Generality

Contents

This chapter is about features and conventions of Java which allow classes to be made more general and re-usable:

- generic classes
- inheritance with `extends`
- inheritance with `implements`
- delegation, overloading, statics

Generic types

Most programmers understand one kind of generic type very well, and that is arrays

```
Counter[] chart;
String[] names;
Database[] dbs;
```

Switching from arrays to lists gives:

```
List<Counter> chart;
List<String> names;
List<Database> dbs;
```

Using generics

To use generic library classes:

```
List<Item> xs = new ArrayList<Item>();
List<Integer> ys = new ArrayList<Integer>();
List<Integer> zs = new ArrayList<>();
```

`xs` is a list of objects of type `Item`; you choose which implementation of lists at creation time

`ys` works like a list of `int`, using a wrapper class, with and boxing/unboxing

`zs` illustrates the diamond abbreviation

(`ArrayList` is a choice of implementation of `List`)

Arrays of generics

Generics are recent (Java 5) so there are restrictions, e.g. to make an array of lists, this doesn't work:

```
List<Item>[] xs = new List<>[10];                    bad
```

But this does:

```
List<Item>[] xs = newArray(10);
...
@SuppressWarnings("unchecked")
List<Item>[] newArray(int n) {
    return new List[n];
}
```

Note a list of arrays `List<Item[]>` works fine

Your own generic classes

Generic classes can be useful if not overused, but they are very good for reducing dependencies

In the maze program, a state class is needed which doesn't depend on the entity class:

State.java

```
class State<Entity> {
    private Entity entity;
    private int counter;
    ...
}
```

You could use `C` in place of `Entity` - it is a parameter which stands for any class

Unit testing

The unit testing for the state class is:

State.java

```
class State<Entity> {
    ...
    public static void main(String[] args) {
        State<String> s = new State<>();
        ...
    }
}
```

Here, the parameter `Entity` is replaced by `String` for testing (strings are used as 'mock entities')

Makefile

Here is a very good Makefile technique for projects:

Makefile

```
State = State.java
Direction = Direction.java
Grid = Grid.java Direction.java
...

%: %.java
        javac $($@)
        java -ea $@
```

Each line lists the source files for compiling a class

Note `make Grid` only works if `Grid.java` exists

Using generics

Elsewhere in the maze program, there is this:

```
State<Entity> s = new State<>();
```

This time, `Entity` is the real entity class, being passed as an 'argument' to the `State` class

The result is a 'state of entities' as we wanted

The grid class

The maze program needs a generic grid class:

Grid.java

```
class Grid<Entity> {
    private Entity[][] cells;

    @SuppressWarnings("unchecked")
    Grid(int w, int h) {
        cells = (Entity[][]) new Object[w][h];
    }
    ...
}
```

It mainly holds a 2D array of entities, again `Entity` is a parameter, but there is a problem with initialization

Arrays of parameter type:

Arrays of a parameter type E have a problem, similar to arrays of a generic type, e.g. this doesn't work:

```
private E[] xs = new E[10];                    bad
```

But this does:

```
private E[] xs = newArray(10);
...
@SuppressWarnings("unchecked")
private E[] newArray(int n) {
    return (E[]) new Object[n];
}
```

Direct inheritance

Direct inheritance means creating subclasses using the `extends` keyword

The idea is to take a class which ***nearly*** does what you want, and define a new class from it which adapts it so that it does exactly what you want

The class that you adapt is called the *parent* (or *base*) class

Example: Person

Start with a class for holding some personal details:

Person.java

```
class Person {
    final String surname, forenames;
    private String username;
    ...
}
```

There may be a need to store different information about different types of people

This class represents what the different types have in common

Contructor

The `Person` class has a constructor:

Person.java

```
...
Person(String surname0, String forenames0) {
    surname = surname0;
    forenames = forenames0;
}
...
```

Overloading

The code contains both a ***field*** called `username`, and two ***methods*** (getter/setter) called `username`:

Person.java

```
private String username;
...
String username() { return username; }
...
void username(String u) { username = u; }
```

Java can sort this out by context, given that the two methods have different numbers of arguments

Java can handle methods with the same number of arguments, if the types are different

A subclass

Here's a subclass of `Person`:

```
class Student extends Person {
    private String programme;
    ...
```

Student.java

A subclass ***implicitly*** has the same fields and methods as the parent class

It can add extra fields, add extra methods, and *override* old methods, i.e. redefine them

Constructor

The `Student` class needs a constructor:

Student.java

```
...
Student(String surname0, String forenames0) {
    super(surname0, forenames0);
}
...
```

Without it, the default constructor for `Student` would rely on the default constructor for `Person` which ***doesn't exist***

