CSE 403
Lecture 11

Effective Java Design

Reading:

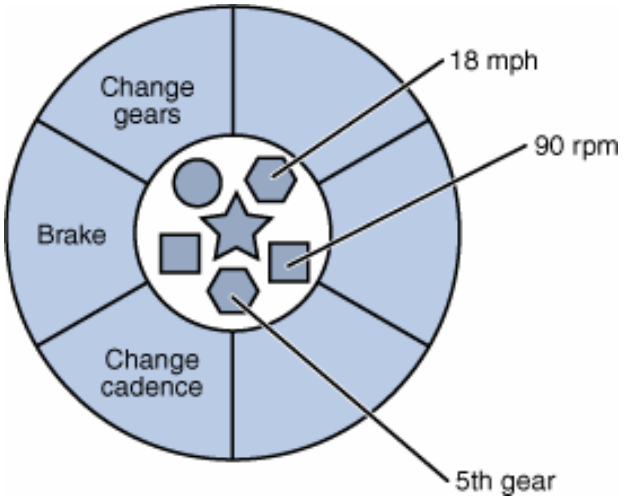
Effective Java, 2nd edition, Joshua Bloch

slides created by Marty Stepp
http://www.cs.washington.edu/403/

Levels of access

- **private**: visible only to one class
- **package-private** (default): visible to current package
- **protected**: visible to current packages, and any subclasses
- **public**: visible to any other class

  – Which level of access is best for fields? For methods? Constructors? For classes?

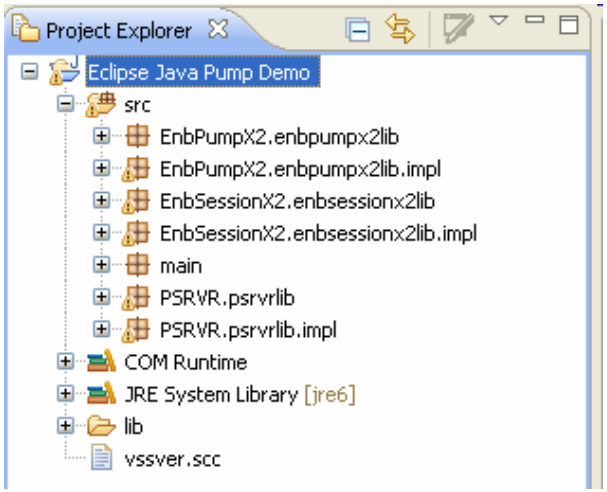


Effective Java Item 13

- **Item 13**: Minimize the accessibility of classes and members.
- **Item 14**: In public classes, use accessor methods, not public fields.
  - Should fields *always* be private?
  - Should methods *always* be public?
  - Why might it be useful to have a non-public class?
  - Poker question: The poker Player class wants to share its internal data "setter" methods with some classes, such as the PokerGame. But it doesn't want some other classes, such as rival players, to be able to set arbitrary data. How can this be achieved?

Packages

- **package**: a group of related Java classes
  - more useful in larger projects
  - helps group related classes and functionality into subsystems
  - creates a new separate *namespace*
  - provides a layer of access control
    - (can set a class or member as private to a package)
  - In Java, packages mirror folders
    - package `foo.bar` should be stored in folder `foo/bar/`



Classpath

- **class path**: The location(s) in which Java looks for class files.

- Can include:
  - the current "working directory" from which you ran javac / java
  - other folders
  - JAR archives
  - URLs
  - ...

- Can set class path manually running Java at command line:
  - `java -cp /home/stepp/libs:/foo/bar/jbl MyClass`

A package declaration

```
package name;

public class name { ...
```

Example:

```
package pacman.model;

public class Ghost extends Sprite {
    ...
}
```

- File `Sprite.java` should go in folder `pacman/model` .

