Chapter 14

Inheritance

So far, we have used interfaces to define the *contract* between an abstract data type and its implementation. In the context of Java's type system, interfaces also induce a *subtyping* relationship between interface and implementor type where we say that the implementor type *is a* interface type. This allows us to substitute an instance of the implementor where ever the interface type is required. We called this flexibility *(subtype) polymorphism*: the ability to write code that works with many types.

Interfaces are a powerful, fundamental mechanism for establishing such a subtyping relationship. However, they carry with them an important limitation: we cannot provide any implementation to the implementor of the interface. While the intent of an interface is that implementors provide different implementation, sometimes we either want to (1) share parts of the implementation or (2) provide some default implementation that implementors are free to *override* if desired. To do this, we require another mechanism for *sharing implementation* in addition to establishing an *"is a"* relationship between types.

14.1 An Example: Shapes

Suppose that we need to represent a collection of shapes such as squares, circles, and triangles in our program. We might design an interface that represents a shape:

```java
// Shape.java
/**
 * A shape, e.g., a square, circle, or triangle.
 */
public interface Shape {
    /**
     * @return the x-coordinate of this shape
     */
    public int getX();

    /**
     * @return the y-coordinate of this shape
     */
    public int getY();

    /**
```

```
     * @return the width of this shape
     */
    public int getWidth();

    /**
     * @return the height of this shape
     */
    public int getHeight();

    /**
     * @return the area of this shape
     */
    public int getArea();
}
```

As well as provide some implementations of this interface:

```
// Rectangle.java
public class Rectangle implements Shape {
    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public int getArea() { return width * height; }
}

// Circle.java
public class Circle implements Shape {
    private int x;
    private int y;
    private int radius;

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

    public int getX() { return x; }
    public int getY() { return y; }
    public int getWidth() { return 2 * radius; }
    public int getHeight() { return 2 * radius; }
    public int getArea() { return (int) (Math.pi * (Math.pow(radius, 2))); }
}
```

Naturally, Square and Circle have different sorts of representation as they are different shapes. However, they have something in common: the representation of their position as a coordinate pair. Furthermore, the implementation of this coordinate pair is the same—two fields and two getters for each of the components of the pair. Because an interface does not let us provide implementation details, we have to duplicate this code in Square and Circle which is undesirable.

Instead of creating an interface to represent the abstract data type of a shape, we'll instead create an *abstract class* for this purpose:

```
/**
 * A shape, e.g., a square, circle, or triangle.
 */
public abstract class Shape {
    private int x;
    private int y;

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

    /**
     * @return the x-coordinate of this shape
     */
    public int getX() { return x; }

    /**
     * @return the y-coordinate of this shape
     */
    public int getY() { return y; }

    /**
     * @return the width of this shape
     */
    public abstract int getWidth();

    /**
     * @return the height of this shape
     */
    public abstract int getHeight();

    /**
     * @return the area of this shape
     */
    public abstract int getArea();
}
```

A class marked `abstract` sits between an `interface` and a `class`:

- An abstract class cannot be instantiated. In other words, the expression `new Shape(0, 0)` will give a compiler error complaining that you are trying to instantiate an abstract class. This makes sense because `Shape` is an "abstract" entity, *i.e.*, shapes do not exist but instead there are squares, triangles, circles, *etc.*.
- Because an abstract class is a class, it can provide implementation like a normal class.
- An abstract class can optionally mark methods as `abstract`. An `abstract` method does not have an implementation; a class that *extends* this abstract class will need to provide an implementation of this method.

Like interfaces, we can declare a subtype relation between an abstract class and a normal class. However, rather than *implementing* an interface, we say that the (normal) class *extends* the abstract class.

```
// Rectangle.java
public class Rectangle extends Shape {
    private int width;
    private int height;

    public Rectangle(int x, int y, int width, int height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public int getArea() { return width * height; }
}

// Circle.java
public class Circle implements Shape {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }
    public int getWidth() { return 2 * radius; }
    public int getHeight() { return 2 * radius; }
    public int getArea() { return (int) (Math.PI * (Math.pow(radius, 2))); }
}
```

By extending the Shape abstract class (using the `extends` clause on the class declaration), both Circle and Rectangle *inherit* the state and behavior defined by Shape. We say that Shape is the *parent class* or *super class* of `Circle` and `Rectangle`. Conversely, we say that Circle and Rectangle are *subclasses* of Shape. Namely, they both possess two fields, `x` and `y` of type `int` and two getter methods `getX()` and `getY()`. To extend the `Shape` class, `Rectangle` and `Circle` must provide implementations for the abstract methods `getWidth()`, `getHeight()`, and `getArea()`.

