OOP Fundamentals

ADTs, Interfaces, Inheritance & Polymorphism

CS 5010 Fall 2022

Instructors: Brian Cross and Tamara Bonaci

Credits: materials adapted from Maria Zontak's CS 5010 and Tony Mullen

Administrivia

Lab 1&2 due Friday @ 11:59pm

- Create a branch named **lab1**
- Copy lab code into directory named Lab1
- Push to server
- Create Pull Request
  - *Assign to TAs*

Administrivia

Homework #1:

Pre-design due on today by 11:59pm

Complete homework due on Monday, 9/26 by 11:59pm

Have you started?

Questions?

Abstract Data Types (ADTs)

- Defining a data type in terms of its *behavior* from the point of view of a user (client), as opposed to focusing on the details of the implementation
- Not specific to any specific programming language!
  - Don't confuse the general concept of ADTs with Java Abstract classes.

Abstract Data Types: Why?

Easier to understand: user only needs to understand specific relevant operations, not the underlying implementation

Easier to change: implementation can be modified without altering how client code interacts with the ADT

Less prone to bugs: ADT state can be insulated from bugs in client code

Abstract Data Types: Design by Contract

Can provide the basis of behaviors for ***design by contract***, extended by specifying

- **Preconditions**: conditions that must be true in order for some behavior to be initiated
- **Postconditions**: conditions that must be true when the behavior has completed
- **Invariants**: properties of a program which are always true, for any runtime state of the program

OOD Principles

Abstraction: Omitting details & focusing on general ideas and commonalities

Encapsulation: Building walls around components such that the component is responsible for its own behavior, and bugs in other parts of the system don't damage its integrity

Inheritance: A way to re-use code in a way that takes advantage of common functionality across classes and subclasses

Polymorphism: The ability for an object to exhibit different behaviors based on context

Related Design Concepts

- **Modularity**: dividing a system into components which can be designed, reasoned about, implemented, tested, and re-used independently of others
- **Information hiding**: concealing details of a component's implementation so that those details can be changed later without changing the rest of the system
- **Separation of concerns**: ensuring that related areas of functionality are grouped within appropriate components, rather than spread out across multiple components

Classifying Types

- **Mutable**: objects can be changed and provide operations to execute this change
- **Immutable**: objects' state cannot be changed. Java `String` objects are an example of an immutable type.
  - Immutability is an example of an invariant
  - Once created, the immutable object represents the same value through its lifetime

Classifying Operations

- **Creators**: create new objects of a type. May take an object as an argument, but not an object of the type being constructed.
- **Producers**: create new objects of the type from old objects of the same type (e.g., `concat` of String).
- **Observers**: take objects of the abstract type and return objects of a different type (e.g., `size` of List returns an int).
- **Mutators**: change objects (e.g., `add` method of List adds an element to the end of the list.)

ADT Examples in Java: int

Primitive integer type. `int` is immutable, so it has no mutators.

- **Creators**: numeric literals 0, 1, 2, ...
- **Producers**: arithmetic operators +. -, *, /
- **Observers**: comparison operators ==, !=, <, >
- **Mutators**: none

ADT Examples in Java: List

Java's list type. List is also an interface: other classes provide concrete implementation (e.g., ArrayList and LinkedList). List defines mutators, but they are optional for implementations.

- **Creators**: e.g., ArrayList and LinkedList constructors (both mutable), Collections.singletonList (immutable)
- **Producers**: Collections.unmodifiableList
- **Observers**: size, get
- **Mutators**: add, remove, addAll, Collections.sort

ADT Examples in Java: String

Java's string type. Immutable.

- **Creators**: String constructors
- **Producers**: concat, substring, toUpperCase
- **Observers**: length, charAt
- **Mutators**: none

Interfaces

What are they?

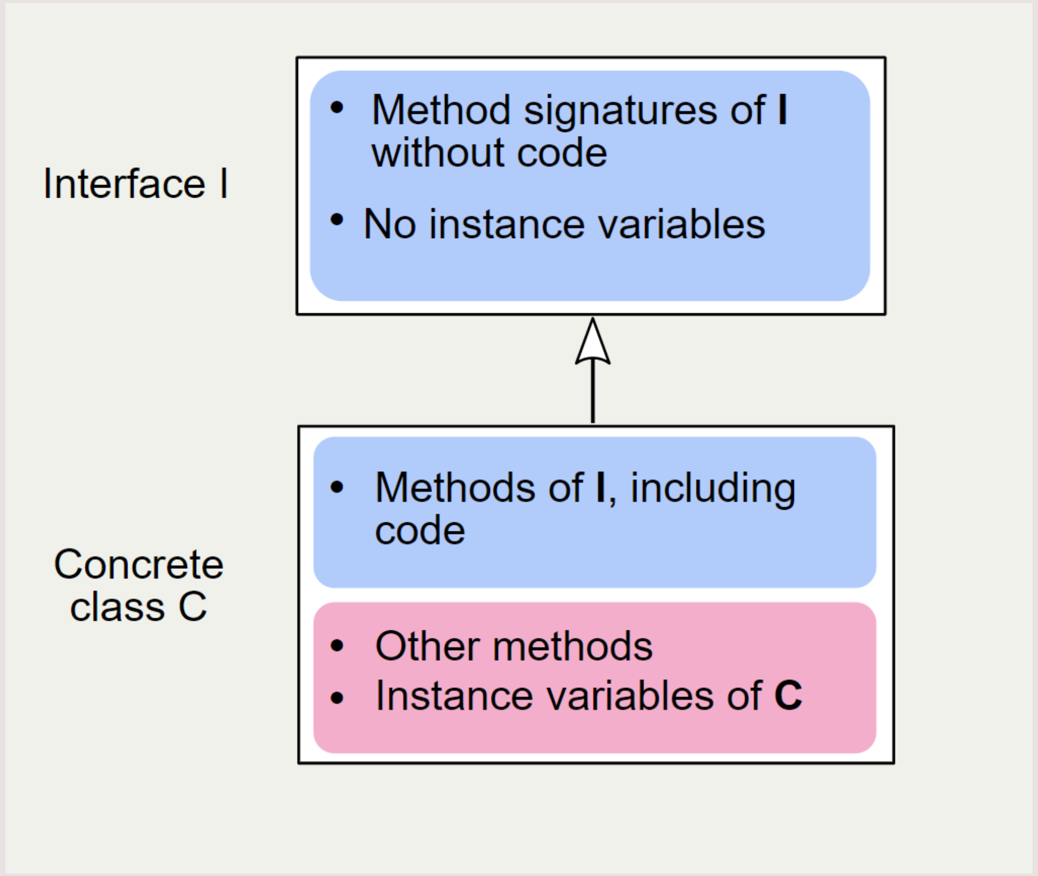
- Useful language mechanism for expressing an abstract data type
- Set of method declarations (signatures) for common behaviors
- A "contract" or protocol for what classes can do
- A class that agrees to (implements) an interface MUST implement its behaviors