Importing a package

```
import packageName.*;          // all classes
```

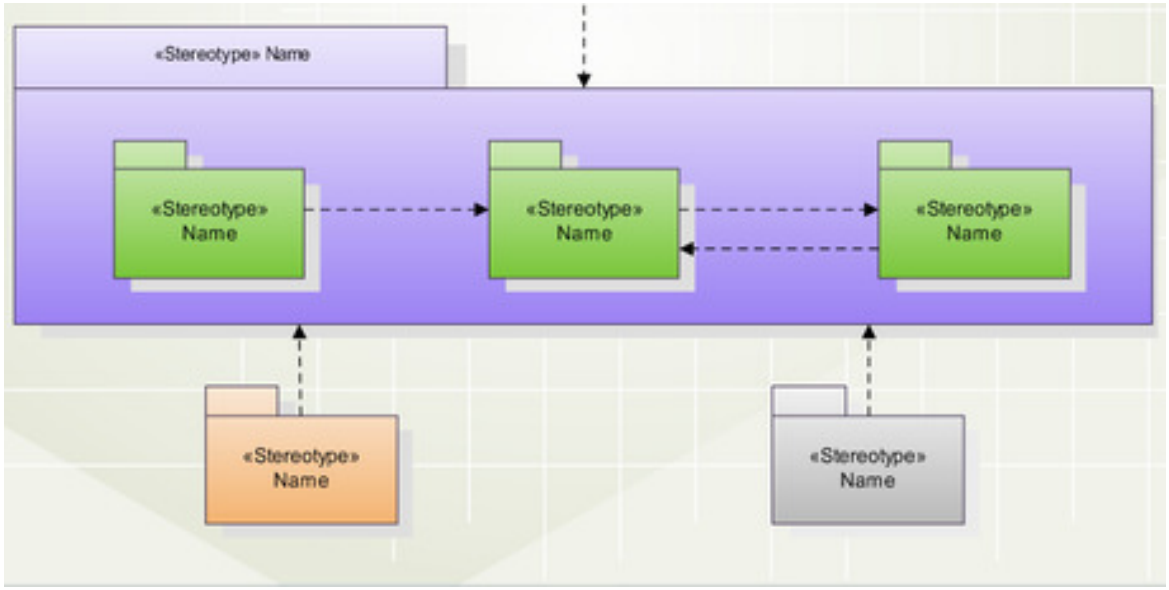
Example:

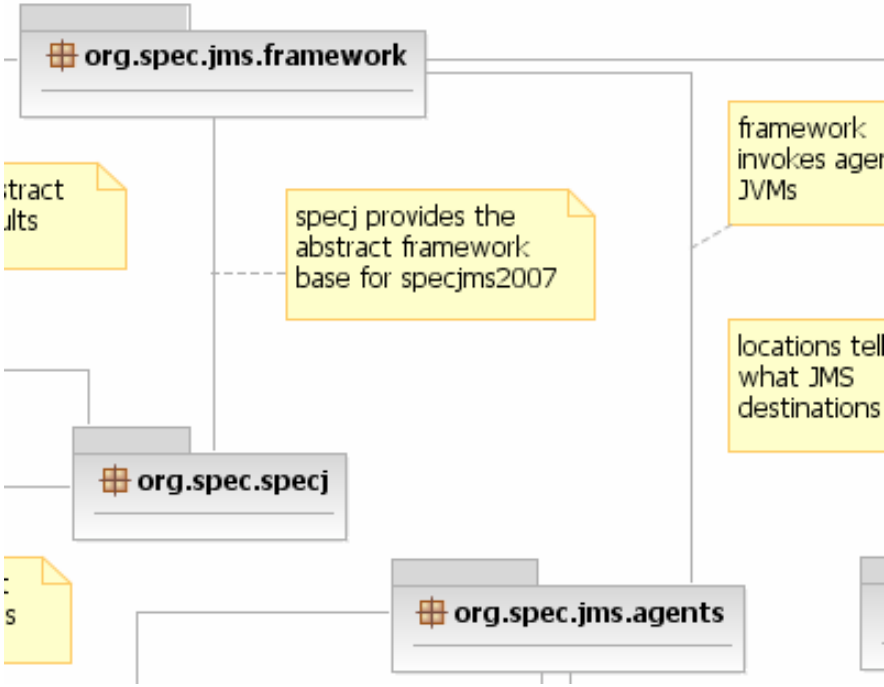
```
package pacman.gui;
import pacman.model.*;

public class PacManGui {
    ...
    Ghost blinky = new Ghost();
}
```