However, since we inherit these fields, how do we initialize them? Note that because `x` and `y` are both declared as `private` in `Shape`, we cannot access them from `Square` or `Circle`. This does not mean we do not inherit them, it just means that their *names* are not visible from within `Square` or `Circle`. To initialize these fields, we must access the constructor of the `Shape` class. To do this, we use a *super class constructor invocation* as the first line of the constructors of `Circle` and `Rectangle`. The syntax of a super class constructor invocation is:

```
super(<expr>, ..., <expr>);
```

You can think of `super` as standing in for the name of the super class. For example, in `Circle`, we invoke the `Shape` constructor that takes values to initialize `x` and `y` with the call `super(x, y)`.

By extending `Shape`, `Rectangle` and `Circle` are both considered subtypes of `Shape`, just like with an interface. In particular, the following code snippet works as you'd expect:

```
Shape shape1 = new Square(0, 0, 100, 100);
Shape shape2 = new Circle(10, 10, 100);
System.out.println(shape1.getArea());    // 100 * 100
System.out.println(shape2.getArea());    // pi * 100^2
```

We can also extend normal classes, not just abstract classes. For example, if we wish to specialize `Rectangle` further to a `Square`, we can extend `Rectangle`:

```
public class Square extends Rectangle {
    public Square(int x, int y, int length) {
        super(x, y, length, length);
    }
}
```

This might seem redundant; we could simply create a square by specifying a `Rectangle` with equal width and height. However, by encoding a `Square` as a class, the type checker can now ensure that the user provides squares when necessary. Thus we turn what would otherwise be runtime errors into compile time errors.

14.2 Restrictions on Class Inheritance

After walking through this example, it seems like class inheritance is strictly more powerful than interface extension. In addition to providing `abstract` methods, we can also provide implementation—both fields and methods—with an abstract class. So what's the downside?

The downside is that a class may only *extend a single class*, but it may *implement multiple interfaces*. Java elects to be a *single-inheritance object-oriented programming language* because it turns out there are lots of complications involved with *multiple inheritance*, *e.g.*, what happens if two super classes provide a field or method with the same name? Rather than dealing with these problems (the *diamond problem*), Java tries to keep things simple by requiring that a class can only inherit from at most one super class.

14.3 Visibility and Dynamic Dispatch

Previously, we explored a particular case study of using *inheritance* over interface implementation: we want to share some implementation details between subclasses. It turns out that this one addition complicates the mechanics of our program greatly. We'll now turn our attention to looking at this mechanics in detail, in particular, three language features:

1. Visibility with subclasses.
2. Accessing and invoking subclass state and behavior.
3. Overriding subclass behavior.

14.4 Visibility with Subclasses

Recall that a member marked `private` was only accessible to code *within* the declaring class, for example,

```java
public class C {
    private int x;
    private int foo() { return 0; }
    public C() { this.x = 5; }
    public static void test() {
        C c = new C();
        System.out.println(c.x);      // 5
        System.out.println(c.foo());  // 0
    }
}

class Bad {
    public static void test() {
        C c = new C();
        System.out.println(c.x);      // error: x not visible
        System.out.println(c.foo());  // error: foo not visible
    }
}
```

We use privacy modifiers to hide implementation details from clients of a class. However, subclasses (established with inheritance and `extends`) sit somewhere between "client" and "author". On one hand, they are a client of their superclass in the sense that they are using its functionality. On the other hand, it is plausible that they will need direct access to some of these implementation details, *e.g.*, to update a field based on the subclass's refine behavior.

Therefore, we need a refined accessibility modifier that allows subclasses to access superclass members without exposing those members to everyone. This accessibility modifier is `protected`. For example, consider modifying a subset of our Shape class from before with `protected` members:

```java
public abstract class Shape {
    protected int x;
    protected int y;

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

    /**
     * @return the x-coordinate of this shape
     */
    public int getX() { return x; }

    /**
     * @return the y-coordinate of this shape
     */
    public int getY() { return y; }
}
```

Now, our `Rectangle` subclass can modify the `x` and `y` fields directly:

```java
public class Rectangle {
    // ...
    public void translate(int dx, int dy) {
        this.x += dx;
        this.y += dy;
    }
}
```

So why not make all our variables `private` instead of `protected`? Again, this becomes a question of *least priviledge*. Exposing, in particular, a field allows us to perform two operations:

1. Writing to the field.
2. Reading from that field.

If a superclass does not wish to allow subclasses both of these behaviors, then the field should be `private` instead of `protected`. This is desirable because if there's some *invariant* that exists between the superclass's fields (*e.g.*, in our `ArrayStack` implementation, that the `top` field always pointed to the top of the stack in the `data` array), we would like to not give a subclass the opportunity to mess it up.