Interface

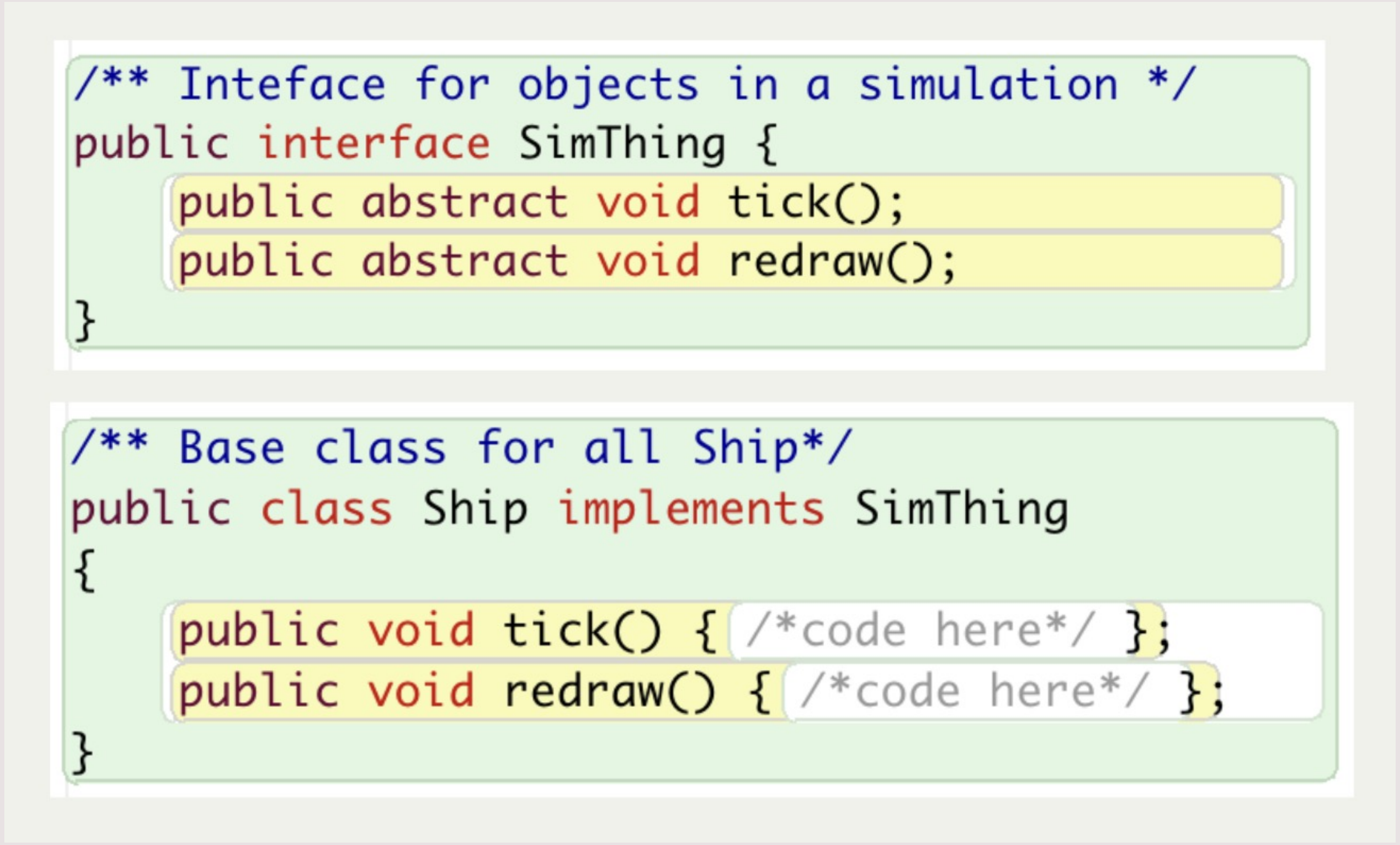
Why do we need them?

- Enable interaction without knowing specific implementation
- Take advantage of multiple inheritance *of type* for one class
- Enable polymorphism: classes that implement the same interface can be treated similarly

Interface: UML diagram


Interface I
• Method signatures of **I** without code
• No instance variables
Concrete class C
• Methods of **I**, including code
• Other methods
• Instance variables of **C**