The keyword `super` refers to the contructor in the parent class

Override

The `Student` overrides the `username` method:

Student.java

```
...
@Override
void username(String u) {
    ...
    super.username(u);
}
...
```

The `@Override` annotation is optional (it guards against mis-spellings or type differences)

`super.username` refers to the `username` method in the parent class

Static

Constructors are effectively static, and statics are:

- global, so should be avoided where possible
- not included in the inheritance rules, therefore not properly object oriented, therefore to be avoided where possible
- essential for constructors, constants, useful for unit testing, and used extensively in the libraries

Extend again

Here's another subclass of `Person`:

```
class Staff extends Person {
    private String job;
    ...
```

Staff.java

You can build up a hierarchy of classes as complicated as you like, with child, grandchild, great-grandchild classes and so on

Runtime types

Inheritance gives a partial system of runtime types

Suppose you have a variety of shapes:

```
class Shape {
    void draw() { ... }
    ...
```

```
class Circle extends Shape {
    void draw() { ... }
```

```
class Square extends Shape {
    void draw() { ... }
```

Protected

The `Shape` fields need the `protected` keyword:

Shape.java

```
class Shape {
    protected int x, y;
    ...
```

It is not appropriate to make `x, y` accessible and `final` because they are going to change

If they are `private`, they can't be mentioned in subclasses; `protected` means visible to subclasses

They are also visible to all classes in the current folder, a serious Java weakness, so you might prefer getters

Ask objects to do things

An array of type `Shape[]` can contain shapes from any subclass of `Shape`

Shapes.java

```
Shape[] shapes = ...;
shapes[0] = new Circle(...);
...
for (Shape s : shapes) s.draw();
```

In the call `s.draw()`, Java checks at runtime what actual class the object in `s` has, and calls the right `draw`

This is *dispatch* and only applies to the `s`, not to method arguments

Avoiding inheritance

When OO was young, inheritance was used too much, and beginners also tend to use it too much

Current thinking is that inheritance (a) often creates classes which are too tightly coupled and (b) causes severe problems when used inappropriately

Modern advice is only to use it for reasonably stable classes, and to ask whether "an X is a Y" makes sense, e.g. "a Circle is a Shape"

Bad inheritance

In the libraries, a `Stack` is a `List` (but a stack is not a list, it is ***like*** a list) library source code

But that means you might accidentally use non-stack methods like `clear` or `addAll` which break the stack properties

Almost all the list methods are ***now*** overridden to make them 'safe', but need to be reviewed if lists are changed

In other words, `Stack` is not decoupled from `List`

Library source code

To read the source code of library class, type something like "java Stack source code" into Google

Find a link to `docjar.com` or `grepcode.com`

Also, there is complete source code in `src.zip` which comes with the JDK

To save unzipping everything (big!) try:

```
> unzip -f src.zip java/util/Stack.java
```

Delegation

How should stacks have been defined?

A stack should use a list in its implementation

So, a stack should have a private field which is a list to store its items, and it should use list methods to implement its own methods

Then the two classes would be much better decoupled

In general, *delegation* (passing on work to subobjects) is often better than inheritance

Binary search

The Java libraries have (overloaded) methods `Collections.binarySearch` for lists (and `Arrays.binarySearch` for arrays)

These only work if you pass a ***sorted*** list

Fans of "programming by contract" would say that callers have to promise to pass something sorted, as part of the contract of the method

But it is too expensive to check automatically

Sorting

What the libraries ***could*** have done is to define a subclass `SortedList` of `List`

The subclass has ***the same*** fields and methods, but overrides the update methods to enforce an extra ***guarantee*** of sortedness

For example a `sort` method in `List` could return a `SortedList`, and an `insert` method on `SortedList` could fail if not inserting in the right place

Lists

An example of inheritance in the libraries is that `List` forms a parent class (technically it is an interface)

There are two main subclasses `ArrayList` and `LinkedList`

`ArrayList` allows direct indexing, and uses a flexible array

`LinkedList` can be better when there is a high percentage of insertions and deletions in the middle

Using lists


```
List<String> names;
names = new ArrayList<String>();
```

The variable has type `List`, and there is just one place where the object is created, where you decide to use the `ArrayList` class for the implementation

Using `List` means it is guaranteed that your code uses only official `List` methods, not anything special to `ArrayList`

So you can switch to `LinkedList` by making a name change in one place

Streams

Another example of inheritance in the libraries is streams

There is a parent class `InputStream` for byte streams (technically an abstract class)

One of its subclasses is `FilterInputStream` which applies some sort of processing to the data

That in turn has a subclass `BufferedInputStream` for buffering

You might also want to look at the `java.nio` package for a newer approach

Entities

Entities in the maze game seem like an ideal situation to use inheritance; the base class is:

```
class Entity {
    private int x, y;
    ...
    private Grid<Entity> grid;
    private State<Entity> state;
```

An entity has fields pointing to the grid and state

That makes it autonomous, able to carry out its own actions, without the grid and state being passed in as arguments to its methods (that's OO thinking)

Initialization

An entity is created in two goes, first `e = new Entity()` and then `e.set(...)`

```
void set(Grid<Entity> g, ...) { ... }
```

Why not pass the items into the constructor?

