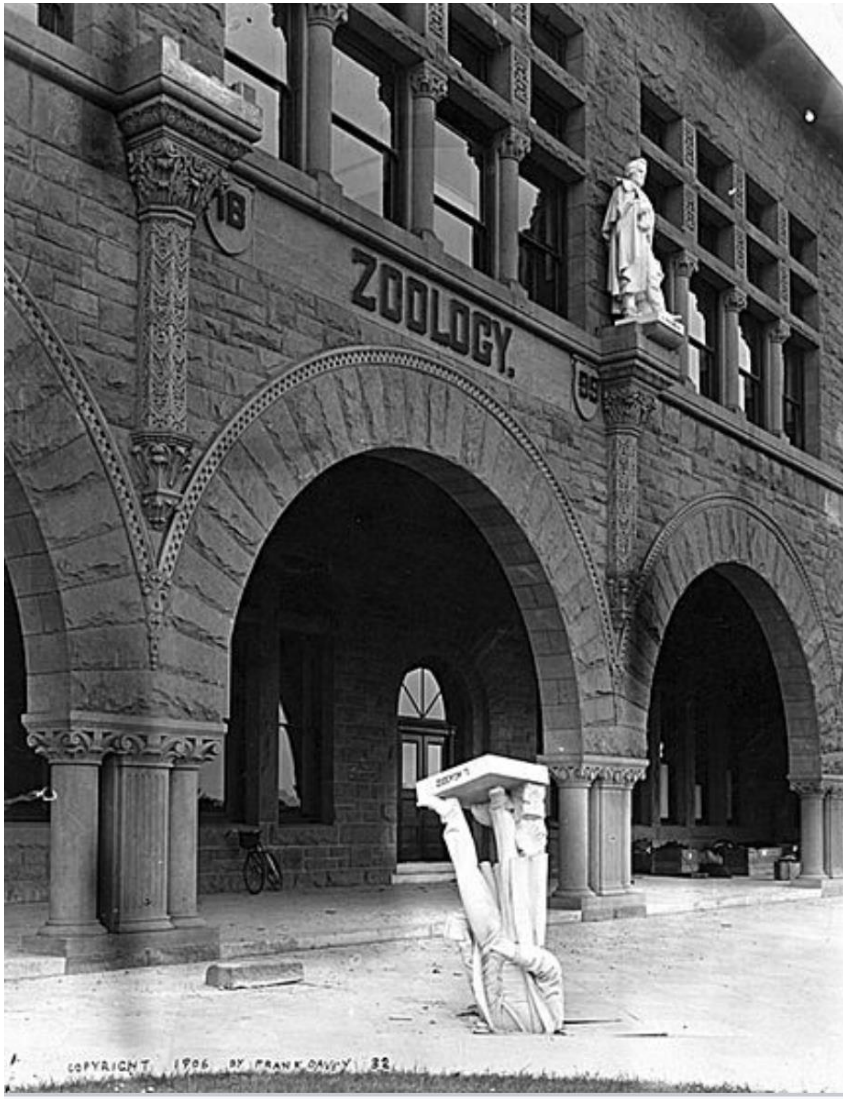
# Topic 10

# Abstract Classes

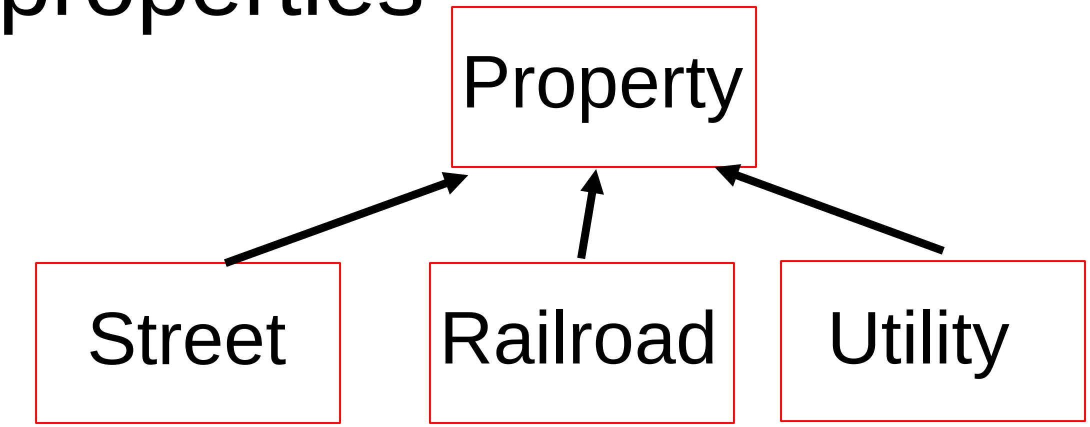
"I prefer Agassiz in the abstract, rather than in the concrete."

- Statue of Biologist Louis Agassiz that fell from a ledge on the Stanford Quad during the 1906 San Francisco earthquake.