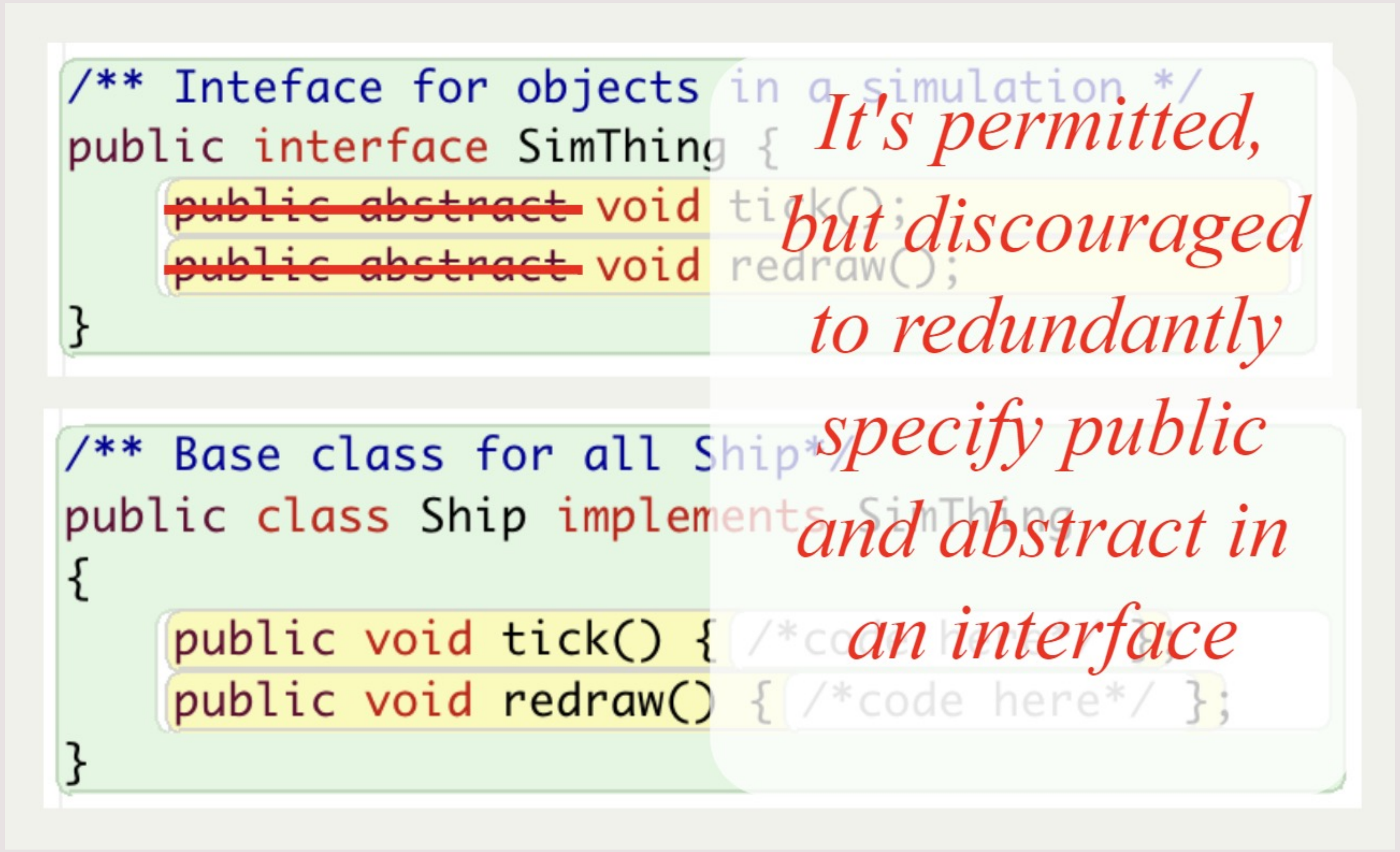
SimThing Interface



```
/** Inteface for objects in a simulation */
public interface SimThing {
    public abstract void tick();
    public abstract void redraw();
}
```

```
/** Base class for all Ship*/
public class Ship implements SimThing
{
    public void tick() { /*code here*/ };
    public void redraw() { /*code here*/ };
}
```

SimThing Interface



```
/** Inteface for objects in a simulation */
public interface SimThing {
    ~~public abstract~~ void tick();
    ~~public abstract~~ void redraw();
}
```

```
/** Base class for all Ship*/
public class Ship implements SimThing
{
    public void tick() { /*code here*/ };
    public void redraw() { /*code here*/ };
}
```

It's permitted, but discouraged to redundantly specify public and abstract in an interface

`implements`

Java `interface` declares a set of method signatures

- Says what behavior exists
- Does not say how behavior is implemented
- Does not describe state (but may include "final" constants)

Concrete class implements an interface

- contains `implements InterfaceName` in class declaration
- Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface

`Shape` Interface

GOAL: design a program that manipulates shapes on a canvas

- Possible shapes:
  - A circle **has** a pin (its center) and a radius
  - A square **has** a pin (top left corner) and a side length
  - A rectangle **has** a pin (top left corner), a width, and a height

Is this sufficient for writing an interface?

No, an interface defines *behavior*

Shape Interface

```
/** Interface for all shapes on canvas */
public interface Shape {
    void moveX();
    void moveY();
    double area();
    double circumference();
}
```

Are there other ways to translate the same behavior into an interface?

Shape Interface

We could consider shapes immutable and represent moves as producers...

```java
/** Interface for all shapes on canvas */
public interface Shape {
    Shape moveX();
    Shape moveY();
    double area();
    double circumference();
}
```

What is the Type of an Object?

Types in Java

1. **Primitive**: int/long, char, boolean, floating point (double, float)
2. **Non-primitive**: class types, interface implementations, arrays ...

Any instance of a class named `Example` conforms to these types:

1. The named class (`Example`)
2. Every interface that `Example` implements
3. More to come...

An instance can be used anywhere one of its types is appropriate.

Abstracting Common Behavior and Fields

- Possible shapes:
  - A circle **has** a pin (its center) and a radius
  - A square **has** a pin (top left corner) and a side length
  - A rectangle **has** a pin (top left corner), a width, and a height

Is there anything in common for all the shapes above?

How can we abstract this in a common class?

Abstracting Common Behavior and Fields

```
public abstract class AbstractShape implements Shape {
    private Point2D pin;
    public AbstractShape(Point2D pin) {
        this.pin = pin;
    }
}
```

```
public class Circle extends AbstractShape {
    private int radius;
    public Circle(Point2D pin, int radius) {
        super(pin);
        this.radius = radius;
    }
}
```

Shape Interface

Remember the interface?

