CSE 331

Review: Classes, Inheritance, and Collections

slides created by Marty Stepp
http://www.cs.washington.edu/331/
Recall: A typical Java class

public class Point {
    private int x; // fields
    private int y;

    public Point(int x, int y) { // constructor
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; } // accessor
    public int getY() { return y; }

    public void translate(int dx, int dy) {
        x += dx;
        y += dy; // mutator
    }

    public String toString() { // for printing
        return "(" + x + ", " + y + ")";
    }
}

Effective Java Tip #10

Throughout this course, we will refer to design heuristics from Joshua Bloch's excellent *Effective Java (2nd edition)* book.

**Tip #10:** Always override `toString`.

**Why?**

- If you can print your objects, you can easily see their state.
- Clients can print your objects, which is a very common thing to do.
- Clients can put them into collections and print the collection.
- Nobody likes to see the default "ClassName@a97e2f" output.
- Helps with debugging your own code as you're writing it.
Multiple constructors

```java
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor
Class question

• We are given a class `BankAccount` where each object represents a user's bank data such as name and balance.

• We must add functionality to the class so that each account object is automatically given a new unique ID number as it is created.
  - First account = ID 1; second account = ID 2; etc.

• How do we do it?
Static fields

private static type name;

or,

private static type name = value;

- Example:
  private static int theAnswer = 42;

**static**: Shared by all instances (objects) of a class.
  - A shared global field that all objects of the class can access/modify.
  - Like a class constant, except that its value can be changed.
What would happen if `objectCount` were non-static? If `id` were static?
**Recall: Inheritance**

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/reuse code between two or more classes
  - introduces **polymorphism** (can treat the classes the same way)
- **superclass**: Parent class being extended.
- **subclass**: Child class that inherits behavior from superclass.
- **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.
A typical subclass

```java
public class CheckingAccount extends BankAccount {
    private double fee;    // adding new state

    public CheckingAccount(String name, double fee) {
        super(name);        // call superclass c'tor
        this.fee = fee;
    }

    // adding new behavior
    public double getFee() {
        return fee;
    }

    // overriding existing behavior
    public void withdraw(double amount) {
        super.withdraw(amount + fee);
    }
}
```

• *Question:* Why not just add optional fee behavior to `BankAccount`?
Effective Java Tip #20

• **Tip #20:** Prefer class hierarchies to "tagged" classes.

• What's a "tagged" class, and why is it bad?
  
  ▪ If we add the fee code to BankAccount, each object will need some kind of field to "tag" or flag whether it uses fees or not.
  
  ▪ Adding that code complicates the existing class.
    • The new behavior will add ifs and logic to otherwise simple code.
  
  ▪ BankAccount already works; why risk breaking it?
  
  ▪ inheritance = **additive** rather than **invasive** change
    • The fee / no-fee logic will be decided entirely by the object type used.
Polymorphism

- **polymorphism**: Quality where the same code can be used with different kinds of objects and will behave in different ways.

- We can store a subclass object in a superclass variable.
  
  ```java
  BankAccount acct = new CheckingAccount("Bob", 1.50);
  ```

- We can pass a subclass object as a superclass parameter.
  
  ```java
  doStuff(acct);
  ...
  public static void doStuff(BankAccount ba) {
  ```

- The object we pass will always behave the same way ("its" way).
  - If `doStuff` calls `withdraw` on `acct`, the version from `CheckingAccount` is used.
Recall: Interfaces

- **interface**: A list of methods that a class can promise to implement.
  - Gives an is-a relationship and polymorphism *without* code sharing.

- Consider shape classes *Circle*, *Rectangle*, and *Triangle*.

- Some things are common to all shapes but computed differently:
  - perimeter: distance around the outside of the shape
  - area: amount of 2D space occupied by the shape
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}

Example:
public interface Shape {
    public double area();
    public double perimeter();
}

```
«interface»

Shape

area()
perimeter()

Circle
radius
Circle(radius)
area()
perimeter()

Rectangle
width, height
Rectangle(w, h)
area()
perimeter()

Triangle
a, b, c
Triangle(a, b, c)
area()
perimeter()
```
Implementing an interface