As a rule of thumb, if you predict that your subclasses will not need to both read from and write to a field (and you do not need to "wrap" the reading/writing, *e.g.*, to perform a pre-condition check), then that field should be marked `private` rather than `protected` to prevent this behavior. This is the same logic that you should apply to determining if a field ought to be `public`. In practice, this means few of your fields will be marked `protected` or `public` because you will rarely want to expose fields in this manner.

14.5 Overriding Subclass Behavior

One of the important benefit of extending a class is to inherit its (public and protected) fields and methods. However, what if we wish to *override* the behavior of one of these methods? The canonical example is a superclass that provides some default behavior with the intention that a subclass provide more refined behavior. As a toy example, consider a small class hierarchy for animals.

```
public abstract class Animal {
    private boolean alive;
    public Animal() {
        this.alive = true;
    }
    public boolean isAlive() { return alive; }
    public void makeNoise() { System.out.println("Ding"); }
}

public class Dog extends Animal {
    public Dog() { /* Implicitly calls super() */ }
}

public class Cat extends Animal {
    public Cat() { /* Implicitly calls super() */ }

    @Override
    public void makeNoise() { System.out.println("Meow"); }
}

public class Lion extends Cat {
    public Lion() { /* Implicitly calls super() */ }

    @Override
    public void makeNoise() { System.out.println("Roar"); }
}
```

Here, we have a *class hierarchy* consisting of four classes. The relationship can be drawn as follows:

```
      Animal
         |
     ---------
     |       |
    Cat     Dog
     |
   Lion
```

Where `Animal` is the top-most superclass, a `Cat` is an `Animal`, a `Dog` is an `Animal` (but is not a cat), and a `Lion` is a `Cat`. `Animal` provides a private field and public method for determining if an `Animal` is alive. `Cat`, `Dog`, and `Lion` all inherit these members.

However, Animal also specifies a method makeNoise with a generic sound suitable as a default for any Animal. The Dog class inherits this generic sound, so the following code produces the expected result:

```
Dog d = new Dog();
d.makeNoise();          // Ding
```

This is, of course, not the sound that a Dog makes. We need some way of providing our own class-specific behavior makeNoise. Cat and Lion both do this by *overriding* the makeNoise method to provide their own implementation. For example, the following code:

```
Cat c  = new Cat();
Lion l = new Lion();
c.makeNoise();          // Meow
l.makeNoise();          // Roar
```

produces refined behavior for makeNoise based on the type of the object we call the method on: "Meow" for cats and "Roar" for lions. To override a method in a subclass, we simply include an implementation of that method in the subclass. However, we also include an *annotation* on this method, @Override, to note that this method declaration intentionally overrides the makeNoise method declared in the superclass. (Note that the @Override annotation is a new addition in Java 1.5 and is technically optional. However, you should always use @Override to be explicit when you are overriding superclass behavior.)

Method overriding seems straightforward. However, things quickly become less clear when we combine method overriding with subtyping. Consider the following variable declarations, method calls, and their results:

```
Animal a1 = new Dog();
Animal a2 = new Cat();
Animal a3 = new Lion();
Cat c = new Lion();
a1.makeNoise();         // Ding
a2.makeNoise();         // Meow
a3.makeNoise();         // Roar
c.makeNoise();          // Roar
```

Note that even though the type of the variable is Animal in the first three cases, we use the *actual type of the object* assigned to the variable to determine which method to call. We distinguish between the two sorts of the types accordingly:

- The *static type* of a value is the type of the value as it is known to the compiler. In the case of variable declarations, this is the type of the declared variable. The static type determines what is the allowable set of methods we can call on an object.
- The *dynamic type* of a value is the actual type of the object. The dynamic type determines what implementation of a method we actually invoke.

For example, for the above code, here are the static and dynamic types for the various combinations of variable declarations and assignment that our subtyping relationships allow:

```
Animal a1 = new Dog();    // static type: Animal, dynamic type: Dog
Animal a2 = new Cat();    // static type: Animal, dynamic type: Cat
Animal a3 = new Lion();   // static type: Animal, dynamic type: Lion
Cat c1  = new Cat();      // static type: Cat, dynamic type: Cat
Cat c2 = new Lion();      // static type: Cat, dynamic type: Lion
Lion l = new Lion();      // static type: Lion, dynamic type: Lion
Dog d = new Dog();        // static type: Dog, dynamic type: Dog
```

We can use this information to determine which of the method implementations actually fire for any call to `makeNoise`.

