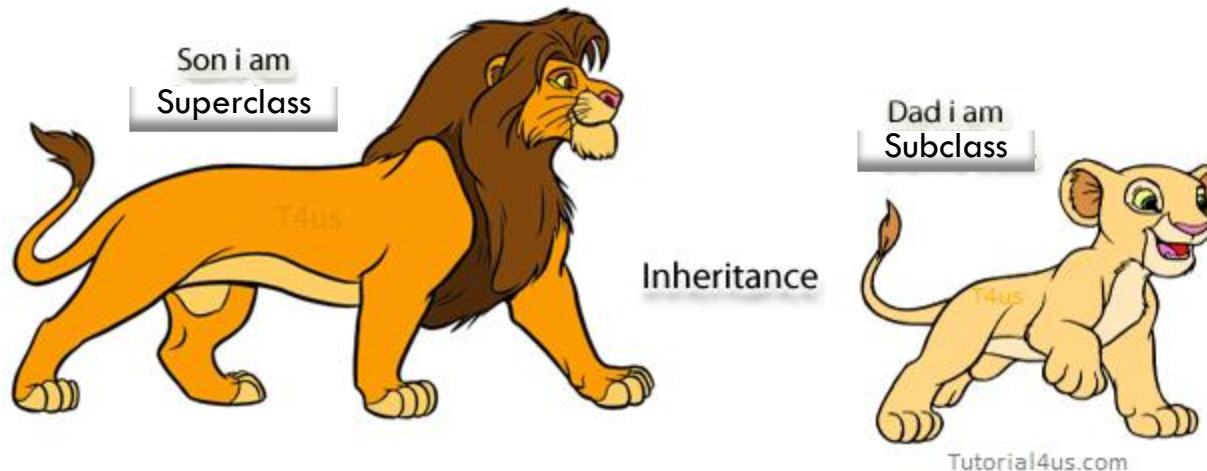


ADTs, Interfaces, Inheritance & Polymorphism

Maria Zontak



Credits: CS143 course I taught in North Seattle College
CS5004 course built by Dr. Therapon Skotiniotis here in Northeastern
Materials from MIT, Software Constructions – 6.005

Abstract Data Type

What ?

- Abstract data types are an instance of a general principle in software engineering
- ADT defines a **contract** for a data type with specific **invariants**

Why?

- **Safe from bugs** - a well-defined ADT preserves its invariants, thus will be less vulnerable to bugs in the ADT's clients. Violations of the invariants can be more easily isolated within the implementation of the ADT itself.
- **Easy to understand** – complex implementation is hidden behind a set of simple operations, so the ADT client only needs to understand the operations, not the details of the implementation.
- **Ready for change** – Representation independence allows the implementation of an ADT to change without requiring changes from its clients.

Who cares?

- The clients only cares about ADT (what each operation does)
- The developer cares about Concrete Implementation and needs to
 - Do the right thing!*
understand what each operation should do.
 - Do the thing right!*
design a program that implements the specification correctly

Bread and Butter of any Software Engineer

- **Abstraction** - Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity** - Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation** - Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.
- **Information hiding** - Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.
- **Separation of concerns** - Making a feature (or “concern”) the responsibility of a single module, rather than spreading it across multiple modules.

Classifying Types

- **Mutable** - The objects of a mutable type can be changed → must provide operations which when executed cause a change of state of this
- **Immutable** – State cannot be changed. For example: `String` is immutable, because its operations create new `String` objects rather than changing existing ones.

Classifying Operations

- **Creators** - create new objects of the type. A creator may take an object as an argument, but NOT an object of the type being constructed. **Schematically:** $t^* \rightarrow T$ [$*$ occurs zero or more time]
- **Producers** - create NEW objects from OLD objects of the type. The For example: `concat` of `String`: it takes two strings and produces a new one representing their concatenation.

Schematically: $T+, t^* \rightarrow T$ [$+$ occurs one or more times]

- **Observers** - take objects of the abstract type and return objects of a different type. For example: `size` of `List`: returns an `int`.

Schematically: $T+, t^* \rightarrow t$

- **Mutators** - change objects. For example: `add` method of `List`, mutates a list by adding an element to the end.

Schematically: $T+, t^* \rightarrow \text{void} | t | T$

ADT examples in Java

int is Java's primitive integer type. **int** is **immutable**, so it has no mutators.

