Announcement

• Homework released
  – Start early
  – Test often

• Goal:
  – Waste time playing othello 😊
  – Learn to work with objects
  – Learn to use AI
  – Learn to implement graphics (java framework)
Homework hints

• Start early
• Work with your UML sketches
• Don’t be afraid of updating/changes

• Focus:

Graphical displays

• If you are working on your local machine no need for this

• If you want to work on cunix you need to run a local xserver to display graphics on your end
  1. Putty needs x tunneling turned on
  2. Download and run xwin32 on your machine
Outline

- Hashing ..plenty of details
- Copying (again)
- Working with unknown objects ..reflection
- Generic Objects inner working

- Reading for today: 7.3-7.8
- For next time 7.8 - end

Quick question

- Given a set of dictionary word

- How would you build a spell checker ?

- Take a second to describe some pseudo code

- How fast does this run?
HashTable

• what is happening in the background of the following code:

ht.put("Sh553","Shlomo Hershkop");

if(ht.containsKey("sh553")){
    String info = ht.get("sh553");
}

• What is the difference between contains and containsKey??

Now we can begin to ask

• What is happening in our black box?

• What algorithm is being used?

• Is it the best/fastest or can we improve this.
Side issue

• When is it worth actually trying to improve your code?

Rule of improvement

• Informally:

• When we are trying to improve something, we can only improve it by the percentage of improved code contribution

• Example: if the improvement if used 10% of the time, say we improved by double the speed, it will only have a fractional effect on the whole system
Amdahl’s Law

• Formally
• \( P \) = proportion which we are improving
• \( S \) = speedup

\[
\text{Improvement} = \frac{1}{(1 - P) + \frac{P}{S}}
\]

• So if you make 10% of your program 10 times faster…. 

• Rule 2: generally speaking 90% of the work is being done by 10% of the code

• What does that tell us?
Hashing Components

• At the end of today’s class you should be confident enough to know what these all mean….

• Hash function
• Hash table
• Collision
• Load

Hash function

• This is an algorithm for taking inputs and producing fixed value outputs

  JAVA → 34h3423j43
  Tango → 6w2

• Used on many different areas
  – Security
  – Error checking
  – Cryptography
  – Today’s class
Hashing

- `hashCode` method used in HashMap, HashSet is the standard java hashing algorithm
- Computes some (hopefully unique) int from each object
- Rule: if two hashes are different then the input differs in some way

One way hashes

- Given some input we can compute `Hash(input)`
- Almost impossible to find: `Hash(x) == Hash(y)`
- Given the results of the Hash can’t compute the input, but given the input, can verify the hash
Example: MD5

- Message digest algorithm
  - MD5("The quick brown fox jumps over the lazy dog") =
    - 9e107d9d372bb6826bd81d3542a419d6
  - MD5("The quick brown fox jumps over the lazy cog") =
    - 1055d3e698d289f2af8663725127bd4b
  - MD5("")
    - d41d8cd98f00b204e9800998ecf8427e

Algorithm

- Example: hash code of String

```java
int h = 0;
for (int i = 0; i < s.length(); i++)
    { h = 31 * h + s.charAt(i); }
```

- Hash code of "eat" is 100184
- Hash code of "tea" is 114704
Hashing

• Whatever returned be compatible with equals:
• if x.equals(y), then has to x.hashCode() == y.hashCode()
• Object.hashCode hashes memory address
• NOT compatible with redefined equals
• Remedy: Hash all fields and combine codes:

```java
public class Employee {
    public int hashCode() {
        return name.hashCode() + new Double(salary).hashCode();
    }
    ...
}
```
Shallow vs. Deep Copy

- Assignment (copy = e) makes shallow copy
- Clone should be defined to make deep copy
- Employee cloned = (Employee)e.clone();
Cloning

- `Object.clone` makes new object and copies all fields
- Cloning is subtle
- `Object.clone` is protected
- Subclass must redefine clone to be public

```java
public class Employee {
    public Object clone() {
        // not complete
        return super.clone();
    }
    ...
}
```

Cloneable Interface

- `Object.clone` is nervous about cloning
- Will only clone objects that implement Cloneable interface

```java
public interface Cloneable {
}
```

- Interface has no methods!
- Tagging interface—used in test if `x` implements Cloneable
- `Object.clone` throws `CloneNotSupportedException`
- A checked exception!
clone

```java
public class Employee
    implements Cloneable
{
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            return null; // won't happen
        }
    }
    ...
}
```

default

• By default the clone method is very lazy

• Shallow copy!!!
  – For immutable objects not a problem
• Why doesn't clone make a deep copy?
  – Wouldn't work for cyclic data structures
• Not a problem for immutable fields
• You must manually clone mutable fields
Deep cloning

```java
public class Employee
    implements Cloneable
{
    public Object clone()
    {
        try
        {
            Employee cloned = (Employee)super.clone();
            cloned.hireDate = (Date)hiredate.clone();
            return cloned;
        }
        catch(CloneNotSupportedException e)
        {
            return null; // won't happen
        }
    }
    ...
}
```
Cloning and Inheritance

- Object.clone is paranoid
  - clone is protected
  - clone only clones Cloneable objects
  - clone throws checked exception
- You don't have that luxury
- Manager.clone must be defined if Manager adds mutable fields
- Rule of thumb: if you extend a class that defines clone, redefine clone
- Lesson to learn: Tagging interfaces are inherited. Use them only to tag properties that inherit

Objects

- So you understand…
  - Object equals
  - Object class objects
  - Object clone
Working with the unknown

• Generally when you have Object from some class,
  – you wrote it yourself, so have doc/source
  – Using standard library, have docs
  – Unknown class, have no idea how to:
    • Instantiated
    • Construct
    • If you don’t know how to use, probably not a good idea to use 😊

Reflection

• Ability of running program to find out about its objects and classes
• Class object reveals
  – superclass
  – interfaces
  – package
  – names and types of fields
  – names, parameter types, return types of methods
  – parameter types of constructors
• Great of dynamic operation
Who cares?

