Object Oriented Programming and Design in Java

Session 13

Instructor: Bert Huang

Announcements

- Homework 3 out. Due **Monday**, Apr. 5th
- Midterm solutions and grades posted
- Office hour change starting tomorrow:
  - Lauren 11 AM -1 PM, Friday
  - Bert 2-4 PM Wednesday
  - Yipeng 4-6 PM Wednesday

Schedule

Sun	Mon	Tue	Wed	Thu	Fri
John 1-3	Class 11-12:15		Class 11-12:15 Bert 2-4 Yipeng 4-6		Lauren 11-1

Review

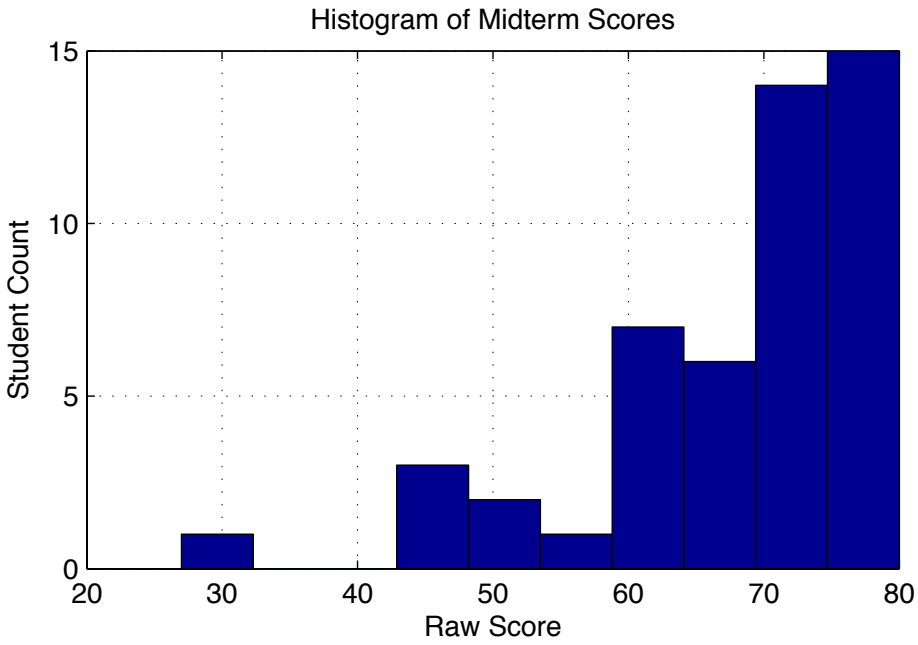
- Java Types
  - Arrays, enums
- The Object Class
  - toString(), equals(), clone(), hashCode()
- Hash tables