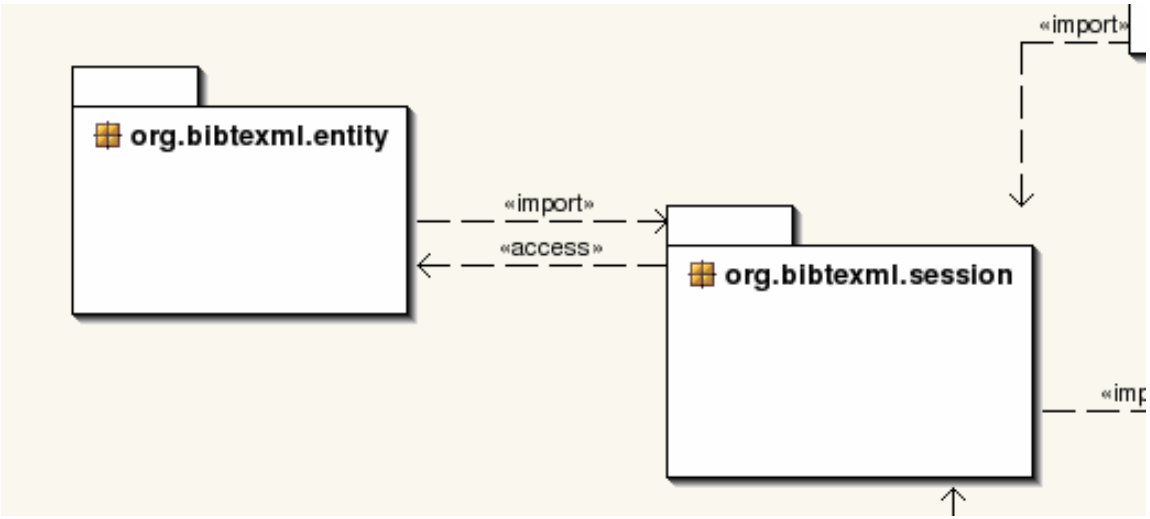
- `PacManGui` must import the model package to use it.

Packages in UML

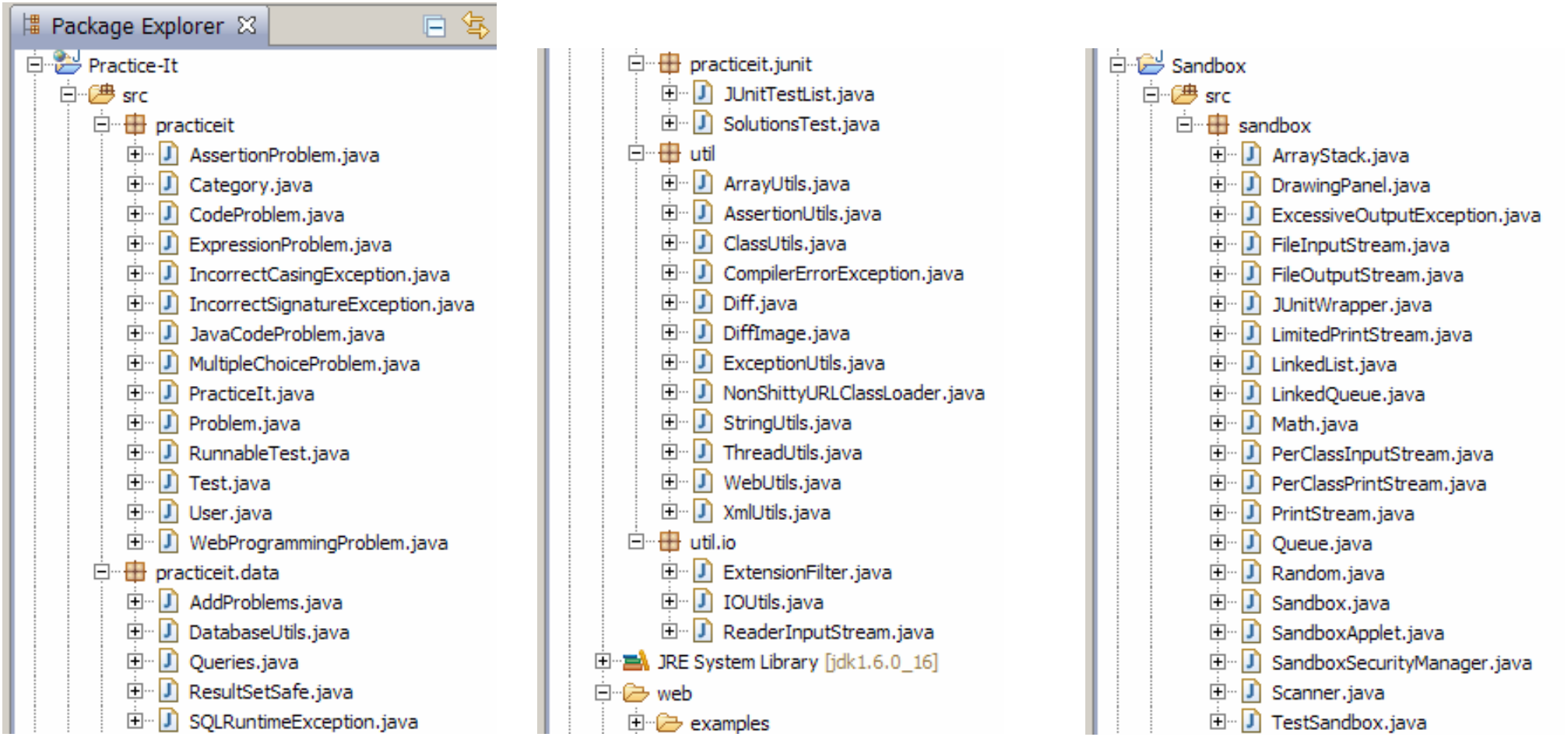
- UML class diagrams can depict packages as "folders":