- creators: the numeric literals 0, 1, 2, ...
- producers: arithmetic operators +, -, ×, ÷
- observers: comparison operators ==, !=, <, >
- mutators: none (it's immutable)

List is Java's list type. **List** is mutable. **List** is also an interface → other classes provide the concrete implementation (For example: `ArrayList` and `LinkedList`).

- creators: `ArrayList` and `LinkedList` constructors, `Collections.singletonList`
- producers: `Collections.unmodifiableList`
- observers: `size`, `get`
- mutators: `add`, `remove`, `addAll`, `Collections.sort`

String is Java's string type. **String** is **immutable**.

- creators: `String` constructors
- producers: `concat`, `substring`, `toUpperCase`
- observers: `length`, `charAt`
- mutators: none (it's immutable)

Interface

WHAT?

WHY needed?

UML Diagramm

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

Interface declaration

```
/** Interface for all objects involved in the simulation */  
public interface SimThing {  
    public abstract void tick( );  
    public abstract void redraw( );  
}
```

A method that is
declared but NOT
implemented

It is permitted, but
discouraged (a matter of
style), to redundantly
specify the **public** and/or
abstract modifier for a
method declared in an
interface.

Class declaration using the interface

```
/** Base class for all Ship */  
public class Ship implements SimThing {  
    /** tick method for Ships */  
    public void tick( ) { ... }  
    /** redraw method for Ships */  
    public void redraw( ) { ... }  
}
```

The implementation
comes in the class
that implements the
interface

implements

- Java **interface** declares a set of method signatures
 - Says what behavior exists
 - Does not say how the behavior is implemented (no code for the methods)
 - Does not describe any state (but may include “final” constants)
- **Concrete** class that implements an interface
 - Contains `implements InterfaceName` in the class declaration
 - Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface

What is the Type of an Object?

Types in Java:

1. **Primitive**— int/long, char, boolean and floating point (double, float).
2. **Non-primitive**—Classes, Interfaces, and Arrays.

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

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

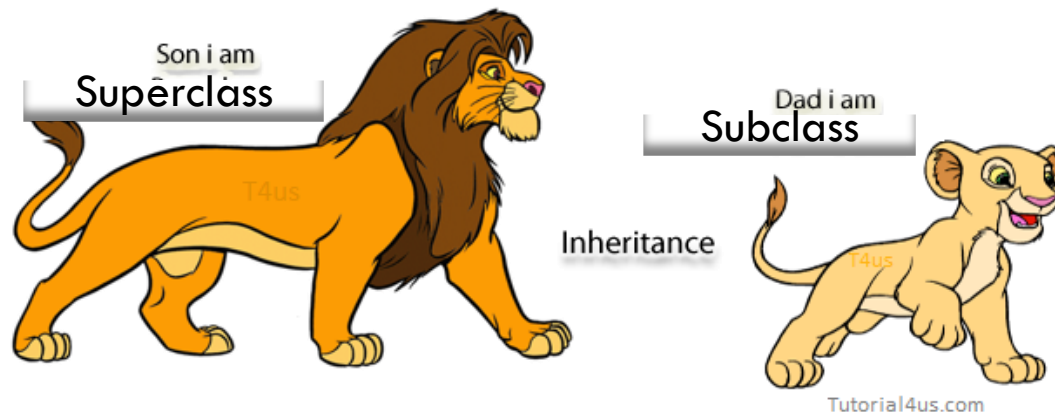
The instance can be used anywhere one of its types is appropriate

- As variables, as arguments, as return values

'Is-a' in Programming: Inheritance

Java, C++ and more provide direct support for "IS - A":

- **Class Inheritance** - new class **extends** existing class
- Key for **good** object-oriented programming:
 - Using the SAME code in MANY contexts → **Reusable code**
 - Reduce bugs → **Robust and maintainable**
- **Terminology:**