Today's Plan

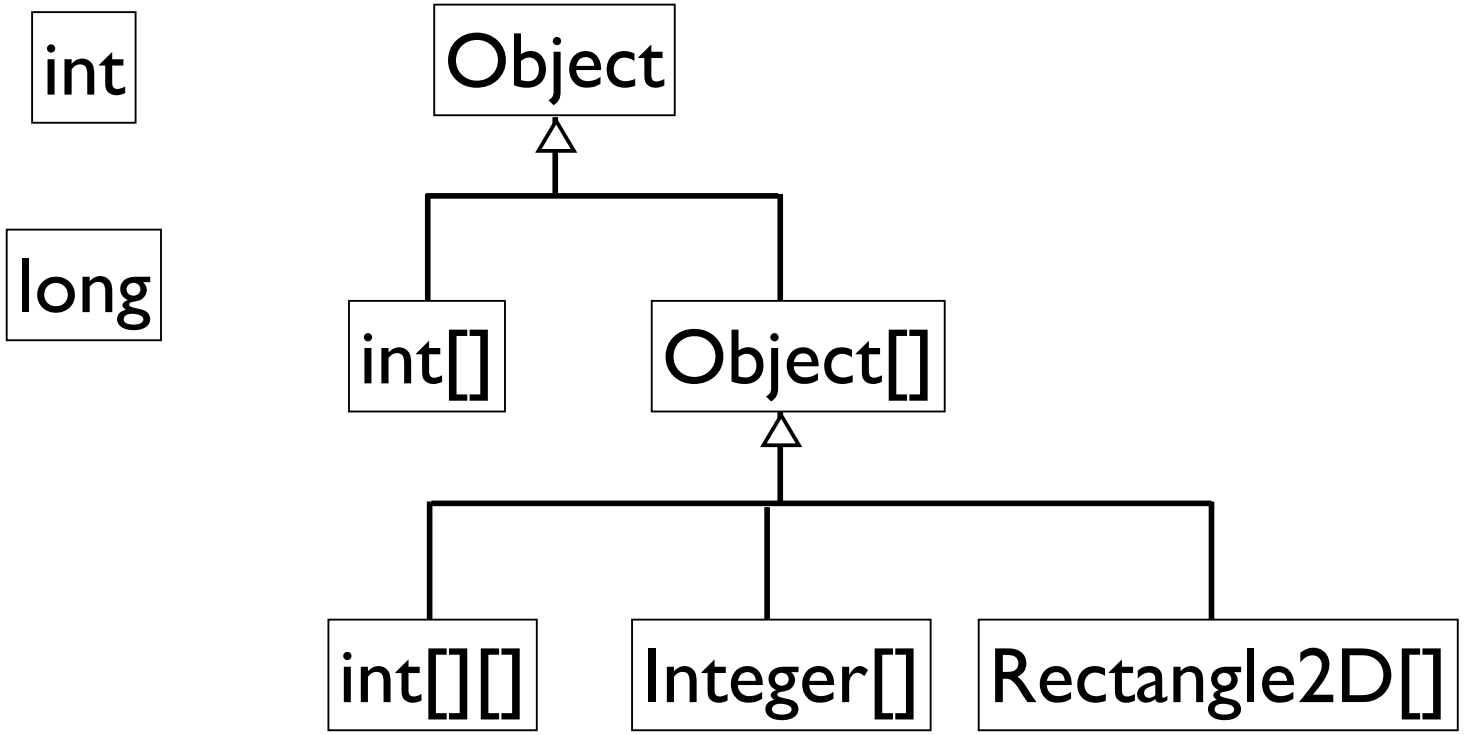
- Go over midterm
  - Statistics, common mistakes
- Cloneable
- Serializable
- Reflection
- Class, Method, Field objects

