“I prefer Agassiz in the abstract, rather than in the concrete.”
Back to the Property Example

- There are properties on a monopoly board
- Railroads, Utilities, and Streets are kinds of properties
- One behavior we want in Property is the getRent method
- problem: How do I get the rent of something that is “just a Property”? 
The Property class

class Property {
    private int cost;
    private String name;

    public int getRent() {
        return hmmmmmm??????
    }
}

Doesn’t seem like we have enough information to get the rent if all we know is it is a Property.
Potential Solutions

1. Just leave it for the sub classes.
   ▸ Have each sub class define getRent()

2. Define getRent() in Property and simply return -1.
   ▸ Sub classes override the method with more meaningful behavior.
public void printRents(Property[] props) {
    for(Property p : props)
        System.out.println(p.getRent());
}

Property[] props= new Property[2];
props[0] = new Railroad("NP", 200, 1);
props[1] = new Utility("Electric", 150, false);
printRents(props);

What is result of above code?
A. 200150   B. different every time
C. Syntax error   D. Class Cast Exception
E. Null Pointer Exception
Fix by Casting

// no getRent() in Property
public void printRents(Property[] props)
{
    for(Property p : props)
    {
        if(p instanceof Railroad)
            System.out.println( ((Railroad)).getRent() );
        else if(p instanceof Utility)
            System.out.println( ((Utility)p).getRent() );
    }
}

Property[] props= new Property[2];
props[0] = new Railroad("NP", 200, 1);
props[1] = new Utility("Electric", 150, false);
printRents( props);

What happens as we add more sub classes of Property?

What happens if one of the objects is just a Property?
Fix with Dummy Method

// getRent() in Property returns -1

color]} Fix with Dummy Method

public void printRents(Property[] props) {
  for(Property p : props)
    System.out.println(p.getRent());
}

Property[] props= new Property[2];
props[0] = new Railroad("NP", 200, 1);
props[1] = new Utility("Electric", 150, false);
printRents( props);

What happens if sub classes don't override getRent()?

Is that a good answer?
A Better Fix

- We know we want to be able to find the rent of objects that are instances of `Property`.
- The problem is we don’t know how to do that if all we know is it a `Property`.
- Make `getRent` an abstract method.
- Java keyword.
Making getRent Abstract

class Property {

    private int cost;
    private String name;

    public abstract int getRent();
    // I know I want it.
    // Just don’t know how, yet...

}

Methods that are declared abstract have no body
an undefined behavior.

All methods in a Java interface are abstract.
Problems with Abstract Methods

Given `getRent()` is now an abstract method
what is wrong with the following code?

```java
Property s = new Property();
System.out.println(s.getRent());
```
Undefined Behavior = Bad

- Not good to have undefined behaviors
- If a class has 1 or more abstract methods, the class must also be declared abstract.
  - version of Property shown would cause a compile error
- Even if a class has zero abstract methods a programmer can still choose to make it abstract
  - if it models some abstract thing
  - is there anything that is just a “Mammal”?
Abstract Classes

public abstract class Property {

    private int cost;
    private String name;

    public abstract double getRent();
    // I know I want it.
    // Just don’t know how, yet…

}
// Other methods not shown

If a class is abstract the compiler will not allow constructors of that class to be called
Property s = new Property(1, 2); //syntax error
Abstract Classes

- In other words you can’t create instances of objects where the lowest or most specific class type is an abstract class
- Prevents having an object with an undefined behavior
- Why would you still want to have constructors in an abstract class?
- Object variables of classes that are abstract types may still be declared

```javascript
Property s; //okay
```
Sub Classes of Abstract Classes

- Classes that extend an abstract class must provided a working version of any abstract methods from the parent class
  - or they must be declared to be abstract as well
  - could still decide to keep a class abstract regardless of status of abstract methods
Implementing getRent()

```java
public class Railroad extends Property {

    private static int[] rents = {25, 50, 10, 200};

    private int numOtherRailroadsOwned;

    public double getRent() {
        return rents[numOtherRailroadsOwned];
    }

    // other methods not shown
}
```
public class Utility extends Property {

    private static final int ONEUTILITYRENT = 4;
    private static final int TWOUTILITYRENT = 10;

    private boolean ownOtherUtility;

    public Utility(String n, int c, boolean other) {
        super(n, c);
    }

    public String toString() {
        return "Utility. own other utility? " + ownOtherUtility;
    }

    public int getRent(int roll) {
        return ownOtherUtility ? roll * TWOUTILITYRENT : roll * TWOUTILITYRENT;
    }
}
Polymorphism in Action

// getRent() in Property is abstract

public void printRents(Property[] props) {
    for(Property p : props)
        System.out.println(p.getRent());
}

• Add the Street class. What needs to change in printRents method?
• Inheritance is can be described as new code using old code.
• Polymorphism can be described as old code using new code.
Comparable in Property

```java
public abstract class Property
  implements Comparable<Property> {

  private int cost;
  private String name;

  public abstract int getRent();

  public int compareTo(Property other) {
    return this.getRent() - otherProperty.getRent();
  }
}
```
We suggested having a list interface

```java
public interface IList<E> extends Iterable<E> {
    public void add(E value);
    public int size();
    public E get(int location);
    public E remove(int location);
    public boolean contains(E value);
    public void addAll(List<E> other);
    public boolean containsAll(List<E> other);
}
```
Data Structures

When implementing data structures:

- Specify an interface
- Create an abstract class that is *skeletal implementation* interface
- Create classes that extend the skeletal interface