Circle must implement the below methods.

```
/** Interface for all shapes on canvas */
public interface Shape {
    void moveX();
    void moveY();
    double area();
    double circumference();
}
```

‘Is-a’ relationships: Inheritance

Java, C++, and others support 'Is-a' relations with inheritance.

- Class inheritance uses `extends` keyword
- **subclass** object *is a* **superclass** object

Good practice

- Code reuse (DRY)
- More robust & maintainable code

Inheritance

"Derived class" or "subclass"

- "Inherits" all public or protected instance variables and methods of "base" class
  - Private fields/methods are not accessible
  - Subclassing does not break encapsulation
- Subclass can add additional methods and instance variables
- Subclass can provide different versions of inherited methods (override)
  - Use `@Override` annotation for JavaDoc
  - Signature must remain the same (except: may widen access (e.g., from protected to public), may return a subtype of type returned by overridden method)
  - Overridden method can be called with `super.<method-name>(<args>)`

Member Access in Subclasses

- **public**: accessible anywhere the class can be accessed
- **private**: accessible only from within the same class (not accessible to subclasses)
- **protected**: accessible inside the defining class and all of its subclasses. Also accessible to other classes in the same package.
- **no modifier (aka 'package private')**: accessible to other classes in the same package. Not accessible to sub-classes defined in other packages.

What is the Type of an Object?

Types in Java

1. **Primitive**: int/long, char, boolean, floating point (double, float)
2. **Non-primitive**: class types, interface implementations, arrays ...

Any instance of a class named `Example` conforms to these types:

1. The named class (`Example`)
2. Every interface that `Example` implements
3. Every superclass that `Example` extends directly or indirectly

An instance can be used anywhere one of its types is appropriate.

Abstract Classes

```
public abstract class AbstractShape implements Shape {
    private Point2D pin;
    public AbstractShape(Point2D pin) {
        this.pin = pin;
    }
}
```

Abstract classes **cannot** be instantiated.

So, something like this:

```
AbstractShape myShape = new AbstractShape();
```

is **illegal**

Abstract Classes

Observation - although this is illegal:

```
AbstractShape myShape = new AbstractShape();
```

Both of these are legal:

```
AbstractShape shape;
AbstractShape[] shapes;
```

Interfaces vs Inheritance

	Interface	Inheritance
Is-A relationship	Yes	Yes
Code sharing	No (Yes in Java 8 with `default`)	Yes
B → A	**B** *implements* interface **A. B** inherits the method signatures from **A** and *must implement them*. **A** provides **specification**.	**B** *extends* **A**. **B** inherits everything from **A** (including method code & instance variables). **B** may generally override (unless declared in **A** as `final`). **A** provides **implementation**.

Both specify a type

Which to Use?

ABSTRACT CLASS

- A class can extend at most one superclass (abstract or not)
- Can include instance variables
- Can include `private` and `protected` access modifiers
- Can specify constructors, which subclasses can invoke with `super`

INTERFACE

- A class can implement any number of interfaces
- Separates behavior from state (methods only)
- Fewer constraints on algorithms and data structures
- Can be tedious to implement interfaces with many method specs (non-default implementations can't be shared across implementing classes)

Practice

Interfaces from Java 8 on

- Before Java 8: pure specification
- Starting from Java 8
  - `default` implementations
  - Static methods (never inherited)
    - *They are part of the type, not the instance*

Extending Interfaces

An interface can extend (an)other interface(s).

1. Don't mention base interface's default → extended interface inherits default method
2. Redeclare base interface's default → makes it into an abstract method
3. Redefine base interface's default → overrides it as a default for the extended interface

What happens when more than one base interface implements conflicting defaults?

Default Implementations for Interface

```java
public interface A {
    default void foo() {
        System.out.println("Default for A");
    }
}

public interface B {
    default void foo() {
        System.out.println("Default for B");
    }
}

public class ClassAB implements A, B {
    ...
}
```

Default Implementations for Interface

Conflicting implementations of defaults lead to the "diamond problem"

Class will fail to compile with the following error:

```
java: class ClassAB inherits unrelated defaults for foo() from
types A and B
```

To avoid this, ClassAB *must* override the conflicted method.

Default Implementations for Interface

```java
public interface A {
    default void foo() {
        System.out.println("Default for A");
    }
}

public interface B {
    default void foo() {
        System.out.println("Default for B");
    }
}

public class ClassAB implements A, B {
    public void foo() {
        System.out.println("Implemented for AB")
    }
}
```

Default Implementations for Interface

If we wanted specifically to inherit A's `foo` method, we can access it via `super`.

