Abstraction and Encapsulation

"Make cohesive, intuitive abstractions"

Reduce Complexity

Make Sense

Tight, Little Classes

Coming up with appropriate abstract objects is one of the major challenges in OOD

Abstraction

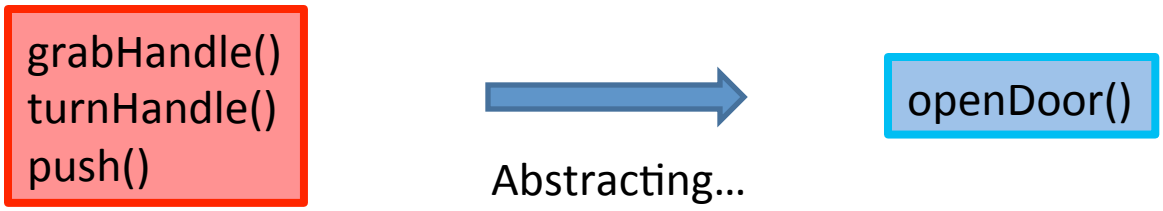
"The ability to view a complex operation in a simplified form."

- Wood, nails, glass, shingles -> House
- Houses, businesses, Tech, people -> Town

Business, Tech, people are all abstractions of aggregates.

Abstractions let us think about the essence or spirit of the data, rather than its implementation.

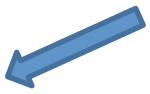
Abstractions provide interfaces to the complexity they abstract.



Interfaces

The first step to creating high-quality classes is creating a good interface—that class's public facing, how we interact with it.

"A class interface provides an abstraction of the implementation that's hidden behind the interface. The interface should offer a group of routines that clearly belong together."



`True`

thisDiscussion.interfaceWord != java.lang.interfaceWord

Review : ADTs

```
public class Queue {

    public void enqueue(Object QueueItem) {…}
    public Object dequeue() {…}
    public Object peek()  {…}
    public boolean isEmpty()  {…}

        //details
}
```

This is our abstraction of the queue, and provides our *interface* to the queue.
Reflects how we like to think about queues.
Using this, we don't need to care about what data structure this class manipulates to do Queue things, or how it pulls that off.
Every method works toward a consistent end.

Usually, we don't worry about pins or springs or strikeplates. We use the door.

Interface Presenting a Poor Abstraction

```
public class Program {

  public void initCommandStack() {…}

  public void pushCommand(Command c) {…}

  public  Command popCommand() {…}

  public void shutdownCommandStack() {…}

  public void initReportFormatting() {…}

  public void formatReport(Report r) {…}

  public void printReport(Report r) {…}

  public void initGlobalData() {…}

  public void shutdownGlobalData() {…}

   //private stuff
```

A more *consistent* abstraction

```
public class Program {

   public initializeProgram();

   public shutdownProgram();

   //private stuff
}
```

Move some of that junk to other (more appropriate) classes, and make the rest private for use by these two methods.

So how do we go about designing good abstract interfaces?

Simplify Complexity!

Remember, we don't expose the nodes or pointers in our linked lists. We expose a nice interface with things like insert() and retrieve(), because that's how we want to deal with linked lists.

Next: Bullets from Code Complete II

Classes represent ONE thing

A class should implement exactly ONE ADT. If you find it implementing more, or you can't figure out what ADT it implements, grab your thinking cap, some caffeine, and reorganize.

That's important.

Present *Consistent* Level of Abstraction

Is it a List, an EmployeeList, and what about those names?

```
class EmployeeList extends
   ListContainer {

public void add(Employee e) {…}
public void remove(Employee e) {…}
public Employee nextItemInList()
   {…}
public Employee firstItem() {…}
public Employee LastItem() {…}
```

Consistent Abstraction

```
class EmployeeList {

public void add(Employee e) {…}
public void remove(Employee e) {…}
public Employee next() {…}
public Employee previous() {…}
public Employee first() {…}
public Employee last() {…}

// stuff

private ListContainer theList;

}
```

Mixed levels of abstraction will contribute to the degredation of code through maintenance until it becomes impossible to understand.