14.6 The Mechanics of Dynamic Dispatch

Recall the rules for resolving overridden method calls:

1. We use the *static type* of an expression to determine whether the method call typechecks.
2. We use the *dynamic type* of an expression to determine which method we actually invoke.

Note that the static type is usually a more *general type* (the supertype) than what we have and the dynamic type is the more specific type (the subtype).

This rule helps us resolve method calls. However, this resolution process becomes tricky with trickier set ups of inheritance hierarchies. To better understand the process of resolving overridden method calls, we'll study how this is implemented in Java with our mental model of computation.

14.6.1 Objects on the Heap

Recall that all objects are allocated on the heap. An object is a collection of fields along with a *tag* that says what type the object is, its dynamic type. More concretely, consider the following basic class hierarchy:

```
public class C1 {
    public int x;
    public void foo() { System.out.println("C1.foo()"); }
}

public class C2 extends C1 {
    public int y;
    @Override
    public void foo() { System.out.println("C2.foo()"); }
    public void bar() { System.out.println("C2.bar()"); }
}

public class C3 extends C1 {
    public int z;
    @Override
    public void foo() { System.out.println("C3.foo()"); }
    public void baz() { System.out.println("C3.baz()"); }
}

public class C4 extends C3 {
    @Override
    public void foo() { System.out.println("C4.foo()"); }
}
```

Instances of each of these classes look like this on the heap:

```
-------   -------   -------   -------
| C1  |   | C2  |   | C3  |   | C4  |
|x| 0 |   |x| 0 |   |x| 0 |   |x| 0 |
-------   |y| 0 |   |z| 0 |   |z| 0 |
          -------   -------   -------
```

The process of resolving overridden methods is called *dynamic dispatch*. How does the dynamic dispatch process work? The key insight is that the type tag associated with each object is in actually a *pointer* to a table containing the methods for that class. For example, for `C3` we have:

```
-------
| C4  |------> [vtable for C4]
|x| 0 |
|z| 0 |
-------
```

This table of methods is called a *virtual table*, or vtable, owing from its C++ roots where overridable methods are annotated with the `virtual` keyword. The vtable for a class contains an entry for every overridden method in the class as well as a pointer to the superclass's vtable:

```
--------------           --------------           --------------
| C4 vtable  |     ----> | C3 vtable  |     ----> | C1 vtable  |
--------------     |     --------------     |     --------------
|super vtable| ----|     |super vtable| ----|     |super vtable|
|  C4.foo()  |           |  C3.foo()  |           |  C1.foo()  |
--------------           |  C3.baz()  |           --------------
                         --------------
```

Suppose that we have a variable declaration and method invocation:

```
C1 c = new C4();
c.foo();
```

How does dynamic dispatch proceed? We first note that this code *typechecks and compiles* because `C1`, its static type, declares a `foo()` method. Then, at runtime, we:

1. Follow the vtable pointer of the object we invoke the method.
2. Lookup the desired method in the vtable.
3. If the method exists, we invoke that method. Otherwise, we follow the superclass vtable pointer and repeat the process.

For the above code, this amounts to looking at the vtable pointer for the `C4` object. In the corresponding table, we find an entry for `C4.foo()` so we invoke this version of `foo()`. In contrast, consider this method invocation instead:

```
C1 c = new C4();
c.baz();
```

We follow the vtable pointer for the C4 object and look up an entry for `baz()`. We do not find such an entry, so we follow the superclass vtable pointer to C3's vtable. We then look for a `baz()` entry, find such an entry, `C3.baz()`, and invoke that method.

As an exercise it's a good to consider all of the possible combinations of variable assignments we can have with our subtyping relationship and invocation of methods on those variables. Those combinations should either result in a *type error* at compile time or the invocation of one of the versions of these methods. Try writing out these various combinations and use this mental model of dynamic dispatch to predict the result.

14.6.2 The Object Class

While C2, C3, and C4 above have superclasses, it appears that C1 does not. However, this turns out to be untrue; C1 has an implicit superclass called `Object`. `Object` is the *root* of the class hierarchy in Java—all objects are ultimately subclasses of the Object class. Because of this, we say that Java has a *unified class hierarchy* where all classes have a common base class.