```java
public class name implements interface {
    ...
}
```

- Example:
  ```java
  public class Rectangle implements Shape {
      ...
      public double area() { ... }
      public double perimeter() { ... }
  }
  ```

- A class can declare that it "implements" an interface.
  - The class promises to implement each method in that interface.
    (Otherwise it will fail to compile.)
Collections as fields

• Many objects must store a collection of structured data.
  ▪ Many data structures to choose from:
    • array, list, set, map, stack, queue, ...
  ▪ Most kinds of collections have multiple implementations:
    • List: ArrayList, LinkedList
    • Set: HashSet, TreeSet, LinkedHashSet
    • Map: HashMap, TreeMap, LinkedHashMap

• Which structure is best to use depends on the situation:
  • Does the data need to be in a particular order?
  • Are duplicates allowed?
  • Do we need to store pairs or look things up by partial values ("keys")?
  • How will we access the data (randomly, in order, etc.)?
  • ...
# Collections summary

<table>
<thead>
<tr>
<th>collection</th>
<th>ordering</th>
<th>benefits</th>
<th>weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>by index</td>
<td>fast; simple</td>
<td>little functionality; cannot resize</td>
</tr>
<tr>
<td>ArrayList</td>
<td>by insertion, by index</td>
<td>random access; fast</td>
<td>slow to modify in middle/front</td>
</tr>
<tr>
<td>LinkedList</td>
<td>by insertion, by index</td>
<td>fast to modify at both</td>
<td>poor random access</td>
</tr>
<tr>
<td>TreeSet</td>
<td>sorted order</td>
<td>sorted; $O(\log N)$</td>
<td>elements must be comparable</td>
</tr>
<tr>
<td>HashSet</td>
<td>unpredictable</td>
<td>very fast; $O(1)$</td>
<td>unordered</td>
</tr>
<tr>
<td>LinkedHashSet</td>
<td>order of insertion</td>
<td>very fast; $O(1)$</td>
<td>uses extra memory</td>
</tr>
<tr>
<td>TreeMap</td>
<td>sorted order</td>
<td>sorted; $O(\log N)$</td>
<td>elements must be comparable</td>
</tr>
<tr>
<td>HashMap</td>
<td>unpredictable</td>
<td>very fast; $O(1)$</td>
<td>unordered</td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>order of insertion</td>
<td>very fast; $O(1)$</td>
<td>uses extra memory</td>
</tr>
</tbody>
</table>
Effective Java Tip #25

• Tip #25: Prefer lists to arrays.

• In the majority of cases where you want to store structured data, a list works much better than an array. Why?
  ▪ Lists automatically resize.
  ▪ Lists contain more useful operations such as insertion, removal, toString, and searching (indexOf / contains).
  ▪ Lists are more type-safe than arrays in certain cases.
    • Works: BankAccount[] a = new CheckingAccount[10]; // bad
    • Fails: List<BankAccount> l = new ArrayList<CheckingAccount>();
Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - The external view of a given type of objects.
  - Describes *what* an object does, not *how* it does it.
  - When you write classes, you are creating new ADTs.

- Clients of the object don't know exactly how its behavior is implemented, and they don't need to.
  - They just need to understand the idea of what the object represents and what operations it can perform.
Effective Java Tip #52

• **Tip #52:** Item 52: Refer to objects by their interfaces.
  - **Bad:** `ArrayList<String> list = new ArrayList<String>();`
  - **Good:** `List<String> list = new ArrayList<String>();`

• Why?
  - allows you to switch list implementations later if needed
  - keeps you from relying on behavior exclusive to ArrayList
  - also use the above style for declaring parameter / return types!
    ```java
    public static List<String> read(String file) {...
    ```
From spec to code

• As developers, we are often given a spec and asked to implement it.

• The spec may tell us what classes and public methods to write. (Later in this course, it won't...!)
  ▪ Either way, it does not describe in detail how to implement them.

• We must figure out what internal state (fields) and helping behavior (methods) are necessary to implement the spec.
Spec-to-code question

• Let's implement a class BuddyList whose objects store all information about a user's instant messenger buddy list.

• Required functionality:
  ▪ create a new empty buddy list for a given user name
  ▪ add new buddies to the list (an object of type Buddy)
  ▪ examine the buddies in the list, in unspecified order
  ▪ search for a buddy in the list by name
  ▪ broadcast a message to all of the buddies in the list
    • Note: All methods should be as efficient as possible.

• How should the class be implemented?
  ▪ What are its methods and fields? What data structures to use?
Effective Java Tip #16

- **Tip #16**: Favor composition over inheritance.

- A BuddyList is similar to one of the existing Java collections, but with a bit of added functionality. So why not extend `HashMap`, etc.?
  - When you extend a class, your subclass inherits *all* of its behavior.
  - We don't want our buddy list to have all of those various methods.
    - BuddyList would now have methods like `clear`, `retainAll`, `keySet`, ...
    - This might expose the internal buddies data in ways we don't want.
  - This isn't a true "is-a" relationship. A buddy list isn't a map; it *uses* a map to help implement its functionality. It "has-a" map.
    - **composition**: Using another object as part of your state.