Packages in Practice-It



- total lines of code:
  - practiceit (4302) / data (277), junit (692); util (2905) / io (289); sandbox (6688) ==> total = **15153**

Poker design question

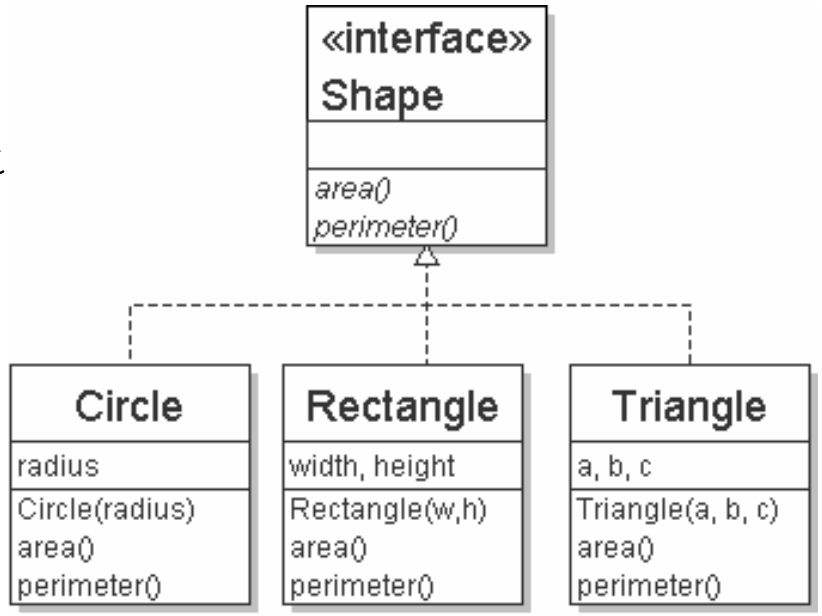
- The game has different kinds of players, including human, computer and network.

  How should these types of players be represented in the code?

  - As a `play()` method in the PokerGame class with lots of if/elses.
  - A Player class with a `type` field such as HUMAN, COMPUTER, etc.
  - A Player class with subclasses like HumanPlayer, etc.
  - A Player interface implemented by classes HumanPlayer, etc.