The `Object` class provides a number of methods available to all objects in Java. These include the familiar `toString()` method that returns a String representation for an object. All objects in Java have this method, and it is used to implicitly convert an object to a `String`, *e.g.*, when printing it with `System.out.println`. Another important method is the `equals()` method. Here is the signature for equals along with the implementation provided by Java:

```
/** @return true if this object is "equal" to the other object */
public boolean equals(Object obj) {
    return this == obj;
}
```

By default, `equals` uses *reference equality* which checks to see if the argument is the exact same object as the one we invoke equals on. If we want to determine equality between objects that we define, we ought to override equals behavior to be appropriate for the objects in question, typically *structural equality* where we check to see if the fields of two objects are equal recursively. To see this in action, consider adding an equals method for our good old Point class

```
public class Point {
    private int x;
    private int y;
    // ...
}
```

Intuitively, how should we implement equality between two points? The interpretation of structural equality for points says two points are equal if and only if their components are equal.

This is easy enough to state, but we immediately run into problems implementing this behavior in our `equals` method:

```
// in class Point
@Override
public boolean equals(Object obj) { /* ... */ }
```

The problem is that the type of the argument is `Object` so that if we tried to access the `x` field of the argument, we get a type error:

```
public boolean equals(Object obj) {
    boolean isXEq = this.x = obj.x;   // type error: Object doesn't have an x field
}
```

This is because we may call `equals` with an argument that is not a `Point`—any class is a subclass of `Object` after all.

So as a first step, we must check to see if the argument is actually a `Point` object. To do this, we use the `instanceof` operator:

```
public boolean equals(Object obj) {
    if (obj instanceof Point) {
        // compare fields
    } else {
        return false;
    }
}
```

Think of `instanceof` as a binary operator that takes an object on the left-hand side and a class name on the right-hand side. `instanceof` evaluates to `true` if the object on the left-hand side is an object that is a *subtype* of the right-hand side.

When the input is not a `Point`, we can immediately return `false`. When the input is a `Point`, we know that `obj` is really a `Point` even though its static type is `Object`. We need a way to tell the compiler "hey, I know this thing is really a `Point`—trust me". The mechanism we use to do this is a *class cast*.

```
public boolean equals(Object obj) {
    if (obj instanceof Point) {
        Point p = (Point) obj;
        return this.x == p.x && this.y == p.y;
    } else {
        return false;
    }
}
```

A class cast looks like a normal cast except that it casts objects between types. This particular kind of cast is called a *downcast* because we are casting from a more general type—`Object`—to a more specific type—`Point`. In other words, we are casting *down* the class hierarchy. The class cast makes a runtime check to ensure that expression evaluates to an object value that really is the one promised by the cast, throwing a jiClassCastException if this is not the case. However, we know by virtue of the `instanceof` check that we indeed have a `Point` object so this cast is safe.

In general, our overridden `equals` methods have the following form:

```
public boolean equals(Object obj) {
    // Check to see if input has the appropriate object type
    // Downcast to the object type
    // Perform a structural equality check on this object and the input
}
```

14.7 Is-a Versus Has-a Relationships

When designing a program in an object-oriented programming language, we decompose the program into a collection of objects. In our programs, objects are related in two primary ways:

1. *Has-a* relationships, *e.g.*, an employee has an age.
2. *Is-a* relationships, *e.g.*, an engineer is an employee.

We realize the first kind of relationship with *fields*. For example, an `Employee` class would have an `age` field. We realize the second kind of relationship with *interface implementation* or *class inheritance*. For example, we may have either an `Employee` interface or class and an `Engineer` class that implements or extends `Employee`, respectively.

How do we choose between the two sorts of relationships? In some cases, the choice is obvious just by considering the words "has-a" and "is-a". In the above examples, it seems very wrong to consider an employee an "age" or say that an engineer has an employee. In this sense, *the natural meaning of the objects* defines the relationship between objects.

However, we can concoct scenarios where we could interpret the potential relationship between objects in multiple ways. For example, we might say:

- An engineer *is an* employee.
- An employee *has a* title that is "engineer".

Both are perfectly reasonable interpretations of the relationship between "engineer" and "employee". We can either represent an engineer as a class that inherits from/implements the employee class/interface, or a "title" field of the employee class. But which of them do we choose?

When our *semantic* interpretation of our objects' relationships is ambiguous, we must resort to reasoning about the *operational* implications of our design choice. In other words, if we recognize the relationship as a field (*has-a*) or a subtype (*is-a*), what is the effect on the rest of our program? What are we able to do with one programming construct that we cannot do with the other? What difficulties do we encounter using one construct over the other?

Recall that the primary feature that both interface implementation and class inheritance provide is subtyping. That is, we can substitute a subtype (a class that implements an interface or inherits from a superclass) anywhere that a supertype is expected. This gives us the ability to write (subtype) polymorphic code—code that operates over many types. Note that this feature has to do with the (Java) type of an object. So if a disputed property could function as part of the type of an object, then this sort of question between *is-a* and *has-a* arises.