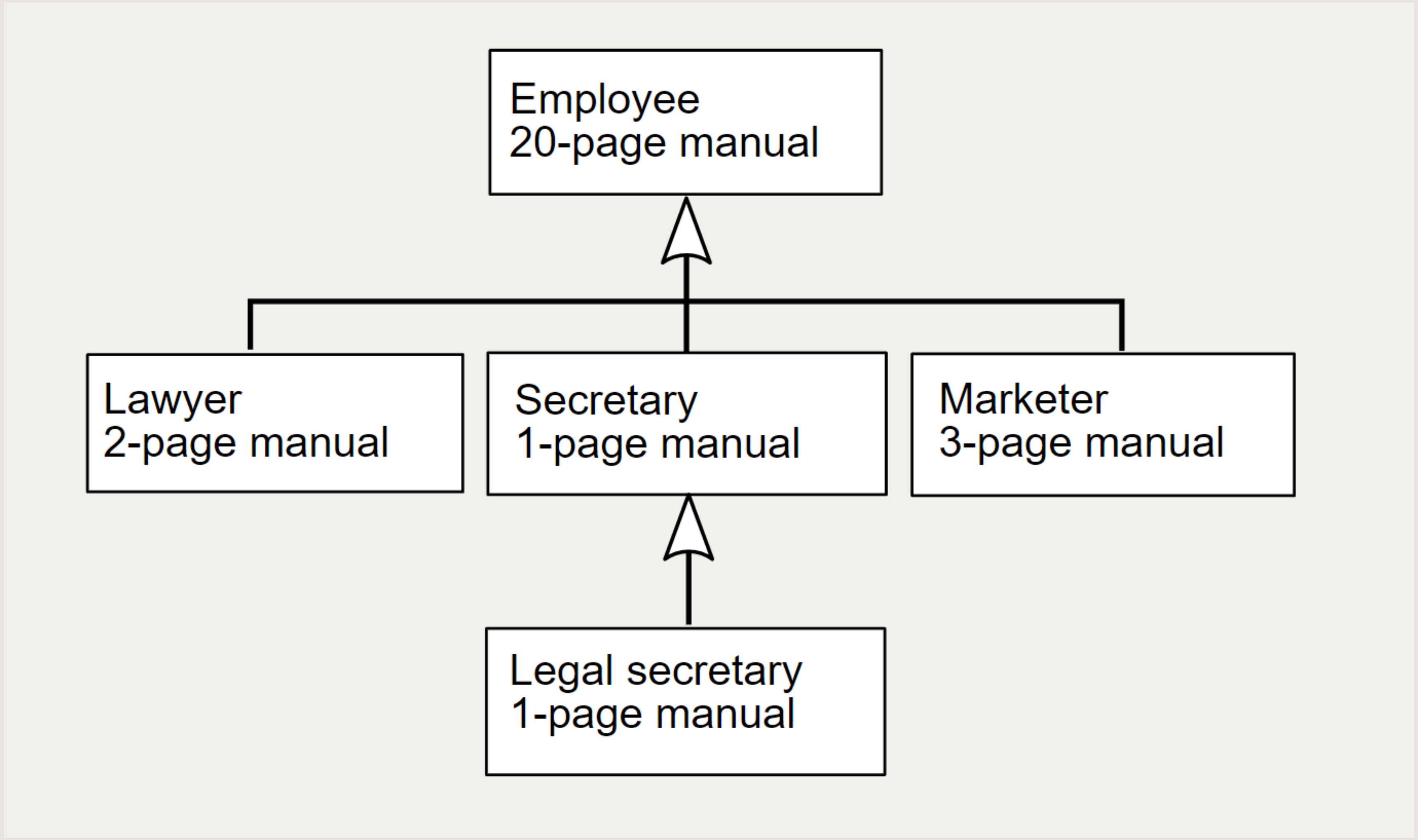
```
public class ClassAB implements A, B {
    public void foo() {
        A.super.foo()
    }
}
```

Design Strategy

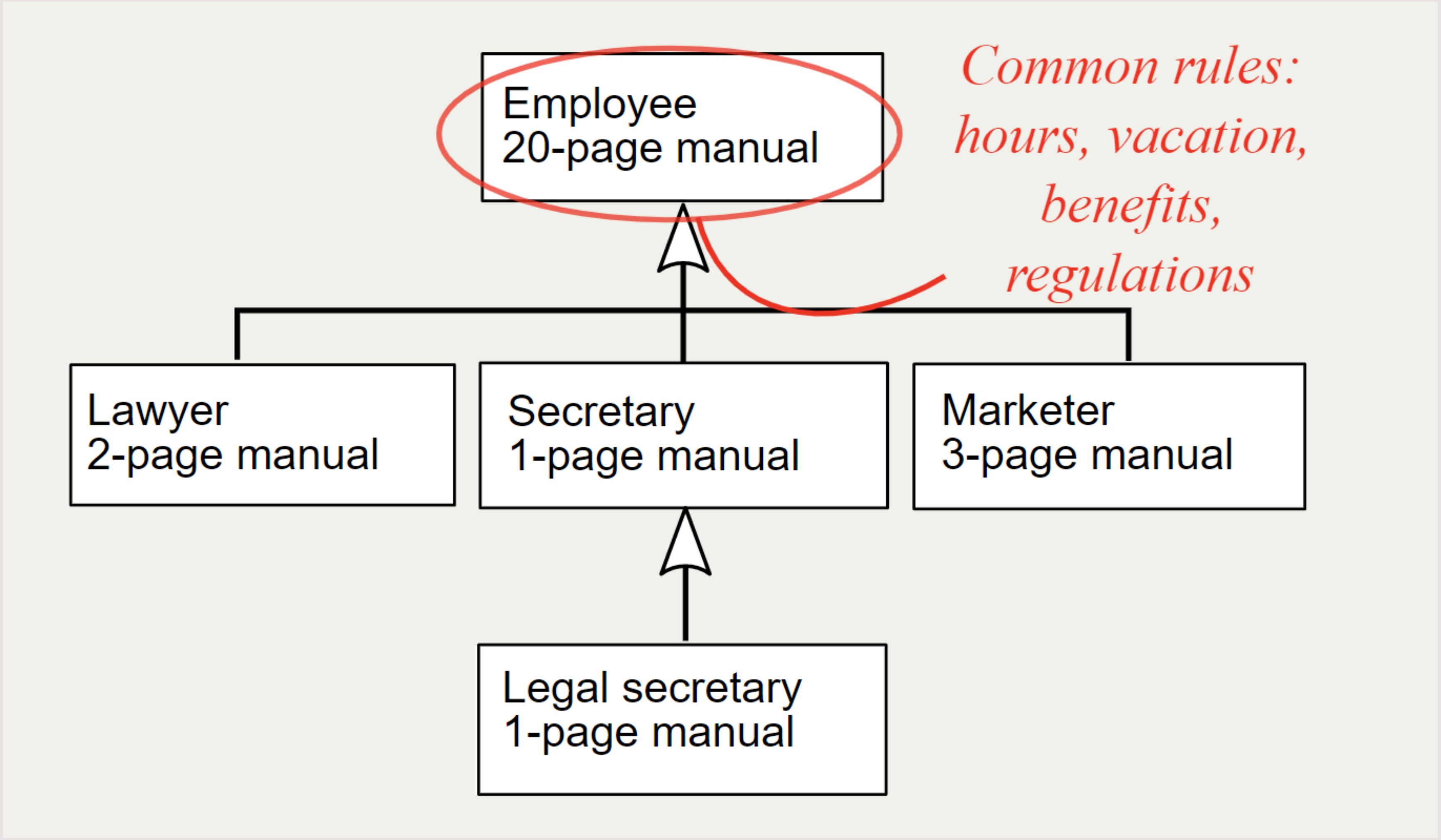
Rules of thumb to design software that can evolve over time.