- Most languages don’t have this
- Allows a program which is handling your stuff to display and access class properties
- Useful in visual environments

Reflection

- Class getSuperclass()
- Class[] getInterfaces()
- Package getPackage()
- Field[] getDeclaredFields()
- Constructor[] getDeclaredConstructors()
- Method[] getDeclaredMethods()
Enumerating Fields

• Print the names of all static fields of the Math class:
  Field[] fields =
      Math.class.getDeclaredFields();
  for (Field f : fields)
      if (Modifier.isStatic(f.getModifiers()))
          System.out.println(f.getName());

Enumerating Constructors

    for (Constructor c : cons)
    {
        Class[] params = cc.getParameterTypes();
        System.out.print("Rectangle(");
        boolean first = true;
        for (Class p : params)
        {
            if (first) first = false; else
                System.out.print‚ (", ");
                System.out.print(p.getName());
        }
    System.out.println(")");
    }
Output

Rectangle()
Rectangle(java.awt.Rectangle)
Rectangle(int, int, int, int)
Rectangle(int, int)
Rectangle(java.awt.Point,
        java.awt.Dimension)
Rectangle(java.awt.Point)
Rectangle(java.awt.Dimension)

Getting a single method descriptor

- Supply method name
- Supply array of parameter types
- Example: Get Rectangle.contains(int, int):
  Method m =
    Rectangle.class.getDeclaredMethod(
        "contains", int.class, int.class);
- Example: Get default Rectangle constructor:
  Constructor c =
    Rectangle.class.getDeclaredConstructor();
- getDeclaredMethod, getDeclaredConstructor are
  varargs methods
Invoking a Method

- Supply implicit parameter (null for static methods)
- Supply array of explicit parameter values
- Wrap primitive types
- Unwrap primitive return value
- Example: Call System.out.println("Hello, World") the hard way.

```java
Method m = PrintStream.class.getDeclaredMethod("println", String.class);
m.invoke(System.out, "Hello, World!");
```
- invoke is a varargs method

Inspecting Objects

- Can obtain object contents at runtime
- Useful for generic debugging tools
- Need to gain access to private fields

```java
Class c = obj.getClass();
Field f = c.getDeclaredField(name);
f.setAccessible(true);
```
- Throws exception if security manager disallows access
- Access field value:

```
Object value = f.get(obj);
f.set(obj, value);
```
- Use wrappers for primitive types
Inspecting Objects

- Example: Peek inside string tokenizer
  Ch7/code/reflect2/FieldTester.java
- Output

```java
int currentPosition=0
int newPosition=-1
int maxPosition=13
java.lang.String str=Hello, World!
java.lang.String delimiters=,
boolean retDelims=false
boolean delimsChanged=false
char maxDelimChar=,
---
int currentPosition=5
...```

Inspecting Array Elements

- Use static methods of Array class
- Object value = Array.get(a, i);
  Array.set(a, i, value);
- int n = Array.getLength(a);
- Construct new array:
  Object a = Array.newInstance(type, length);
Generics

• We’ve spoken about using generics with regards to objects

• How is the code organized?

Generic Types

• A generic type has one or more type variables
• Type variables are instantiated with class or interface types
• Cannot use primitive types, e.g. no ArrayList<int>
• When defining generic classes, use type variables in definition:
  public class ArrayList<E>
  {
    public E get(int i) { . . . }
    public E set(int i, E newValue) { . . . }
    . . .
    private E[] elementData;
  }
Quick question?

- If S a subtype of T,

- Why is ArrayList<S> not a subtype of ArrayList<T>

• Generic method = method with type parameter(s)
public class Utils
{
    public static <E> void fill(ArrayList<E> a, E value, int count)
    {
        for (int i = 0; i < count; i++)
            a.add(value);
    }
}

• A generic method in an ordinary (non-generic) class
• Type parameters are inferred in call
ArrayList<String> ids = new ArrayList<String>();
Utils.fill(ids, "default", 10); // calls
Utils.<String>fill
Generic types

- Advantages?

- Disadvantages?

Type Bounds

- Type variables can be constrained with type bounds
- Constraints can make a method more useful
- The following method is limited:

```java
public static <E> void append(ArrayList<E> a, ArrayList<E> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```

- Cannot append an ArrayList<Rectangle> to an ArrayList<Shape>
Type Bounds

• Overcome limitation with type bound:

```java
public static <E, F extends E> void append(
    ArrayList<E> a, ArrayList<F> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```

• extends means "subtype", i.e. extends or implements
• Can specify multiple bounds:
  E extends Cloneable & Serializable

Wildcards

• Definition of append never uses type F. Can simplify with wildcard:

```java
public static <E> void append(
    ArrayList<E> a, ArrayList<? extends E> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```