For example, we may decide that all employees receive a salary. Therefore, we can enforce this by specifying a `getMonthlySalary()` method in the Employee class. When we need to gather up the total amount of money we need to pay out to the company, we can simply call the method uniformly on each Employee:

```java
List<Employee> employees = /* ... */ ;
int total = 0;
for (Employee e : employees) {
    total += e.getMonthlySalary();
}
System.out.println("The total is: " + total);
```

Implicitly throughout this code, the subtyping relationship between `Employee` and its subclasses is enforced at compile time via typechecking. In contrast, if we had a field of type `String` denoting the title of the employee, we would need to write the `getMonthlySalary()` method of the Employee class as follows:

```
public int getMonthlySalary() {
    if (this.title.equals("Engineer")) {
        return 10000;
    } else if (this.title.equals("CEO")) {
        return 50000;
    } else if (/* other cases */) {
      // ...
    } else  {
        return /* Some default amount... */
    }
}
```

Note that this method is very brittle. Because the case analysis is over Strings, we cannot ensure at compile time that:

1. We provide the names of the titles correctly, *i.e.*, there are no typos, and
2. We have provided cases for *all* possible titles.

In addition to substitution via subtyping, we also gain *static typechecking* by using inheritance to model this relationship. Thus if we need to take advantage of subtype polymorphism, we ought to do so via interfaces or inheritance.

However, otherwise if there's no compelling reason to use subtype polymorphism, we ought to stick with *has-a* relationships and fields, also known as *composition*. Composition has the benefit of being:

1. Simpler. We spent little time talking about the mechanics of fields because they are straightforward to understand relative to inheritance.
2. Flexible. Exploiting inheritance requires that we create classes which must be built at compile time unless we use a complicated mechanism like reflection to load classes dynamically. A field on the other hand can take easily take any value that it domain allows.

Imagine in our employee example that we did not know the full set of job titles ahead of time, *e.g.*, because we're working on a general-purpose employee management system not tied to any one corporate structure. Here, we are unlikely to be able to take advantage of typing checking because the set of possible jobs is not fixed at compile time. In this situation, it may make sense to use composition rather inheritance to represent these positions.

There is a modern maxim that you will likely hear:

Favor composition over inheritance.

This is sound advice at a surface level. Composition is less complicated than inheritance, so you should favor the simpler thing over the more complicated thing. However, as we have seen, there are specific circumstances when you want inheritance over composition: when the relationship involved can be construed as "type-defining"and the benefits of inheritance—subtype polymorphism and static type checking—make sense in context.

14.8 The Expression Problem

When we have identified a relationship between objects as being “type-like”, this raises questions of whether we identify the relationship as *is-a* or *has-a*. One particular situation where this arises is when we need to specify behavior that performs *case analysis* on a set of types. In this situation, using an *is-a* relationship via interface implementation or inheritance shines because we can really put the type system to work for us. Although sometimes, exploiting this relationship is less intuitive than you might imagine.

For example, consider the employee example that we’ve used throughout our discussion of inheritance. Suppose that we did not use inheritance and, instead, defined the type of employee by their job title, a field. And now suppose that we wanted to define a method to retrieve the salary of the employee based on their job title. The method would look like this:

```java
public int getSalary(Employee e) {
    if (employee.getTitle().equals("engineer")) {
        return 100000;
    } else if (employee.getTitle().equals("chef")) {
        return 75000;
    } else if (employee.getTitle().equals("ceo")) {
        return 1000000;
    } else {
        // A default salary
        return 0;
    }
}
```

This method is unsatisfying for a pair of reasons:

1. By encoding the title as a string, we may have some flexibility in specifying jobs at runtime that we didn’t anticipate at compile time. However because of this, we lose the ability to check that we have specified the correct job title at compile time. For example, we may do something silly like misspell “engineer”.
2. Related, in addition to being unable to check for simple typos like this at compile time, we are also unable to check for *exhaustiveness*—did we cover all the possible cases of employees?

There is also a third reason why this approach is unsatisfying. In addition to typos and exhaustiveness, any errors that we introduce with this approach occur at runtime. Furthermore, the errors are *silent*. For example if we misspell or forget “engineer”, then the method return 0 which will not manifest itself as an error until we do some calculation involving the engineer’s salary. This is nearly a worst-case scenario for us! We want errors checked at compile time at the point where they occur, not checked at runtime potentially far away from their origin.