Interfaces

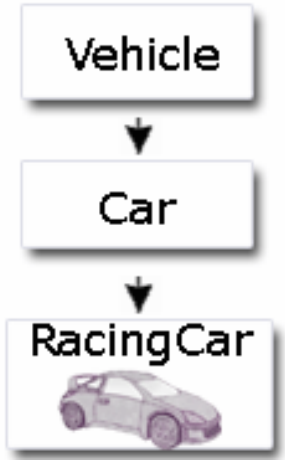
- interfaces should be used when you want:
  - **polymorphism**: treat different types of objects the same way
  - examples:
    - `List` --> `ArrayList`, `LinkedList`
    - `Map` --> `HashMap`, `TreeMap`
    - `Iterator`
    - `Comparable`
    - `ActionListener`



  - Java interfaces are sometimes also used for "tagging" abilities
    - `Cloneable`; `Serializable`; `Iterable`

Inheritance

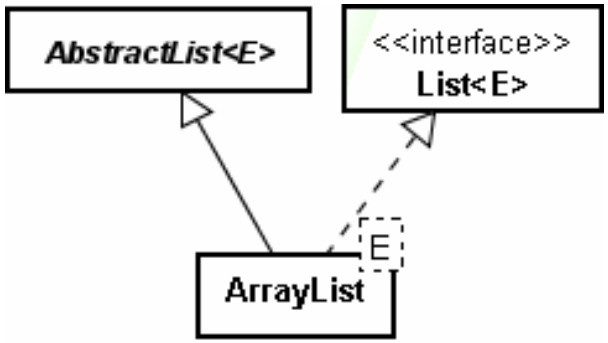
- inheritance should be used when you want:
  - **polymorphism**: treat different types of objects the same way
  - **code sharing**: subclass receives code from superclass
  - **substitutability**: client code can use a subclass object in any situation where a superclass object was expected, and expect the same results
    - ("Liskov Substitutability Principle")



  - example: Java Swing GUI framework `JComponent` class
    - properties: size, location, color, font, text label, border, events ...
    - extended by `JButton`, `JTextField`, `JCheckBox`, `JSlider`, etc.
    - any `JComponent` may be substituted for another in many cases

Effective Java Items 16-20

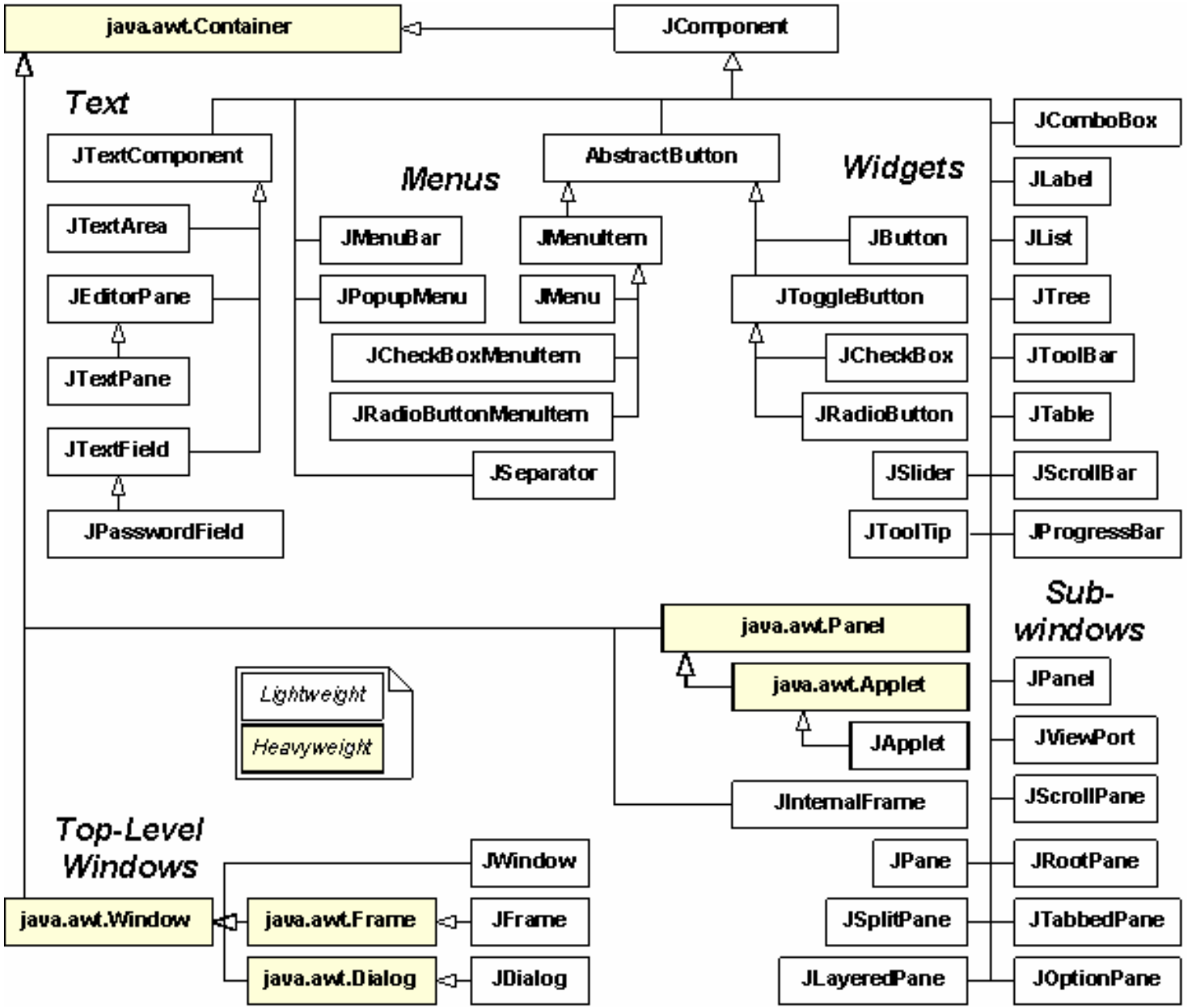
- **Item 16**: Favor composition over inheritance.
  - *has-a* can be better than *is-a*
  - "Lots of people HAVE a lawyer, but ..."