# Back to the Monopoly Property Example

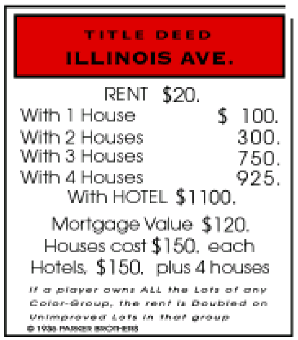
- There are properties on a monopoly board
- Railroads, Utilities, and Streets are kinds of properties


Property

Street

Railroad

Utility


TITLE DEED

ILLINOIS AVE.

RENT $20.

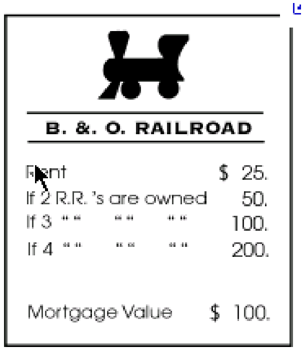
| | |
|---|---|
| With 1 House | $ 100. |
| With 2 Houses | 300. |
| With 3 Houses | 750. |
| With 4 Houses | 925. |

With HOTEL $1100.

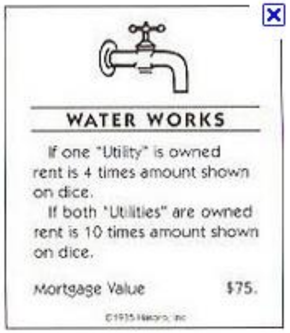
Mortgage Value $120.
Houses cost $150. each
Hotels, $150. plus 4 houses

*If a player owns ALL the Lots of any Color-Group, the rent is Doubled on Unimproved Lots in that group*

© 1935 PARKER BROTHERS


B. &. O. RAILROAD

| | |
|---|---|
| Rent | $ 25. |
| If 2 R.R. 's are owned | 50. |
| If 3 " " " " " " | 100. |
| If 4 " " " " " " | 200. |
| Mortgage Value | $ 100. |


WATER WORKS

If one "Utility" is owned rent is 4 times amount shown on dice.

If both "Utilities" are owned rent is 10 times amount shown on dice.

Mortgage Value $75.

©1935 Hasbro, Inc

# A getRent Behavior

- 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

```
public class Property {

    private int cost;
    private String name;

    public int getRent() {
         return hmmmmm??????;
    }
```

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.

# Leave it to the Sub - Classes

```
// no getRent() in Property
// Railroad and Utility DO have getRent() methods

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);
```

**Clicker 1** - 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) p).getRent());
      else if (p instanceof Utility)
         System.out.println(((Utility) p).getRent());
      else if (p instanceof Street)
          System.out.println(((Street) p).getRent())
   } // GACK!!!!
}
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 Placeholder Return

```
// getRent() in Property returns -1

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 get 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

```
public class Property {

    private int cost;
    private String name;

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

}
```

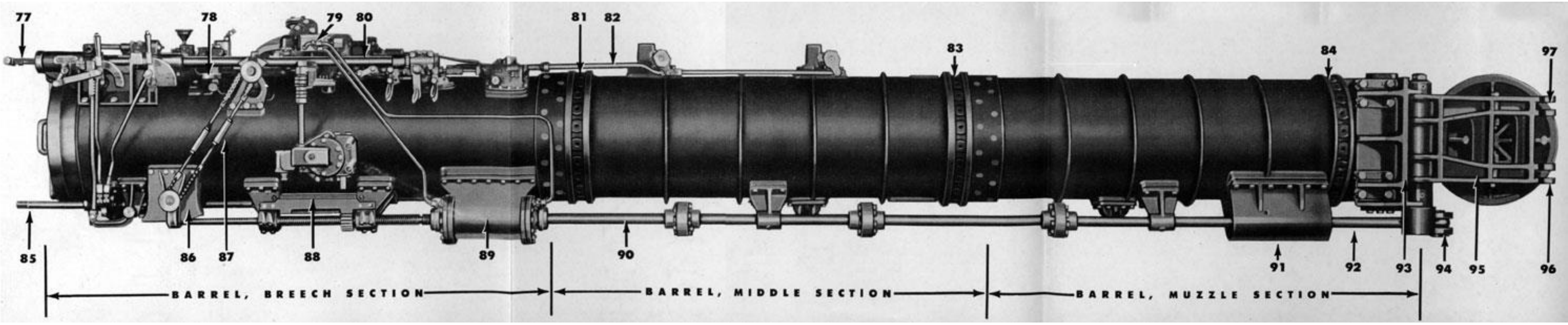
Methods that are declared abstract have no body an undefined behavior.

All non-default 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?