- Contract to client should be captured by an interface
- Major types should be defined in an interface
- Abstract classes for common fields/behaviors
- Client code can choose to:
  - Extend abstract class, overriding where necessary
  - Implement interface directly (necessary if the class already has a superclass)
- This pattern occurs frequently in standard Java libraries

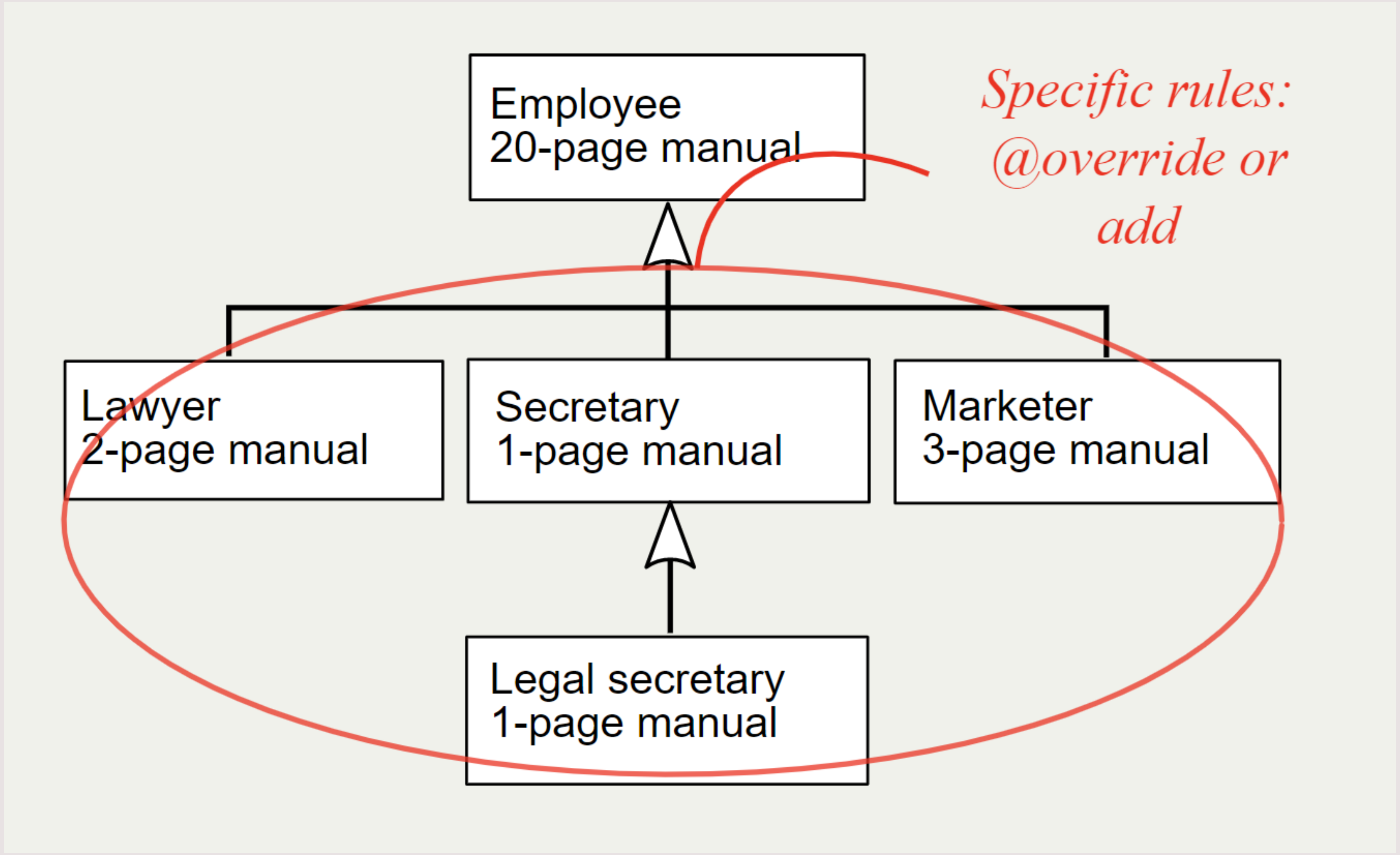
Inheritance Design example: Law Firm Employee



Inheritance Design Example: Law Firm Employee



Inheritance Design Example: Law Firm Employee



Separating Behavior

Advantages of the separate manuals:

- Maintenance: Only one update if a common rule changes
- Locality: Quick discovery of all rules specific to e.g., lawyers

Key ideas from this example:

- General rules are useful (the 20-page manual)
- Specific rules that may override general ones are also useful (it's worthwhile to print the others)

Employee Regulations

	Work hours per week	Salary per year	Paid vacation per year	Color of the leave application form
All Employees	40	$40,000	2 weeks	Yellow
Exceptions		Legal secretaries: +$5,00 ($45,000) Marketers: +$10,000 ($50,000)	Lawyers +1 week (3 total)	Pink

An Employee Class

```java
public class Employee {
    public int getHours() {
        return 40; // works 40 hrs / week
    }

    public double getSalary() {
        return 40000.0; // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10; // 2 weeks paid vacation
    }

    public String getVacationForm() {
        return "yellow"; // use the yellow form
    }
}
```

Employee Unique Behaviors

- Lawyers know how to sue
- Marketers know how to advertise
- Secretaries know how to take dictation
- Legal secretaries know how to prepare legal documents