Midterm Statistics

- 8 problems,
  10 points each
- Average 68/80, 85%



Type Hierarchy



Checked Exceptions

- ```
  class Vehicle {
      travel(City d) throws DistanceException { ... }
  }
  ```
- ```
  class Airplane extends Vehicle {
      travel(City c) throws DistanceException,
                            NoAirportException { ... }
  }
  ```
- ```
  Vehicle a = new Airplane();
  try {
      a.travel(Boston);
  } catch (DistanceException e) { }
  ```
  ```

Accessors and Mutators

- The utility of defining accessors and mutators depends on perception
- Does the method *sound* like an accessor?
  - Any unexpected changes are side effects
- If it sounds like a mutator, does it change what it sounds like it should change?
  - If not, unexpected change is a side effect

java.lang.Object

- All class variables extend the base Java class, java.lang.Object
- Object contains a few implemented methods:
  - String toString()
  - boolean equals(Object other)
  - Object clone()
  - int hashCode()

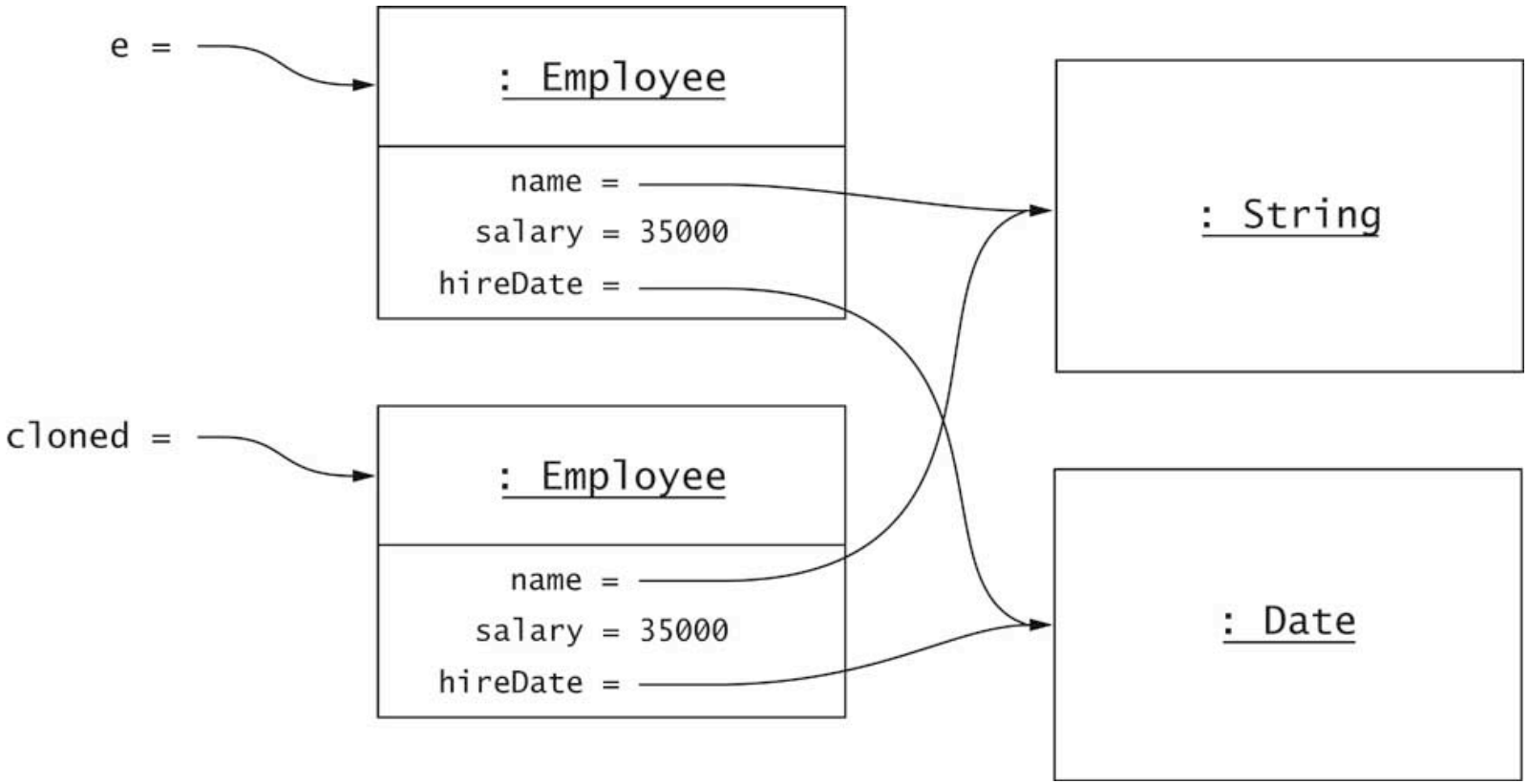
clone()

- Clone is meant to be used when you want an actual copy of an Object instead of another reference
- (x.clone() != x) && (x.clone().equals(x))
- Default clone() copies all fields
- clone() is a protected method by default and can only be used if your subclass implements the Cloneable interface

The Cloneable Interface