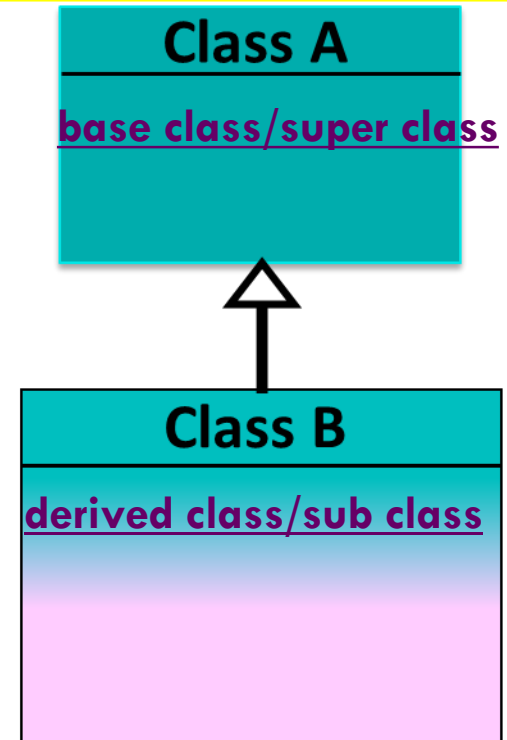


Inheritance - Vocabulary and Principles

Derived/sub class

- Automatically inherits all instance variables and methods of the base class
 - Private fields/methods are not accessible. Why?
 - Reason: subclassing will NOT break encapsulation.
- Can add additional methods and instance variables
- Can provide different versions of inherited methods → override
 - Use the @Override annotation for Javadoc
 - Signature MUST remain the same, excluding:
 - Overriding method can allow more access
 - Overriding method can return a subtype of the type returned by the overridden method.
 - To call overridden method use **super.<method-name>(<args>)**

UML for B extends A



Member Access in Subclasses

What kind of field/method modifiers do you know?

- **public:**
- **private:**
- **protected**

What is the Type of an Object?

Types in Java:

1. Primitive— Integer, Character, Boolean, and Floating Point.
2. Non-primitive—Classes, Interfaces, and Arrays.

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

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

The instance can be used anywhere one of its types is appropriate

- As variables, as arguments, as return values

Interfaces vs Inheritance

	INTERFACE	INHERITANCE
“Is –A” Relationship	V	V
Code Sharing	X (V only in Java 8 and on)	V
B → A	B <i>implements</i> interface A → B inherits the method signatures from A (must implement them) Specification	B <i>extends</i> class A → B inherits everything from A (including any method code and instance variables) Implementation

Both specify a type

Abstract Class

- A class can extend at most one superclass (abstract or not)
- Can include instance variables
- Wider range of modifiers
- Can specify constructors, which subclasses can invoke with *super*

Interface

- A class can implement any number of interfaces
- Keeps state and behavior separate
- Provides fewer constraints on algorithms and data structures
- Interfaces with many method specifications are tedious to implement (implementations cannot be inherited)

Interfaces in Java 8 and on

- Before Java 8 - Pure specification
- Starting from Java 8 :
 - default (“defender”)
 - static methods (never inherited)

'default' Implementation for Interface

Example:

<https://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html>

How to extend an interface (**yes! interface can extend interface**) that contains a default method?

1. Do NOT mention the default method at all
→ extended interface **inherits** the default method
2. Redecclare the default method → makes it **abstract**.
3. Redefine the default method → **overrides** it.

????????????????????????????????????

- How to access overridden method?
- What if the class implements two interfaces and both those interfaces define a default method with the same signature?

'default' Implementation for Interface

```
public interface A {  
    default void foo() {  
        System.out.println("implements A");  
    }  
}  
  
public interface B {  
    default void foo() {  
        System.out.println("implements B");  
    }  
}  
  
public class ClassAB implements A, B {  
    ...}
```

→ DIAMOND PROBLEM

fails to compile with the following result:

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

'default' Implementation for Interface

To fix that, in ClassAC override the conflicting method:

```
public class ClassAB implements A, B {  
    public void foo(){  
        System.out.println("implements A & B");  
    }  
}
```

You can directly call the default implementation of method foo() from interface A (or interface B)