Cater to Intuition: Provide Services in Pairs That Make Sense

- Many operations have corresponding opposites (on/off, add/remove, etc).
- If you have one of these, check yourself to make sure you don't need the other!
- Sometimes you don't want the other, because of the behavior you wish to provide.
- "Code, not mind games," right?

Keep Information Related

- Classes represent how many things?

```
public class CircleAnimate extends Circle {
  private Color currentColor;
  private Color startingColor;
  private int animSpeedInMS;

  public CircleAnimate(float x, float y, float r,
    Color startingColor, int animSpeed) {

    super(x,y,r);
    this.startingColor = startingColor;
    this.animSpeedInMS = animSpeed;
    this.currentColor = startingColor;
    }

  public void draw(Graphics g, float deltaT) {
        calcCurrentColor(deltaT);
        //draw to g with CurrentColor;
    }

  private Color calcCurrentColor(float deltaT) {
    //maths w/rgb values or some nonsense
    return curColor;
  }
  //private methods and stuff
}
```

This class represents a circle whose color changes over time.

Is all of this logic even at the same level of abstraction?

Besides that, a good deal of this class's ("isa" Circle, btw) logic deals with color manipulation.

This is a "siamese" class. Time to operate.

How About…

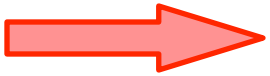
```
public class CircleAnimate extends Circle {
  private ColorCycler cycler;

  public CircleAnimate(float x, float y, float r,
                       Color startingColor, int animSpeedMS) {

    super(x,y,r);
    cycler = new ColorCycler(startingColor, animSpeedMS);
  }

  public void draw(Graphics g, float deltaT) {

    cycler.update(deltaT);
    //draw to g using cycler.getCurrentColor()
  }
}

public class ColorCycler {
  private Color currentColor;
  private int animSpeed;
  private float stateTime;

  public ColorCycler(Color startingColor, int animSpeedMS){…}
  public update(float deltaT){…}
  //other sensible interface methods
}
```

Beware of

Erosion of Interface under Modification

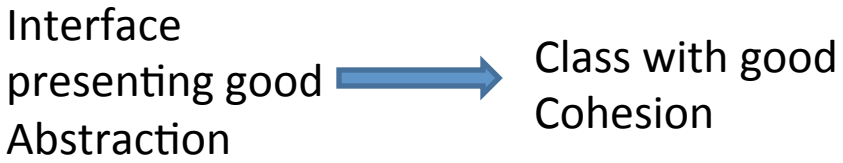
Our problems balloon, of course. Classes get added to and extended. Sometimes we have operations that don't *quite* fit, but it seems a pain in the arse to implement them any other way. Say we want to perform more functions with our Employee class:

```
public class Employee {
  Employee(){…}
  Employee(FullName name,
           String address,
           String phoneNumber,
           TaxID  taxIDNumber,
           JobClass jobClass
  ){…}
  //getters
  //private stuff
}
```



```
public class Employee {
  Employee(){…}
  Employee(FullName name,
           String address,
           String phoneNumber,
           TaxID  taxIDNumber,
           JobClass jobClass
  ){…}
  //getters

  boolean isJobClassValid(JobClass jc){…}
  //..for zip and phone

  SqlQuery getEmployeeCreationQuery(){…}
  //..for modify and retrieve
  //private stuff
}
```

Keep Interface Consistent!



We saw in the last slide that consistency had started to degrade and things began to make less sense.

If class has weak cohesion and you aren't sure how to correct it, see how consistent the abstraction is.

Encapsulation Enforces Abstraction

"The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules."

- Don't expose member data publicly.
- Use getters/setters. That way, nobody will break if you change your floats to a mouse circus or database retrieval.
- If data needs to be validated, your setters/ constructor can do that.
- No need for comments like "Don't put zero here."

Read-Time Convenience > Write-Time Convenience

- Your interfaces' abstractions will degrade, lose consistency.
- That leads to the dark side, where you write mind games rather than code.
- If someone can't figure out how to use your class based on its interface documentation, you should modify that interface and its docs.
- Take the time to make your code make sense. Do it for YOU, and all mankind.