- **Item 18**: Prefer interfaces to abstract classes.
  - compromise: use both (`java.util.List` and `AbstractList`)

- **Item 20**: Prefer class hierarchies to "tagged" classes.
  - if you have a `type` or `kind` field, it's probably bad

Java Swing GUI hierarchy



Case Study: SortedIntList

- In CSE 143, I assign a program `SortedIntList`.
  - `SortedIntList` extends a provided `ArrayIntList`:
    - `public void add(int value)`
    - `public void add(int index, int value)`
    - `public void clear()`
    - `public boolean contains(int value)`
    - `public int get(int index)`
    - `public int indexOf(int value)`
    - `public boolean isEmpty()`
    - `public void remove(int index)`
    - `public void set(int index, int value)`
    - `public int size()`
    - `public String toString()`
  - Is this a good design? Why or why not?

Poker design question

- When a player makes a bet, a Bet object is produced.
  - Other Players need to know about the bet so they can decide how to respond to it (raise, fold, etc.).
  - But if we pass the Bet object to them, they could be evil and change it by calling methods like setAmount, cancel, etc.
  - How do we make it possible to inform the players about the Bet without its object being damaged?

Mutation

- **mutation**: A modification to the state of an object.

```
Point p = new Point(3, 5);
System.out.println(p);      // (3, 5)
p.translate(1, 3);          // mutator
System.out.println(p);      // (4, 8)
```

- Mutation must be done with care.
  - Can the object's state be damaged?
  - Is the old state important? Is it okay to "lose" it?
  - Do any other clients depend on this object?
    - Do they expect that its state will not change?

Effective Java item 15

- **Item 15**: Minimize mutability.
  - **immutable**: Unable to be changed.
    - achieved in Java by omitting `set` methods and/or using `final` fields

```
public class Bet {
    private final int amount;

    public Bet(int amount) {
        this.amount = amount;
    }
    public int getAmount() { return amount; }
    ...
}
```

  - If the Bet's `amount` field cannot be changed, what if the player wants to use a different bet amount later?

Defensive copying

- **defensive copy**: To duplicate an object prior to a mutation.

```
Point p = new Point(3, 5);
System.out.println(p);      // (3, 5)
Point copy = p.clone();
copy.translate(1, 3);       // mutator
System.out.println(copy);   // (4, 8)
System.out.println(p);      // (3, 5)
```

  - Sometimes you then perform the mutation to the copy.
  - Sometimes you perform the mutation to the original object.
    - The copy serves as a "backup" in case the mutation is undesirable.