```
Property p = new Property();
System.out.println(p.getRent());
```



If things can go wrong with a tool, provide safeguards to prevent that from happening.

# 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 Safety

1. A class with one or more abstract methods must be declared abstract.
   - Syntax error if not done.
   - Can still decide to make class abstract even if no abstract methods.
2. Objects of an abstract type cannot be instantiated.
   - Just like interfaces
   - Can still declare variables of this type
3. A subclass must implement all inherited abstract methods or be abstract itself.

# 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

```
Property p; //okay
```

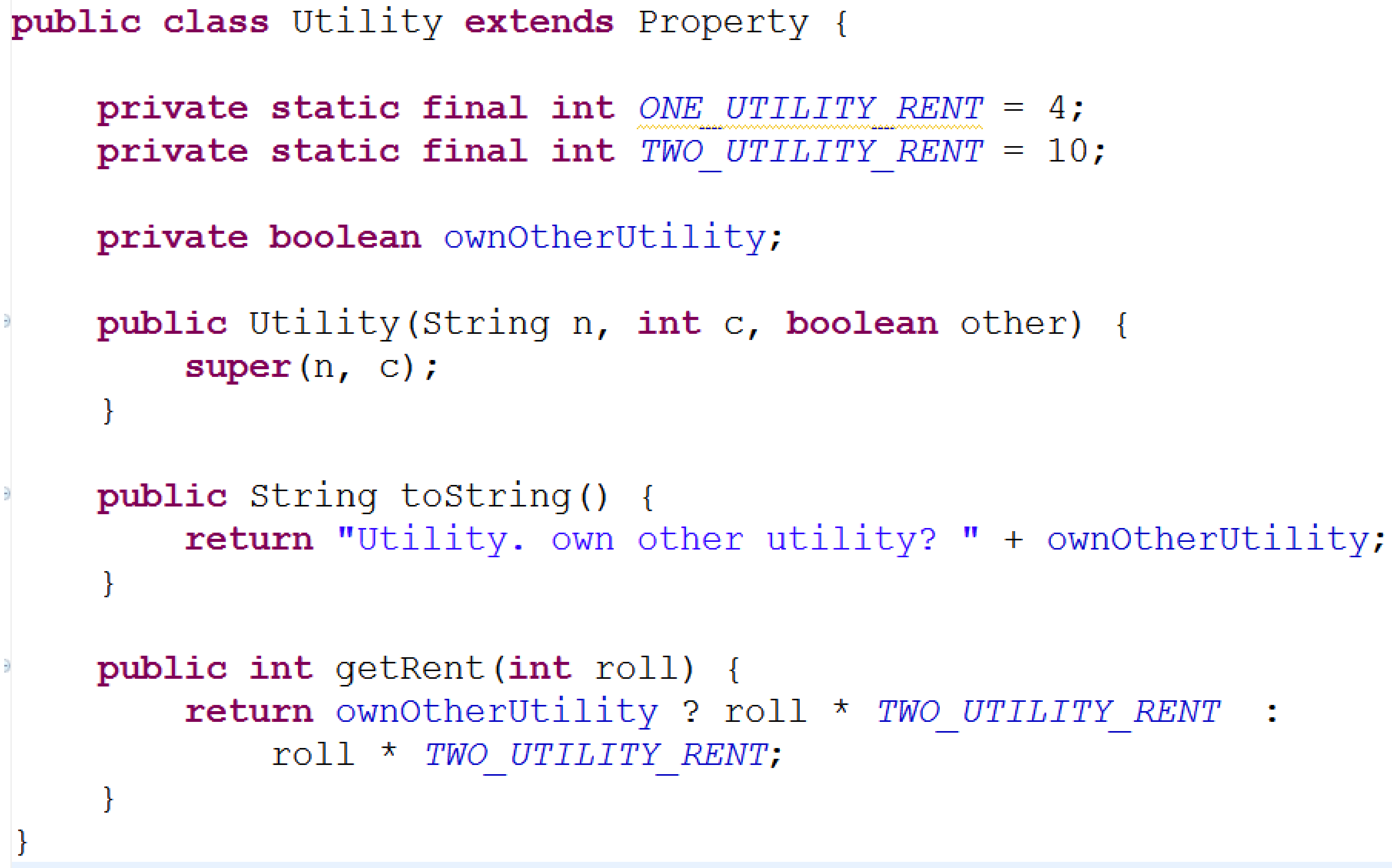
# Sub Classes of Abstract Classes

- Classes that extend an abstract class must provided a working version of any and all 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()

```
public class Railroad extends Property {

   private static int[] rents
          = {25, 50, 100, 200};

   private int numOtherRailroadsOwned;

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

   // other methods not shown
}
```

# A Utility Class



```java
public class Utility extends Property {

    private static final int ONE_UTILITY_RENT = 4;
    private static final int TWO_UTILITY_RENT = 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 * TWO_UTILITY_RENT  :
            roll * TWO_UTILITY_RENT;
    }
}
```

# 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.
- **Koan of Polymorphism: Polymorphism can be described as old code reusing new code.**

# Comparable in Property

```
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();
    }
}
```

# Back to Lists

- We suggested having a list interface

```
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(IList<E> other);
  public boolean containsAll(IList<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

```
public boolean contains(E val) {
    for (E e : this)
        if val.equals(e)
            return true;
return false
```