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

Allow multiple
inheritance ONLY when
you have no other choice

A Design Strategy

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

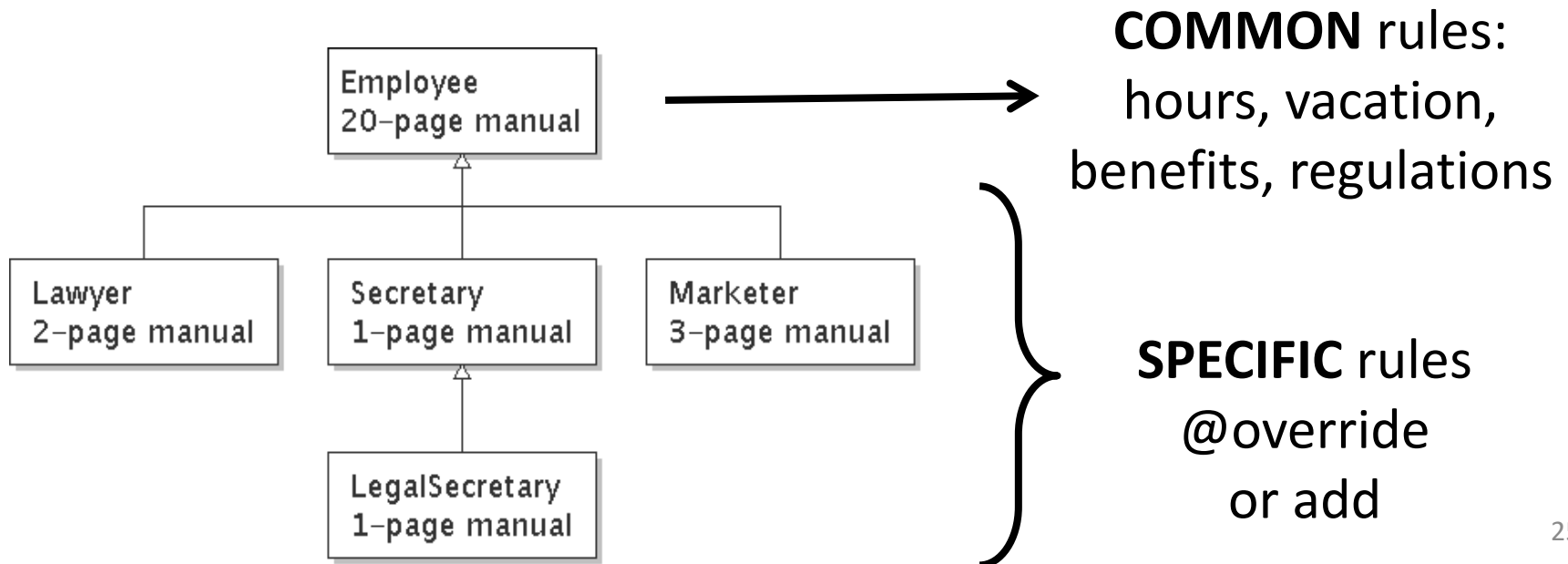
- Your contract to client should be captured by an interface
- Any major type should be defined in an interface
- Abstract out/implement possible common fields/behaviors of the interface using abstract class
- Client code can choose:
 1. Extend the abstract class implementation, overriding methods that need to be changed
 2. Implement the complete interface directly (needed if the class already has a specified superclass)
- This pattern occurs frequently in the standard Java libraries

Additional Example of Inheritance Design: Law firm employee

Goal: demonstration of additional Java rules for inheritance

Disclaimer: The below implementation omits interfaces.

Generally you should make an interface for any major type



Separating behavior

- Advantages of the separate manuals:
 - Maintenance: Only one update if a common rule changes.
 - Locality: Quick discovery of all rules specific to lawyers.
- Key ideas from this example:
 - General rules are useful (the 20-page manual).
 - Specific rules that may override general ones are also useful.

Employee regulations

	Work Hours per week	Salary (\$) per year per employee	Paid Vacation per year	Color of the leave application form
All Employees	40	40,000	2 weeks	Yellow
Exceptions		<u>Legal Secretaries:</u> +5,000 (45,000) <u>Marketers:</u> +10,000 (50,000)	<u>Lawyers:</u> + 1 weeks (3 weeks in total)	<u>Lawyers:</u> Pink

An Employee class

```
// A class to represent employees in general (20-page
manual) .
public class Employee {
    public int getHours() {
        return 40;                // works 40 hours / 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 as a mechanism for Code Sharing

