Objects (continued)
Announcements

- **Homework 0 is out**
  - Due Tuesday at 11:59pm

- **Moving between sections**
  - If you have a conflict, it’s best to change sections officially in the course registration system
  - If all non-conflicting sections are full, you may attend others, but priority goes to those who are registered
    - **Others please wait to take a seat until the section begins**

- **Waitlists**
  - There is room in section A. Room in other sections may open up.
  - If you cannot attend section A, email us. We will give you access to SVN so you can do the first assignment
  - No extra late days for waitlisted students! Show you care about the class by doing the work.
Key object concepts

- **Inside an object**
  - Kinds of members: Fields, Methods, Constructors
  - Visibility from the outside: hiding the members
  - The keyword `this`

- Interfaces and the management of expectations
  - Java interfaces
  - Introduction to types

- Objects and the heap
  - Method dispatch

- Objects and identity
  - Equals vs. `==`

- Class
  - Defining the object template
  - Diagrams can show relationships among classes
  - A class can have its own *static* fields and methods

- Objects and mutability
  - Abstract mutability and implementation mutability
class Point {
    int x, y;
    int getX() { return x; } // a method; getY() is similar
    Point(int px, int py) { x = px; y = py; } // constructor for creating the object
}

class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return origin; }
    int getWidth() { return width; }
    void draw() {
        drawLine(origin.getX(), origin.getY(), // first line
                 origin.getX()+width, origin.getY());
        ... // more lines here
    }
    Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}
Example: Points and Rectangles

class Point {
    int x, y;
    int getX() { return x; } // a method; getY() is similar
    Point(int px, int py) { x = px; y = py; } // constructor for creating the object
}
class Rectangle {
    Point origin;
    int width, height;
    Point getOrigin() { return origin; }
    int getWidth() { return width; }
    void draw() {
        // first line
        drawLine(origin.getX(), origin.getY(),
                 origin.getX()+width, origin.getY());
        // more lines here
    }
    Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}

Some Client Code

Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
What’s really going on?

Method Stack

main()
  o
  r
  rightEnd=5

o : Point
  x = 0
  y = 10

r : Rectangle
  origin
  width = 5
  height = 10

Some Client Code

Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5
Anatomy of a Method Call

r.setX(5)

The **receiver**, an implicit argument, called **this** inside the method

Method **arguments**, just like function arguments

The method **name**. Identifies which method to use, of all the methods the receiver’s class defines
The keyword **this** refers to the “receiver”

class Point {
    int x, y;
    int getX() { return x; }
    Point(int px, int py) { x = px; y = py; }
}

can also be written in this way:

class Point {
    int x, y;
    int getX() { return this.x; }
    Point(int x, int y) { this.x = x; this.y = y; }
}
Controlling access by client code

```java
class Point {
    private int x, y;
    public int getX() { return x; } // a method; getY() is similar
    public Point(int px, int py) { x = px; y = py; } // constructor for creating the object
}
class Rectangle {
    private Point origin;
    private int width, height;
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(origin.getX(), origin.getY(), // first line
                 origin.getX()+width, origin.getY());
        ... // more lines here
    }
    public Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}
```
Hiding interior state

```java
class Point {
    private int x, y;
    public int getX() { return x; }
    public Point(int px, int py) { x = px; y = py; }
}

class Rectangle {
    private Point origin;
    private int width, height;
    public Point getOrigin() { return origin; }
    public int getWidth() { return width; }
    public void draw() {
        drawLine(origin.getX(), origin.getY(),
                 origin.getX()+width, origin.getY());
    
    public Rectangle(Point o, int w, int h) {
        origin = o; width = w; height = h;
    }
}

Some Client Code

Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.getOrigin().getX() + r.getWidth(); // 5

Client Code that will not work in this version

Point o = new Point(0, 10); // allocates memory, calls ctor
Rectangle r = new Rectangle(o, 5, 10);
r.draw();
int rightEnd = r.origin.x + r.width; // trying to “look inside”
```
**Discussion:**

- **What are the benefits of private fields?**
- **Methods can also be private – why is this useful?**
Key object concepts

- Inside an object
  - Kinds of members: Fields, Methods, Constructors
  - Visibility from the outside: hiding the members
  - The keyword `this`

- **Interfaces and the management of expectations**
  - Java interfaces
  - Introduction to types