Because then, the child classes would have to provide a similar constructor and call `super` (constructors aren't inherited) and we want the child classes to be as simple as possible

Mutate

One of the basic methods is `mutate`:

```
class Entity {
    ...
    private Entity under;
    ...
    // Mutate into a different type of entity.
    void mutate() { ... under ... }
```

When a star is found, it mutates into a space, but the `mutate` method can't create a `Space`, because it would cause a cyclic dependency, so a `Star` is initialized with a `Space` object in its `under` field, and mutation is done by replacing the entity with the entity 'under' it

Move

Another basic method is `move`:

```
class Entity {
    ...
    void move() { ... }
```

Moving is done by swapping the entity with the space next to it, assuming there is always exactly one entity in every grid position

If entities could be on top of each other, a different approach would be needed (perhaps making more use of the `under` field)

Act

The method which provides behaviour is `act`:

```
class Entity {
    ...
    void act() { }
```

An entity is given a chance to act when the level is set up (to initialize the state) or when it is an active entity's turn (the player)

The default is to do nothing, and each child class overrides it to provide specific behaviour

Simple entities

Walls and spaces do nothing, and stars add one to the counter during initialization:

```
class Wall extends Entity { }
```
Wall.java

```
class Space extends Entity { }
```
Space.java

```
class Star extends Entity {
    void act() {
        state().counter(state().counter() + 1);
    }
}
```
Star.java

Delegation

Regarding the state, grid and entity classes as framework classes, and the child entity classes as being defined by a user of the framework, it is worth putting effort in to making the child classes simple:

```
... void act() { counter(counter() + 1); } ...
```

Instead of asking for the state, then calling a state function, a child class could directly ask for a counter to be updated

The entity class would provide counter methods, which would delegate their work to the state object

The Player

The player is more complex:

Player.java

```
class Player extends Entity {
    void act() {
        if (state().entity() == null) {
            state().entity(this);
        }
        else {
            Entity e = next();
            if (e instanceof Star) e.mutate();
            if (! (e instanceof Wall)) move();
        }
    }
}
```

There are ways to improve this

Testing objects

The player uses `instanceof`:

Player.java

```
...
    if (e instanceof Star) ...
...
```

The `instanceof` keyword allows the player to test other entities, without any dependency on their classes

Finishing off

To make a complete working maze game, with a text interface rather than graphics for simplicity, here are the remaining classes and Makefile:

```
class Direction ...
```
Direction.java

```
class Level ...
```
Level.java

```
class Maze ...
```
Maze.java

```
State = State.java ...
```
Makefile

Inheritance problems

We've seen two potential problems with `extends`, tight coupling and mis-use

Another problem is that a class can only extend ***one*** other class

This makes sense if you think of `extends` as adding fields, and you think of the Java implementation as needing to know the offsets of fields in the object

Interfaces

Java provides another form of inheritance which gets round some of the problems

```
interface Shape {
    void draw();
}
```

An *interface* is a different kind of class, one where (a) you can't create any objects, (b) there are no fields (c) there are no method bodies, just signatures

All an interface does is to specify method signatures that its subclasses must have

Implements

The keyword which goes with interfaces is `implements` instead of `extends`

```
class Circle implements Shape {
    public void draw() { ... }
}
```

Interface methods are implicitly `public`, otherwise the effect is the same:

```
Shape[] shapes = ...;
...
for (Shape s : shapes) s.draw();
```

One-method interfaces

If an interface has only one method in it, Java allows it to be used as a mechanism for passing functions as arguments

There will be examples in the graphics chapter

Game design

Long experience with game design has led to a dissatisfaction with using extends inheritance to define entities

Imagine a large number of types of entity with quite a lot of common aspects of behaviour

There is then a large number of classes with a complex inheritance hierarchy which is difficult to manage

Component architectures

An alternative is a component scheme

In the maze case, instead of the classes `Wall`, `Space`, `Star`, `Player`, an entity would contain a function, passed in during initialization, to replace `act` (and mutation might be easier)

In more complex cases, there would be several functions for different aspects of behaviour, each with multiple variants