- Tagging interface; contains no methods
- But Object uses it to check that calls to clone() are only on Cloneable objects
  - otherwise throws `CloneNotSupportedException`
- Must be careful; copying fields may still share common aggregated objects

Shallow vs. Deep Copy

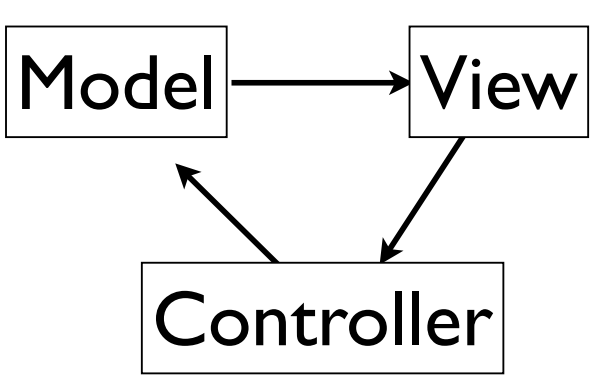


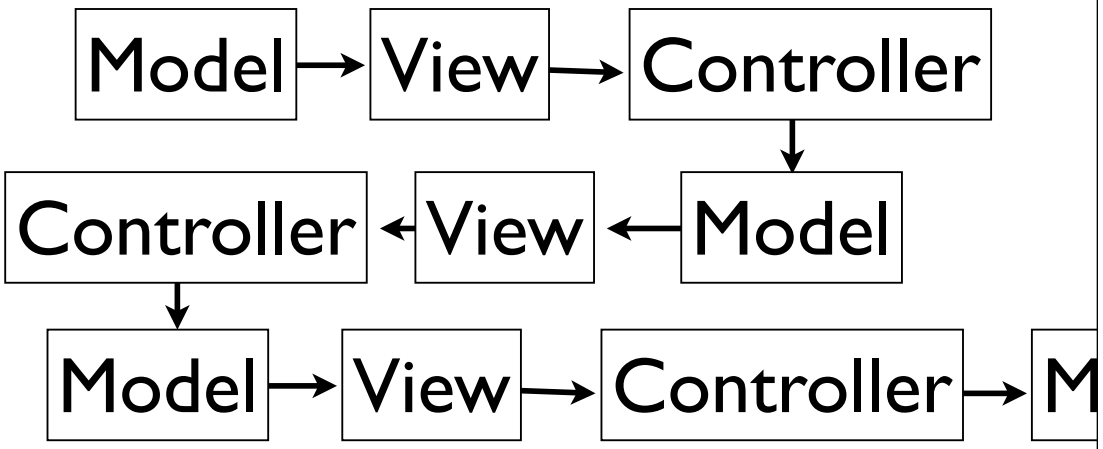
Shallow vs. Deep Copy

- Cloning all fields won't clone any Class variables, like String or Date
- Then if the clone modifies the Date object, the original's Date gets changed
- Instead, we can recursively clone all mutable class objects

Deep Copy Recursion

- Recursively cloning fields can cause very bad things to happen
- Consider MVC objects that store references to each other



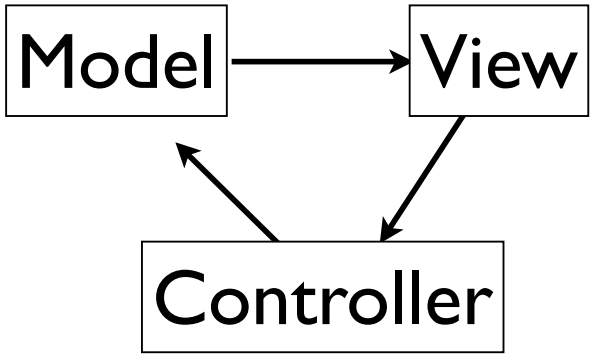


Serializable Interface

- Another tagging interface
- Tells Java that a class is able to be written to file using `ObjectOutputStream`
- `new ObjectOutputStream(FileOutputStream f)`
- `ObjectOutputStream.writeObject(Serializable s)`
- Writes the object and *all its fields and referenced objects* to file
- Fields not to be written can be marked with keyword `transient`

Serializing Circular Structure

- Files assign serial numbers to Objects
- So circular structure can be saved without infinite recursion
- But we can only load one object
- Let's test this with an experiment