We could add inheritance into the mix. However, inheritance alone is not enough, for example, if we introduced `Engineer`, `Chef`, and `CEO` classes that extend `Employee`, we might try the following:

```java
public int getSalary(Employee e) {
    if (employee instanceof Engineer) {
        return 100000;
    } else if (employee instanceof Chef) {
        return 75000;
    } else if (employee instanceof Ceo) {
        return 1000000;
    } else {
        // A default salary
        return 0;
    }
}
```

However, this code is only marginally better than the previous approach. The fact that the different jobs are encoded as types means that the compile will catch typos. However, exhaustiveness is not checked and errors are still silent.

To cover all of these bases, we must combine inheritance with dynamic dispatch. Thus, we arrive at the initial example we used to motivate dynamic dispatch:

```java
public abstract class Employee {
    public abstract int getSalary();
}

public class Engineer extends Employee {
    @Override
    public int getSalary() { return 100000; }
}

public class Chef extends Employee {
    @Override
    public int getSalary() { return 75000; }
}

public class Ceo extends Employee {
    @Override
    public int getSalary() { return 1000000; }
}
```

This final approach allows us to check for typos and exhaustiveness at compile time. Note that exhaustiveness is checked because the abstract class requires that any subclass implement the `getSalary()` method. If we need to implement other operations that also perform case analysis on the possible job types, we:

1. Introduce the method in the superclass. If default behavior is required, we give a default implementation. Otherwise, we mark it abstract so that implementors are forced to override the method.
2. Override the method in each subclass with the case-specific behavior.

This sort of design—defining data by cases and then operations over that data via case analysis—is common place in computer programs. To summarize, the key features of good design in this space is:

- Extensibility of new cases.
- Extensibility of new operations.
- Statically (type) checked.

- *Separate compilation*, the ability to write new code without having to recompile existing code.

Note that the approach that we have settled on does not cover all of these desiderata. While we have static typechecking, the ability to add new cases via additional subclasses, and separate compilation when adding new subclasses, adding new operations is a bit of a bother. Every new operation we add becomes a new method in the superclass and a series of overridden methods in the subclasses. The addition of this new method requires that we recompile all of the subclasses since they have been modified to accommodate this new method. Furthermore, the logic for this operation is spread out among all the subclasses which is problematic for code understanding.

From a programming language design perspective, this problem is known as the *expression problem*. It is difficult to satisfy all four of these desiderata simultaneously and different languages (in particular, *language paradigms* such as object-oriented versus functional programming) make trade-offs in various dimensions when solving this problem. In Java, as long as we leverage inheritance and dynamic dispatch, we can get close, although you should be aware that the approach is still unsatisfying in the ways we have discussed.

14.9 Case Study: Exceptions

Previously, we discussed how to *raise exceptions* using the throw statement:

```java
throw new IllegalArgumentException();
```

The subject of the `throw` statement is an expression that evaluates to an exception object. This led to the question of whether we could define our own exceptions. Indeed we can via subclassing; exceptions turn out to be a great—if not controversial—example of inheritance in Java.

14.9.1 Defining Exceptions

In Java, we define our own exception by writing a subclass for the `Throwable` class or one of its subclasses. The `Throwable` class hierarchy looks like this:

```
                Throwable
                    |
            -----------------
            |               |
        Exception         Error
            |               |
    -----------------      ...
    |               |      (3)
   ...       RuntimeException
   (1)              |
                   ...
                   (2)
```

You can create subclasses for your exception at any of the three points indicated above. Each point represents one of three possible types of exceptions in Java:

1. *Checked exceptions* are subclasses of `Exception` (or its subclasses—technically any subclass of `Throwable` that is not also a subclass of `Error` or `RuntimeException`). Checked exceptions are exceptional conditions that a program ought to anticipate and recover from. For example, the `Scanner` class's constructor throws `FileNotFoundException` if the given file does not exist on disk. This is an exceptional condition, but one that is easily recoverable from—inform the user and exit or ask the user for another file.
2. *Unchecked exceptions* or runtime exceptions are subclasses of `RuntimeException`. These exceptions arise from conditions internal to the application that are not recoverable from. An example of an unchecked exception is the `IllegalArgumentException` that we've thrown when a pre-condition on a method is violated. In such a scenario, we don't want to try to recover from this error as the condition should never occur.
3. *Errors* are subclass of the `Error` class. They are similar to unchecked exceptions in that they are situations in which the program cannot recover, but the conditions in which they occur are external to the application, rather than internal. For example, if we run out of memory—a situation that is not a problem with our program, *per se*, and is not recoverable—Java will throw a `VirtualMachineError`.

In many cases, the exceptions provided by the Java standard library are sufficient. For example, the descendants of `RuntimeException` such as `IllegalArgumentException`, `NullPointerException`, and `IllegalStateException`, cover a lot of the common cases of pre-condition and invariant violation that we might encounter in our code. However, if we would like to provide a more specific exception, *e.g.*, a `BankAccountNegativeException` for when the invariant that our bank account balance goes negative, we can create a subclass of the appropriate type.

