules specific to lawyers.
- Some key ideas from this example:
 - General rules are useful (the 20-page manual).
 - Specific rules that may override general ones are also useful.

Is-a relationships, hierarchies

- **is-a relationship:** A hierarchical connection where one category can be treated as a specialized version of another.
 - every marketer *is an* employee
 - every legal secretary *is a* secretary
- **inheritance hierarchy:** A set of classes connected by is-a relationships that can share common code.



Employee regulations

- Consider the following employee regulations:
 - Employees work 40 hours / week.
 - Employees make \$40,000 per year, except legal secretaries who make \$5,000 extra per year (\$45,000 total), and marketers who make \$10,000 extra per year (\$50,000 total).
 - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
 - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.
- Each type of employee has some unique behavior:
 - Lawyers know how to sue.
 - Marketers know how to advertise.
 - Secretaries know how to take dictation.
 - Legal secretaries know how to prepare legal documents.

An Employee class

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;     // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;         // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";   // use the yellow form
    }
}
```

- Exercise: Implement class `Secretary`, based on the previous employee regulations. (Secretaries can take dictation.)

Redundant Secretary class

```
// A redundant class to represent secretaries.
public class Secretary {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;     // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;         // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";   // use the yellow form
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Desire for code-sharing

- `takeDictation` is the only unique behavior in `Secretary`.
- We'd like to be able to say:

// A class to represent secretaries.

```
public class Secretary {  
    copy all the contents from the Employee class;  
  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

Inheritance

- **inheritance:** A way to form new classes based on existing classes, taking on their attributes/behavior.
 - a way to group related classes
 - a way to share code between two or more classes

- One class can *extend* another, absorbing its data/behavior.
 - **superclass:** The parent class that is being extended.
 - **subclass:** The child class that extends the superclass and inherits its behavior.
 - Subclass gets a copy of every field and method from superclass

Inheritance syntax

```
public class name extends superclass {
```

- Example:

```
public class Secretary extends Employee {  
    ...  
}
```

- By extending `Employee`, each `Secretary` object now:
 - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
 - can be treated as an `Employee` by client code (seen later)

Improved Secretary code

```
// A class to represent secretaries.  
public class Secretary extends Employee {  
    public void takeDictation(String text) {  
        System.out.println("Taking dictation of text: " + text);  
    }  
}
```

- Now we only write the parts unique to each type.
 - Secretary **inherits** `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` **methods from** `Employee`.
 - Secretary **adds the** `takeDictation` **method**.