```
import java.io.*;

public class SerializationTest
{
   public static class Link implements Serializable
   {
      public Link next;
      public String name;
   }

   public static void main(String [] args)
   {
      Link A = new Link();
      Link B = new Link();
      A.name = "Batman";
      B.name = "Robin";

      A.next = B;
      try {
         ObjectOutputStream out = new ObjectOutputStream(
               new FileOutputStream("A.dat"));
         out.writeObject(A);
         out.close();
```

```
        A.next = B;
        try {
           ObjectOutputStream out = new ObjectOutputStream(
                 new FileOutputStream("A.dat"));
           out.writeObject(A);
           out.close();

           ObjectInputStream in = new ObjectInputStream(
                 new FileInputStream("A.dat"));

           B.name = "Superman";

           Link C = (Link) in.readObject();
           in.close();

           System.out.println("Read " + C.name);
           System.out.println("C.next = " + C.next.name);

        } catch(Exception e) {
           e.printStackTrace();
        }
     }
  }
```

```
Read Batman
C.next = Robin
```

Reflection

- Reflection is the ability of a program to find out about the capabilities of objects at *runtime*
- Java provides these classes to describe features of types:
  - Class, Package, Field, Method, Constructor, Array

Class Objects

- `(obj instanceof Shape)` only tells you if variable `obj` is a subtype of `Shape`
- If you want to know the exact class, you need to use a class object `obj.getClass()`
- JVM keeps one object of each known class, so use == operator to check class equality
- Can also directly get class objects by `Shape.class == obj.getClass()`

Class Attributes

- `Shape.class.getSuperClass() //returns Class`
- `Shape.class.getInterfaces() //returns Interface[]`
- `Shape.class.getPackage() //returns Package`
- `Shape.class.getDeclaredMethods() //returns Method[]`
- `Shape.class.getDeclaredFields() //returns Field[]`
- `Shape.class.getDeclaredConstructors()//Constructor[]`

Method Objects

- `m.getName(), m.getParameterTypes()`
- Also can get Method objects using
  `Method m = getDeclaredMethod(name, params, ...)`
- Then call methods with `m.invoke(params)`
- Rarely useful, but can be used to build general testing programs

Field Objects

- `Class getType()`
- `int getModifiers() // binary flags`
  - `Modifier.isAbstract(), isPrivate(), isFinal(), etc`
- `Object get(Object obj) // reads field`
- `void set(Object obj, Object value)`
- `void setAccessible(boolean b) // changes whether private`
  `// fields are accessible. Wait, what???!`

  - Java programs allow this by default, applets and servlets do not.

```
public static void main(String [] args)
{
   PasswordChecker pc = new PasswordChecker("secretpassword");

   Class c = pc.getClass();
   Field f;
   try {
      f = c.getDeclaredField("password");

      f.setAccessible(true);

      String stolenPassword = (String) f.get(pc);

      System.out.println("Old password was " + stolenPassword);

      f.set(pc, "malicious_password");
      System.out.println("Trying old password. " +
            pc.checkPassword("secretpassword"));
   }
   catch (SecurityException e) { }
   catch (NoSuchFieldException e) { }
   catch (IllegalArgumentException e) { }
   catch (IllegalAccessException e) { }
}
```

```
Old password was secretpassword
Trying old password. false
```

Why Reflection?

- Pros:
  - Extremely powerful way to dynamically retrieve information about Classes by name
  - Retains Object Oriented ideas
  - Allows for meta-programs (like JUnit)
- Cons:
  - Can break encapsulation
  - Some anti-polymorphism ideas, e.g., checking an actual class type instead of trusting hierarchy

Reading

- Horstmann Ch. 7.5 - 7.7