14.9.2 Handling Exceptions

Checked exceptions require that we recover from them in some way. Java provides two mechanisms for this:

1. The `throws` clause on methods indicating that a method generates an exception and that any caller of this method must handle it. For example, when we created a new `Scanner` with a File, we had to add `throws FileNotFoundException` to indicate that the method that contained this code could throw this exception.
2. Try-catch statements which allow us to *try* to execute some code that may throw an exception and then *catch* any exceptions that arise from them.

In Java, all *checked exceptions* must be dealt handled in one of these two ways—throws clauses or try-catch. While you can use these mechanisms to document and catch runtime exceptions and errors, it is strongly advised not to do so—these exceptions occur specifically because you cannot recover from them.

The try-catch has the following form, *e.g.*, to properly create a `Scanner` from a `File`:

```java
import java.io.FileNotFoundException;
// ...
Scanner in = null;
File file = new File("foo.txt");
if (file.exists() && !file.isDirectory()) {
    try {
        in = new Scanner(new File("foo.txt"));
    } catch (FileNotFoundException e) {
        System.out.println("Error: file not found " + e.getMessage());
    }
    if (in != null) {
        // Use the scanner...
    }
}
```

Here, we try to create a scanner that reads from `foo.txt`. If no exception is thrown, then `in` is loaded with an appropriate `Scanner` object. If the line `in = new Scanner(new File("foo.txt"))` throws a `FileNotFoundException`, then control flows to the *exception handler* for the `FileNotFoundException`, *i.e.*, the block of code that prints the error message. In either case, control flows to the line after the try-catch block—the if-statement. We check to see if `in` is `null` after the try-catch block because if `FileNotFoundException` is thrown, then `in` will never be loaded with a `Scanner` object.

Note that even though we check to see if the file exists, we still need to use a try-catch to catch the potential `FileNotFoundException`. This is because `FileNotFoundException` is a checked exception, but it is marked as such because even though we have done the proper check, the file might be deleted from disk or made unavailable after the check executes! Also note that we cannot declare the `Scanner` variable inside the try-catch block. This is because of a technicality with scoping. If we wrote this code:

```
import java.io.FileNotFoundException;
// ...
if (file.exists() && !file.isDirectory()) {
    try {
        Scanner in = null;
        File file = new File("foo.txt");
        in = new Scanner(new File("foo.txt"));
    } catch (FileNotFoundException e) {
        System.out.println("Error: file not found " + e.getMessage());
    }
    // Error: `in` not in scope!
    if (in != null) {
        // Use the scanner...
    }
}
```

We receive a compiler error stating that the occurrence of `in` in the conditional is not in scope. This is because the lifetime of a local variable is the curly braces that enclose it. Therefore, the lifetime of `in` in this case is the try-block. We want `in` to exist outside of the try-block, so we have to declare it outside of the try-block.

We can catch any number of exceptions in a try-catch block by nesting branches, *e.g.*:

```
try {
    // Code that throws either FileNotFoundException
    // or IndexOutOfBoundsException...
} catch (FileNotFoundException e) {
    // If FileNotFoundException is thrown...
} catch (IndexOutOfBoundsException e) {
    // If IndexOutOfBoundsException is thrown...
}
```

Furthermore, a catch block catches a specified exception as well as any of its subtypes. `FileNotFoundException` is a subtype of `IOException`, so we could use `IOException` in its place if we also anticipated other subtypes of `IOExceptions` to be thrown or an `IOException` itself (usually reserved for general IO problems):

```
try {
    // Code that throws any IOException...
} catch (IOException e) {
    // Catches any IOException or a subclass of IOException.
}
```

Because of subtyping, we could theoretically catch any exception—checked or unchecked—by catching `Exception`. This is typically used as a "catch-all" catch case:

```
try {
    // Code that can throw an exception
} catch (Exception e) {
    // Handles any (checked) exception...
}
```

However, this is usually undesirable because if you are required to handle a particular (checked) exception, you will likely want to handle it in its own specific way, *e.g.*, issuing an error to the user. Using this "catch-all" style does not allows this. Worse yet, is combining "catch-all" with *swallowing the exception*:

```
try {
    // Code that can throw an exception
} catch (Exception e) {
    // Do nothing if we receive an exception
}
```

This silently handles any checked exception that is raised by the code in the try-block. While convenient, again, you will likely need to have some recovery logic for each specified checked exception that is raised. At the very least, you will want to avoid swallowing the unchecked exceptions that this code could throw as these represent unrecoverable errors.