  - **EJ Tip #39**: Make defensive copies when needed.

A poor design

- Suppose we have a `BankAccount` class:
  - `getBalance` method returns the bank account's balance.
  - The method also charges a $1.50 fee if you ask for the balance too many times (calling the method more than 3x per day).

- Why is this a poor design?
  - Don't combine a crucial accessor with an unrelated mutation.
    - Impossible to access without (unintentionally?) mutating.
  - Client might call it without thinking, to print balance etc.
    - Another client might have already called the method before me.

  - **side effects**: Additional externally visible behavior beyond the core functionality of a method.

Minimizing mutability

- **Effective Java Tip #15**: Minimize mutability.

- Why?
  - easier to design, implement, and use immutable objects
  - less prone to developer error
  - less prone to misuse by clients
  - more secure
  - can be optimized for better performance / memory use (sometimes)

  - from Effective Java: *"Classes should be immutable unless there is a very good reason to make them mutable."*
    - *"If a class cannot be immutable, limit its mutability as much as possible."*

Making a class immutable

- 1. Don't provide any methods that modify the object's state.
- 2. Ensure that the class cannot be extended.
- 3. Make all fields `final`.
- 4. Make all fields `private`. (ensure encapsulation)
- 5. Ensure exclusive access to any mutable object fields.
  - Don't let a client get a reference to a field that is a mutable object.

    (Don't allow any mutable representation exposure.)

The final keyword

- **final**: Unchangeable; unable to be redefined or overridden.
- Can be used with:
  - local variables (value can be set once, and can never be changed)
  - fields
  - static fields (they become "class constants")
  - classes (the class becomes unable to be subclassed)
  - methods (the method becomes unable to be overridden)
- **Effective Java Tip #17:** Design and document for inheritance or else prohibit it (by making your class `final`).

Examples of final

- on a local variable: `final int answer = 42;`
- on a field: `private final String name;`
  - set in constructor
- on a static constant: `public static final int DAYS = 7;`
- on a class: `public final class Point {`
  - no class can extend `Point`
- on a method: `public final int getX()`
  - no class can override `getX` (not necessary if class is already `final`)

Mutable Fraction class

```
public class Fraction implements Cloneable, Comparable<Fraction> {
    private int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()
    public void setNumerator(int n),  setDenominator(int d)

    public Fraction clone()
    public int compareTo(Fraction other)
    public boolean equals(Object o)
    public String toString()

    public void add(Fraction other)
    public void subtract(Fraction other)
    public void multiply(Fraction other)
    public void divide(Fraction other)
}
```

- How would we make this class immutable?

Immutable Fraction class

```
public final class Fraction implements Comparable<Fraction> {
    private final int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()
    // no more setN/D methods

    // no clone method needed
    public int compareTo(Fraction other)
    public boolean equals(Object o)
    public String toString()

    public Fraction add(Fraction other)         // past mutators
    public Fraction subtract(Fraction other)    // are producers
    public Fraction multiply(Fraction other)    // (return a new
    public Fraction divide(Fraction other)      // (object)
}
```

Other design principles

- **The Open/Closed Principle**: Software entities (classes, modules, etc) should be open for extension, but closed for modification.
- **The Liskov Substitution Principle**: Derived classes must be usable through the base class interface without the need for the user to know the difference.
- **The Dependency Inversion Principle**: Details should depend upon abstractions. Abstractions should not depend upon details.
- **The Interface Segregation Principle**: Many client-specific interfaces are better than one general purpose interface.
- **The Reuse/Release Equivalency Principle**: The granule of reuse is the same as that release. Only components released through a tracking system can be effectively reused.
- **The Common Closure Principle**: Classes that change together, belong together.
- **The Common Reuse Principle**: Classes that aren't reused together shouldn't be grouped.
- **The Acyclic Dependencies Principle**: The dependency structure for released components must be a directed acyclic graph. There can be no cycles.
- **The Stable Dependencies Principle**: Dependencies between released categories must run in the direction of stability. The dependee must be more stable than the depender.
- **The Stable Abstractions Principle**: The more stable a class category is, the more it must consist of abstract classes. A completely stable category is nothing but abstract classes.