Inheritance for Code Sharing

A class to represent secretaries:

```java
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: "+ text);
    }
}
```

Inheritance for Code Sharing

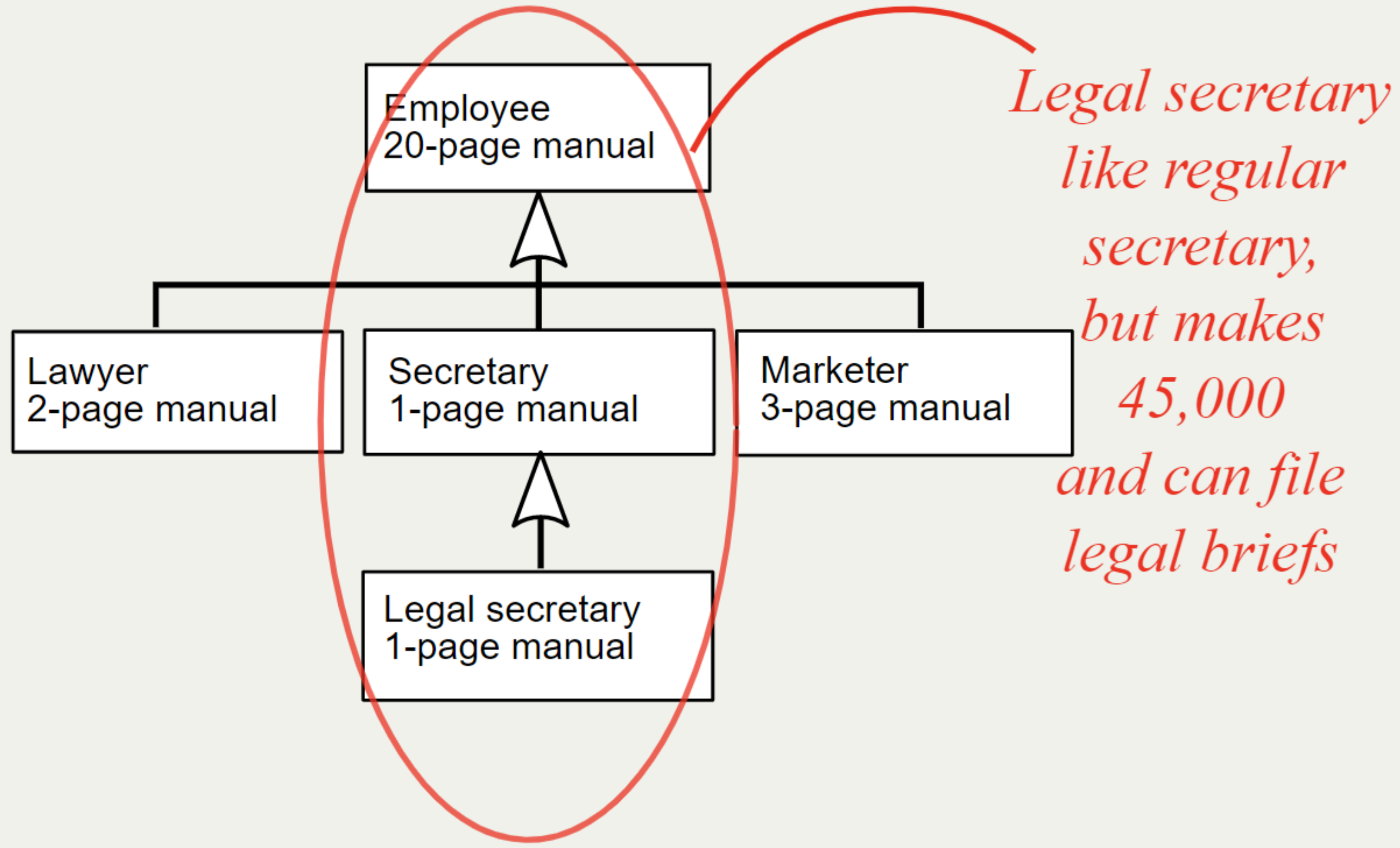
A class to represent lawyers:

```
public class Lawyer extends Employee {
    public int getVacationDays() {
        return 15; // 3 weeks paid vacation
    }

    public String getVacationForm() {
        return "pink"; // use the pink form
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

Multiple Levels of Inheritance



Inheritance for Code Sharing

A class to represent legal secretaries:

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 45000.0; // $45,000.00 / year
    }

    public void fileLegalBriefs() {
        System.out.println("Filed a brief!");
    }
}
```

Changes to Common Behavior

- Everyone is given a $10,000 raise due to inflation
  - Base employee salary now $50,000
  - Legal secretaries now make $55,000
  - Marketers now make $60,000
- Code should be modified to reflect this change:
  - `Employee` class
  - Any class that has overridden `getSalary()`

This is a poor design! Subclass salaries are based on employee salary, but the code doesn't reflect this.

After Modification

```
public class Employee {
    ...
    public double getSalary() {
        return 50000.0; // $50,000.00 / year
    }
    ...
```

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ...
```

After Modification

```
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }
    public String getVacationDays() {
        return super.getVacationDays() + 5;
    }
    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

After Modification

```
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Inheritance and Constructors

- Adding more vacation days
  - For each year, award 2 additional vacation days
  - Upon construction of `Employee` object, pass in the number of years the person has been with the company.
- This requires modifying the `Employee` class:
  - Adding new state
  - Adding new behavior

Modified Employee Class

```java
public class Employee {
    protected int years;
    public Employee(int initialYears) {
        years = initialYears;
    }
    public int getVacationDays() {
        return 10 + 2 * years;
    }
    public int getHours() {
        return 40;
    }
    public double getSalary() {
        return 50000.0;
    }
    public String getVacationForm() {
        return "yellow";
    }
}
```

- Subclasses will no longer compile as previously written
- Constructors are NOT inherited. If a superclass has a constructor with parameters, subclasses must have this implemented as well.

Subclass Constructor

Subclasses receive a default constructor that takes no arguments, but this is not valid if a constructor which takes arguments has been implemented on the superclass.