```
// A class to represent secretaries.
```

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

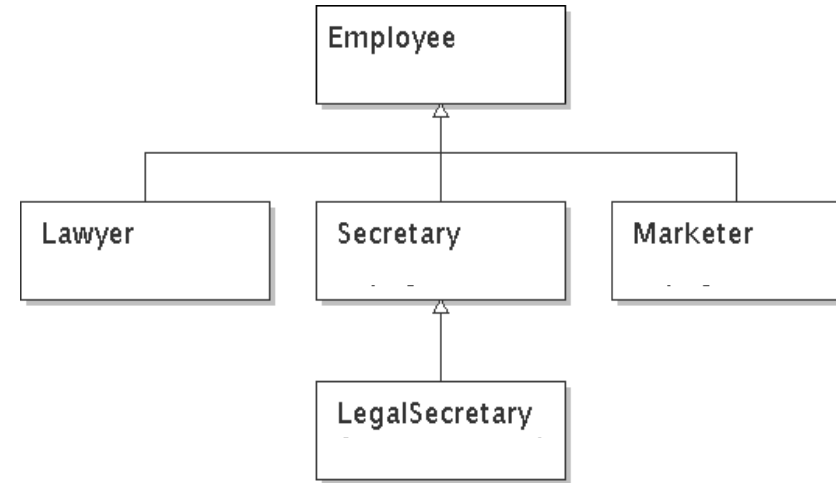
```
// A class to represent lawyers.
```

```
public class Lawyer extends Employee {  
    // overrides getVacationForm from Employee class  
    public String getVacationForm() {  
        return "pink";  
    }  
  
    // overrides getVacationDays from Employee class  
    public int getVacationDays() {  
        return 15;           // 3 weeks vacation  
    }  
  
    public void sue() {  
        System.out.println("I'll see you in court!");  
    }  
}
```

Multiple Levels of inheritance

Example - Legal secretary:

- Same as a regular secretary
- Unique behaviours:
 - Makes more money (\$45,000)
 - Can file legal briefs.



// A class to represent legal secretaries.

```
public class LegalSecretary extends Secretary {

    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }

    public double getSalary() {
        return 45000.0;           // $45,000.00 / year
    }

}
```

Changes to common behavior

- Everyone is given a \$10,000 raise due to inflation.
 - The base employee salary is now \$50,000.
 - Legal secretaries now make \$55,000.
 - Marketers now make \$60,000.
 - Code should be modified, to reflect this policy change.
 - Following our previous design we need to modify:
 - `Employee` class
 - Every subclass that has overridden `getSalary()`
- This is a poor design. Why?

Inheritance and constructors

Adding more vacation days:

- For each year - award 2 additional vacation days.
- Upon construction of the `Employee` object - pass in the number of years the person has been with the company.

→ This requires to modify the `Employee` class:

- add new state
- add new behavior

Inheritance and constructors - RULES to remember:

1. NO constructor is written in a class → Java assumes there is an empty, zero argument constructor `ClassName () { }`
 - If you write any constructor, Java does not make this assumption
 2. `super (...)` is not written as the first line of an extended class constructor → the compiler assumes the constructor starts with a call to `super ()` – default super constructor;
- BE SPECIFIC in your code for readability

Upon construction of an extended class object

→ there must be a constructor in the parent class with parameter list that matches the explicit or implicit call to `super (...)`

- **Corollary:** a constructor is always called at each level of the inheritance chain when an object is created

Inheritance Summary

If class B extends A (inherits from A) ...

- Class B inherits all methods and fields from class A
- But... "all" is too strong
 - constructors are **not** inherited but there is a way to use super class constructors during object creation
 - same is true of static methods and static fields
 - private data is hidden from derived class implementation but can access through get/set methods from base class (if they exist!) OR just set it to be **protected**
- Class B may contain additional (new) methods and fields
 - Has no way to delete any

Intro to Polymorphism

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

What methods can we call?

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

What methods can we call?

```
e = h;
```

Is this legal? What's going on here?

Northeastern University

An Employee class

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / 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
    }
}
```

28

Northeastern University

Lawyer class



- Lawyer's regulations:
 - Extra week of paid vacation (a total of 3) → @override
 - Use pink form when applying for vacation leave → @override
 - Unique behavior - know how to sue.