- Objects and the heap
  - Method Dispatch

- Objects and identity
  - Equals vs. `==`

- Class
  - Defining the object template
  - Diagrams can show relationships among classes
  - A class can have its own `static` fields and methods

- Objects and mutability
  - Abstract mutability and implementation mutability
Contracts and Clients

- **Contract of service provider and client**
  - Interface specification
  - Functionality and correctness expectations
  - Performance expectations
  - Hiding of respective implementation details
Java **interfaces** and **classes**

**Object-orientation**

1. Organize program functionality around kinds of abstract “objects”
   - For each object kind, offer a specific set of operations on the objects
   - Objects are otherwise opaque
     - Details of representation are hidden
   - “Messages to the receiving object”

2. Distinguish **interface** from **class**
   - **Interface**: expectations
   - **Class**: delivery on expectations (the implementation)

3. Explicitly represent the taxonomy of object types
   - This is the “inheritance hierarchy”
     - A **square** is a **shape**
Functional Lists of Integers

- Some operations we expect to see:
  - **create** a new list
    - empty, or by adding an integer to an existing list
  - return the **size** of the list
  - **get** the $i^{th}$ integer in the list
  - **concatenate** two lists into a new list

- Key questions
  - How to **implement** the lists?
    - Many options
      - Arrays, linked lists, etc
      - How to hide the details of this choice from client code?
      - Why do this?
  - How to state **expectations**?
    - A variable $v$ can reference a list of integers
Two ways to put a new empty list into a variable

`IntList l = new EmptyIntList();`
`EmptyIntList al = new EmptyIntList();`
Interfaces – stating **expectations**

- The IntList interface

```java
public interface IntList {
    int size();
    int get(int n);
    IntList concatenate(IntList otherList);
    String toString();
}
```

- The declaration for `v` ensures that any object referenced by `v` will have implementations of the methods `size`, `get`, `concatenate`, and `toString`

```java
Intlist v = ...

int len = v.size();
int third = v.get(2);
System.out.println (v.toString());
```
Implementing lists

• Two options (among many):
  • Arrays
    
    ![Array Example]

  • Linked lists
    
    ![Linked List Example]

• Operations:
  • **create** a new **empty** list
  • return the **size** of the list
  • return the *i*th integer in the list
  • **create** a list by adding to the **front**
  • concatenate two lists into a new list

<table>
<thead>
<tr>
<th>Operations</th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>create a new empty list</td>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>return the size of the list</td>
<td>const</td>
<td>linear</td>
</tr>
<tr>
<td>return the <em>i</em>th integer in the list</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>create a list by adding to the front</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>concatenate two lists into a new list</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Implementation of interfaces

- Classes can implement one or more interfaces.

```
public class IntListCell implements IntList, Cloneable {...}
```

**Semantics**
- **Must provide code** for all methods in the interface(s)

**Best practices**
- Define an interface whenever there may be multiple implementations of a concept
- Variables should have **interface type**, not class type
  ```
  int addList(ArrayList list) { ... // no!
  int addList(List list) { ... // yes!
  ```
The size of a list $L$ is

- $0$ if $L$ is the empty list
- $1 + \text{size of the tail of } L$ otherwise
public class EmptyIntList implements IntList {
    public int size() {
        return 0;
    }
    ...}

public class IntListCell implements IntList {
    public int size() {
        return 1 + next.size();
    }
    ...}
public class EmptyIntList implements IntList {
    public int size() {
        return 0;
    }
    ...
}

Base case

public class IntListCell implements IntList {
    private int value;
    private IntListCell next;

    public int size() {
        return 1 + next.size();
    }
    ...
}

Type is wrong! May be a cell or an empty list!

Inductive case
List Representation (FIXED!)

```java
public class EmptyIntList implements IntList {
    public int size() {
        return 0;
    }
    ...
}
```

**Base case**

```java
public class IntListCell implements IntList {
    private int value;
    private IntList next;

    public int size() {
        return 1 + next.size();
    }
    ...
}
```

**Inductive case**

Interface type provides needed flexibility.
List Constructors

public class EmptyIntList implements IntList {
    public EmptyIntList() {
        // nothing to initialize
    }
    . . .
}

public class IntListCell implements IntList {
    public IntListCell(int val, IntList next) {
        this.value = val;
        this.next = next;
    }

    private int value;
    private IntList next;
    . . .
}
Some Client Code

In main(...)
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
Some Client Code

```java
public IntListCell(int value, IntList next) {
  // value is 5, next is emptyList
  this.value = value; // this is fiveList
  this.next = next;
}
```

In main(...)

IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);

Some Client Code

In main(...)
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList); // what happens?
Implementing Concatenate

public class EmptyIntList implements IntList {
    public IntList concatenate(IntList other) {
        return other;
    }
    ...
}

Base case

public class IntListCell implements IntList {
    public IntList concatenate(IntList other) {
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext);
    }
    ...
}

Inductive case

Two concatenate methods – which do we use?
Some Client Code

In main(…)
IntList emptyList = new EmptyIntList();
IntList fiveList = new IntListCell(5, emptyList);
IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList); // what happens?
Key object concepts

- **Inside an object**
  - Kinds of members: Fields, Methods, Constructors
  - Visibility from the outside: hiding the members
  - The keyword *this*

- **Interfaces and the management of expectations**
  - Java interfaces
  - Introduction to types

- **Objects and the heap**
  - Method dispatch

- **Objects and identity**
  - Equals vs. ==

- **Class**
  - Defining the object template
  - Diagrams can show relationships among classes
  - A class can have its own *static* fields and methods

- **Objects and mutability**
  - Abstract mutability and implementation mutability
Method dispatch (simplified)

Example:

IntList fourList = new IntListCell(4, emptyList);
IntList fourFive = fourList.concatenate(fiveList);

• Step 1 (compile time): determine what type to look in
  ▪ Look at the static type (IntList) of the receiver (fourList)

• Step 2 (compile time): find the method in that type
  ▪ Find the method in the class with the right name
    ▪ Later: there may be more than one such method

  IntList concatenate(IntList otherList);

  ▪ Keep the method only if it is accessible
    ▪ e.g. remove private methods
  ▪ Error if there is no such method
Method dispatch (simplified)

Example:

List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList);

- Step 3 (run time): Determine the run-time type of the receiver
  - Look at the object in the heap and get its class

- Step 4 (run time): Locate the method implementation to invoke
  - Look in the class for an implementation of the method we found statically (step 2)

```java
public IntList concatenate(IntList other) {
    IntList newNext = next.concatenate(other);
    return new IntListCell(value, newNext);
}
```

- Invoke the method
Some Client Code

```java
class IntListCell {
    public IntList concatenate(IntList other) {
        // this is fourList, other is fiveList
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext);
    }
}

List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList); // what happens?
```
A Question for You!

1. What concatenate method is called next?
2. What does the final heap look like?

```java
class IntListCell {
    public IntList concatenate(IntList other) {
        // this is fourList, other is fiveList
        IntList newNext = next.concatenate(other);
        return new IntListCell(value, newNext);
    }
}
```

List fourList = new IntListCell(4, emptyList);

List fourFive = fourList.concatenate(fiveList); // what happens?
In main(...)
List emptyList = new EmptyIntList();
List fiveList = new IntListCell(5, emptyList);
List fourList = new IntListCell(4, emptyList);
List fourFive = fourList.concatenate(fiveList); // what happens?
Key object concepts

- **Inside an object**
  - Kinds of members: Fields, Methods, Constructors
  - Visibility from the outside: hiding the members
  - The keyword *this*

- **Interfaces and the management of expectations**
  - Java interfaces
  - Introduction to types

- **Objects and the heap**
  - Method dispatch

- **Objects and identity**
  - *Equals vs. ==*

- **Class**
  - Defining the object template
  - Diagrams can show relationships among classes
  - A class can have its own *static* fields and methods

- **Objects and mutability**
  - Abstract mutability and implementation mutability
Object identity vs. equality

- There are two notions of equality in OO
  - The *same object*. References are the same.
  - Possibly different objects, but equivalent content
    - From the client perspective!! The actual internals might be different

```java
String s1 = new String ("abc");
String s2 = new String ("abc");
```

- There are two string objects, `s1` and `s2`.
  - The strings are equivalent, but the references are different

```java
if (s1 == s2) { same object } else { different objects }
```

```java
if (s1.equals(s2)) { equivalent content } else { not }
```

- An interesting wrinkle: *literals*

```java
String s3 = "abc";
String s4 = "abc";
```

- These are true: `s3==s4. s3.equals(s2). s2 != s3`
Principles of Software Construction: Objects, Design, and Concurrency