```java
public class Lawyer extends Employee {
    public Lawyer(int years) {
        // calls Employee constructor. Must be first statement in constructor:
        super(years);
    }
    ...
}
```

(If the superclass is overloaded with an explicit constructor that takes no arguments, then the subclass's default constructor is sufficient.)

Inheritance and constructors: rules to remember

1. If no constructor is written, Java automatically assumes a zero-argument constructor. If any constructor is written, Java does not make this assumption.
2. If no call to `super`(...) written in a subclass constructor, Java automatically assumes a zero-argument call to `super`() as the first statement in a subclass constructor. This will yield a compile-time error if no such constructor is defined (implicitly or explicitly).

Modified Secretary Class

- Secretary years of employment are not tracked
- They do not earn extra vacation for years worked

One possibility:

```
public class Secretary extends Employee {
    public Secretary() {
        super(0);
    }
    ...
}
```

But using 0 here could cause some problems. What else could we do?

Inheritance Summary

If class B extends A (inherits from A):

- Class B inherits methods and fields from class A, except:
    - Constructors are not inherited
    - Static methods & fields are not inherited
    - Private data is hidden (protected is accessible)
- Class B can contain additional (new) methods and fields, or override existing
- Class B cannot *delete* inheritied methods or fields

Intro to Polymophism

```
Employee e = new Employee();
```

What methods can we call?

```
Employee h = new Lawyer();
```

What methods can we call?

```
e = h;
```

Can we do this? What’s going on?

Intro to Polymorphism

- If class B implements/extends A...
  - Object B can do anything A can do (because of inheritance)
  - Object B can be used in any context where A is appropriate
- Same code could be used with different types of objects and behave differently with each.

For example: `System.out.println` prints any type of object (each is displayed in its own way)

Intro to Polymorphism

- A variable that can refer to objects of different types is *polymorphic*
- A variable of type T can hold an object of any subclass of T

```
    Employee ed = new Lawyer();
```

- You can call any methods from `Employee` class on `ed`
- When a method is called on `ed`, the `Lawyer` behavior is performed for the method.

```
System.out.println(ed.getSalary()); // 50000.0
System.out.println(ed.getVacationForm()); // pink
```

Static and Dynamic Types

```
Employee ed = new Lawyer();
```

- **Static/compile time type**: the declared type of the reference variable. Used by the compiler to check syntax.
- **Dynamic/run-time type**: the object type the variable currently refers to (can change as the program executes).

Polymorphism & Parameters

You can pass any subtype of a parameter's type.

```java
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer();
        Secretary steve = new Secretary();
        printInfo(lisa);
        printInfo(steve);
    }
    public static void printInfo(Employee empl) {
        System.out.println("Salary: " + empl.getSalary());
        System.out.println("Vac days: " + empl.getVacationDays());
        System.out.println("Vac form: " + empl.getVacationForm());
        System.out.println();
    }
}
```

Polymorphism & Arrays

Arrays of superclass types can store any subtype as elements.

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] e = { new Lawyer(), new Secretary(),
                         new Marketer(), new LegalSecretary() };
        for (int i = 0, i < e.length; i++) {
            System.out.println("Salary: " + e[i].getSalary());
            System.out.println("Vdays: " + e[i].getVacationDays());
            System.out.println("Vform: " + e[i].getVacationForm());
            System.out.println();
        }
    }
}
```

Casting References

- A variable can *only* call that type's *compile-time type* methods...
- Not methods that are specific to the subclass!

```
Employee ed = new Lawyer();
int hours = ed.getHours(); // ok; it's in Employee
ed.sue(); // compiler error!
```

- Reasoning: `ed` can store any kind of employee, but not all of them are able to sue.
- Solution: cast it explicitly

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue(); // ok
((Lawyer) ed).sue(); // shorter version
```

More on Casting

Note that `ed` is already a lawyer! The type of the object does not change. Casting here only tells the compile not to choke on non-employee method calls.

```
Employee ed = new Lawyer();
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue(); // ok
((Lawyer) ed).sue(); // shorter version
```

Whereas

```
Employee ed = new Employee();
Lawyer theRealEd = (Lawyer) ed;
```

Will `not` generate a compiler error (casting tells it not to) but *will* generate a runtime error, because ed is not a lawyer.

More on Casting

Casting is telling the compiler: "Don't worry about it, I know what I'm doing. This thing may not look like an X, but trust me, it's an X".

Generally, it's best to avoid making the compiler trust us.

Avoid program designs that rely on casting.

Dynamic Dispatch

- "Dispatch": placing a method in execution at run-time
- For run-time (dynamic) type, which version of the method do we use?
  - Decision deferred to run-time → **dynamic dispatch**
  - Chosen method matches dynamic (actual) type of object

Dynamic Dispatch Algorithm

- When a message is sent to an object to run a method, the right method to run is the one in the *most specific class* that the object is an instance of.
  - Ensures that method overriding always has an effect
- Method lookup (dynamic dispatch) algorithm:
  - Start with the actual run-time class (dynamic type) of receiver object
  - Search that class for matching method
  - If one is found, invoke it
  - Otherwise, go to the super class and continue searching

Related Topics

- toString()
- instanceof

```
if (otherObject instanceof Blob) {
    Blob bob = (Blob) otherObject;
    ...
}
```

Many Forms of Polymorphism

Three types of polymorphism that we'll explore

- Subtype polymorphism (today)

Coming up next:

- Ad-hoc polymorphism (overloading)
- Parametric polymorphism (Generics)

Overloading Rules

Two (or more) methods are overloaded if:

1. The method name is the same
2. The argument list differs in:
    - The number of arguments
    - The types of the arguments
    - The order of argument types

Multiple methods with identical argument type signatures will not compile.

Return types don't count in distinguishing methods for overloading.

Questions?