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;           // 3 weeks vacation
    }

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

04-31

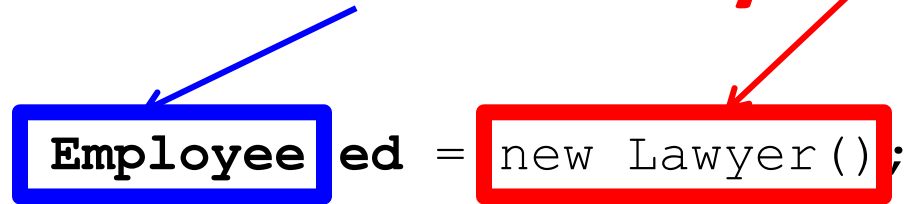
Intro to Polymorphism

- If class B implements/extends A...
 - Object B can do anything that 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 on the console) .

- A variable that can refer to objects of different types is said to be *polymorphic*
 - **Employee ed** = new Lawyer();
A variable of type *T* can hold an object of any subclass of *T*.

Static and Dynamic Types




The diagram illustrates the concepts of static and dynamic types using a code snippet: `Employee ed = new Lawyer();`. A blue box highlights the word `Employee`, with a blue arrow pointing from the word `Static` in the title above. A red box highlights the expression `new Lawyer();`, with a red arrow pointing from the word `Dynamic` in the title above.

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

Polymorphism and parameters

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

```
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("v.days: " + empl.getVacationDays());  
        System.out.println("v.form: " + empl.getVacationForm());  
        System.out.println();  
    }  
}
```

A diagram consisting of two arrows. The first arrow originates from the **printInfo(lisa);** line in the `main` method and points to the `Employee empl` parameter in the `printInfo` method signature. The second arrow originates from the **printInfo(steve);** line in the `main` method and also points to the `Employee empl` parameter in the `printInfo` method signature. This illustrates that both `Lawyer` and `Secretary` objects can be passed to a method that expects an `Employee` parameter.

Polymorphism and 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("v.days: " + e[i].getVacationDays());  
            System.out.println();  
        }  
    }  
}
```

Casting references

- A variable can only call that type's (compile-time type) methods, not a subtype's (runtime type).

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

- Compiler's reasoning :variable `ed` could store any kind of employee, and not all kinds know how to `sue`.
- To use `Lawyer` methods on `ed` → cast it.

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

As a good practice – AVOID creating design where you need to cast.

More about casting

- Do **NOT** cast an object **too far down** the tree.

```
Employee eric = new Secretary();  
((Secretary) eric).takeDictation("hi");           // ok  
((LegalSecretary) eric).fileLegalBriefs();         //  
exception  
// (Secretary object does not know how to file briefs)
```

- Do **NOT** cast **sideways** (only up or down casting works).

```
Lawyer linda = new Lawyer();  
((Secretary) linda).takeDictation("hi");           // error
```

- Casting does **NOT** actually **CHANGE** the object's behavior.

It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()               // pink  
(Lawyer's)
```

Dynamic Dispatch - Summary

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

- “Dispatch” - the act of actually placing a method in execution at run-time
- **Static** types - the compiler knows exactly what method must execute
- **Dynamic** types - the compiler knows the *name* of the method but...

There could be ambiguity about which version of the method will actually be needed at run-time:

- The decision is deferred until run-time → **dynamic dispatch**
- The chosen method matches the dynamic (actual) type of the object

Method Lookup: How Dynamic Dispatch Works

- When a message is sent to an object, 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 (a.k.a. dynamic dispatch) algorithm:
 - Start with the actual *run-time class (dynamic type)* of the receiver object (not the static type!)
 - Search that class for a matching method
 - If one is found, invoke it
 - Otherwise, go to the super class, and continue searching

Related Dynamic Dispatch Topics

toString()

instanceof: <object> instanceof <classOrInterface>

– checking types of generic objects before casting

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

super is NOT polymorphic

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.11.2>

DIT – continue to improve Lawyers firm design

Here are a few suggestions:

- What about `getPay()` ?
- Must be included in `Employee`, so polymorphic code can use it
- ```
/** Return the pay earned by this employee */
public double getPay() {return 0.0;}
```

Does this make sense? Fix at home!!!

Hint: go back to `Shape` example...

You can also add interfaces as discussed...

# Many Shapes to Polymorphism

There are 3 kinds of polymorphism that we will explore:

- **Subtype polymorphism** (we saw today)

Coming next week:

- **Ad-hoc polymorphism** (or overloading)
- **Parametric polymorphism** (or Generics in Java)