Exceptions

Jonathan Aldrich  Charlie Garrod

© 2012 J Aldrich and W Scherlis
Help! What do I do now?

```java
int readFromFile(String file) {
    if (exists(file))
        return ... // read the integer
    else
        // uh, oh!
}

int getValueFromList(int index) {
    if (index < 0)
        // uh, oh!
    else ...
}

void transferFunds(int amount, Account to) {
    if (tooPoor())
        // uh, oh!
    else ...
}
```
Exceptions

- Exceptions notify the caller of an operation of failure

- Benefits of exceptions
  - Provide high-level summary of error and stack trace
    - compare: core dumped in C
Throwing Exceptions

• Programmatically throwing exceptions
  ▪ `throw new` ScaryException(“reason we are scared”);
  ▪ For example: Java library code throws exceptions when I/O fails

• Exceptions thrown by the Java runtime
  ▪ e.g. failed cast, null pointer dereference, array out of bounds, out of memory

• Semantics
  ▪ Exception propagates from callee to caller
  ▪ Until main() is reached, or the exception is caught
Exceptions

• Exceptions notify the caller of an operation of failure

• Benefits of exceptions
  ▪ Provide high-level summary of error and stack trace
    • compare: core dumped in C
  ▪ **Can’t forget to handle common failure modes**
    • compare: using a flag or special return value
  ▪ **Can optionally recover from failure**
    • compare: calling System.exit()
try {
    dangerousOperation();
} catch (MildException e) {
    recover();
} catch (DeadlyException e) {
    revive();
}
Exceptions

• Exceptions notify the caller of an operation of failure

• Benefits of exceptions
  ▪ Provide high-level summary of error and stack trace
    • compare: core dumped in C
  ▪ Can’t forget to handle common failure modes
    • compare: using a flag or special return value
  ▪ Can optionally recover from failure
    • compare: calling System.exit()
  ▪ Can clean up on the way out
Finally

```java
try {
    dangerousOperation();
} catch (MildException e) {
    recover();
} catch (DeadlyException e) {
    revive();
} finally { // called on normal completion, any catch block,
    cleanup(); // or uncaught exception
}
```
Exceptions

- Exceptions notify the caller of an operation of failure

Benefits of exceptions

- Provide high-level summary of error and stack trace
  - compare: core dumped in C
- Can’t forget to handle common failure modes
  - compare: using a flag or special return value
- Can optionally recover from failure
  - compare: calling System.exit()
- Can clean up on the way out

- **Improve code structure**
  - Separates failure code from common-case code
  - Allows handling multiple failure type, and multiple failure locations, with one failure handler
The Exception Hierarchy

Object

Exception

... ...

RuntimeException IOException ClassNotFoundException

NullPointerException IndexOutOfBoundsException

... ...

FileNotFoundException EOFException

IndexOutOfBoundsException

FileNotFoundException
Checked and Unchecked Exceptions

• Unchecked exception
  - subclass of RuntimeException
  - indicates an error which is *highly unlikely* and/or *typically unrecoverable*
  - Examples: failed cast, null pointer dereference, array out of bounds, out of memory

• Checked exception
  - Subclass of Exception but not RuntimeException
  - Indicates an error that every caller should be aware of and explicitly decide to handle or pass on
  - Must declare in your method signature if your method might throw one!
    - whether directly or because you call another method that does

```java
void dangerousOperation() throws DeadlyException { ... }
```
Guidelines for Exception Use

- **Catch and handle all checked exceptions**
  - Unless there is no good way to do so, in which case you should pass them on to your caller or throw a RuntimeException

- **Use runtime exceptions for programming errors**
  - If you receive bad input, throw a subclass of RuntimeException

- **Other good practices**
  - Do not swallow an exception without doing anything. At least print a stack trace! Better yet, try to recover.
  - When you throw an exception, provide an informative string to the constructor explaining the error
Toad’s Take-Home Messages

- **OOP** – code is organized code around 
  *kinds of things*
  - **Objects** correspond to things/concepts of interest
  - Objects embody:
    - State – held in **fields**, which hold or reference data
    - Actions – represented by **methods**, which describe operations on state
    - **Constructors** – how objects are created
  - A **class** is a family of similar objects
  - An **interface** states expectations for classes and their objects

- Objects reside in the **heap**
  - They are accessed by **reference**, which gives the objects **identity**
  - **Dispatch** is used to choose a method implementation based on the **class** of the **receiver**
  - Equivalence (**equals**) does not mean the same object (==)

- **Exceptions** are a structured way to manage failure