Book 3

Chapter 4

Subclasses and Inheritance

Inheritance

A feature of OOP that lets one create classes derived from other classes;

A class will inherit behaviors and attributes of superclass – useful when wanting to create several, different type of classes with common features;

Subclass defines its own implementations of methods/fields;

Inheritance hierarchies

A base class can in turn be used as a base class for another derived class

All classes are derived from a Java class named Object – provides features that are basic to all Java classes;

If a class doesn't specifically state what class it is derived from, it is assumed that it is derived from the Object class

Creating subclasses

Include keyword *extends*

To use inheritance properly, note the following:

Subclass inherits members from its base class;

Constructors are not considered members

Visbility descriptions apply

One can override a method by declaring new member with same signature in the subclass;

Protected hides fields and methods from other classes but is available to subclasses

Overriding methods

Subclass declares a method with the same signature as a public method of the base class, subclass version of the method overrides the base class version;

To override, 3 conditions need to be met:

Class must extend the class that defines the method that you want to override;

Can’t override a private method;

Method in subclass must have the same signature as the method in the base class;

Using Final

Creates a constant whose value can’t be changed after having been initialized;

Creates final methods and final classes

Final methods can’t be overridden by subclasses

Final class can’t be used as a base class

All methods within a class are considered to be final as well

Casting Up and Down (p289)

Casting basically deals with conversion of one data type to another datatype

In objects, one type of object (child or parent) to one another

Upcasting

A child object to a parent object

Provides flexibility and allows one to access parent class members

To access child specified members, need to be overridden methods

Syntax for upcasting:

Parent p = new Child();

e.g Ball b = new Baseball();

Baseball extends Ball – Baseball object is treated as though it’s a Ball object;

Reference to object is assigned to variable b;

Automatic casting doesn’t work the other way around – can’t use a Ball object where a Baseball object is called;

e.g. public void toss(Baseball b);

Ball b = new Baseball();

toss(b); - won’t compile

Ball b = new BaseBall();

toss((BaseBall) b) – you can explicitly cast b variable to a Baseball object

Downcasting

Typecasting of parent object to a child object

Compiler checks if it is possible and if not, ClassCastException occurs

Syntax for downcasting:

Child c = (Child) p;

Calling a method defined by a subclass from an object referenced by a variable of the superclass?

Create a variable to subclass and then use an assignment statement to cast the object:

Ball b = new SoftBall();

SoftBall s = (SoftBall)b; // cast the Ball to a SoftBall

s.riseBall();

Java lets you cast the Ball object to a Softball and call riseBall method in same statement

Ball b = new SoftBall();

((SoftBall) b).riseBall();

((SoftBall) b) – returns object referenced by b variable, cast as Softball

Dot operator used to call any other method of the Softball class

Instanceof operator determines object's class type at run time

Determining an object's type

A variable of one type can refer to an object of another type

e.g.

Employee emp = new SalariedEmployee();

emp variable is Employee but refers to object SalariedEmployee

Make use of instanceof operator in order to determine type of object has been assigned to the variable

```
Employee emp = getEmployee();
String msg;
if (emp instanceof SalariedEmployee)
{
msg = "The employee's salary is ";
msg += ((SalariedEmployee) emp).getFormattedSalary();
 }
else { msg = "The employee's hourly rate is ";
msg += ((HourlyEmployee) emp).getFormattedRate();
}
 System.out.println(msg)
```

Instanceof operator is used in an if statement to determine type of object returned by getEmployee method.

Emp variable cast

Polymorphism

Makes use of inheritance

Has many versions of the same method but prioritises w

- Java ability to use base class variables to refer to subclass objects;
- Keep track of which subclass an object belongs to;
- Overridden methods of the subclass

Late binding –waiting until the program is executing to determine exactly which method to call

Creating Custom Exceptions

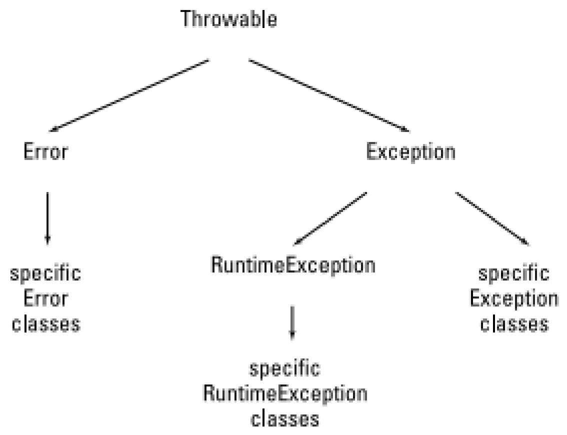
Make use of try/catch exceptions to catch exceptions

Throw statement to throw exceptions

Throwable

Root of exception hierarchy

Represents any object that can be thrown with a throw statement and caught with a catch clause



Error – types of errors that can occur; programs can’t recover from

Exception – Programs should try recover from;

Top of hierarchy of the types of exceptions you catch wuth try/catch statements

RuntimeException – unchecked exceptions; no need to catch or throw unchecked exceptions

e.g. NullPointerException and ArithmeticException

Creating an exception class

Define a class that extends one of the classes in Java exception hierarchy

Extend exception to create a custom checked exception

e.g.

```
public class ProductDataException extends Exception
{
public ProductDataException
{
}
Public ProductDataException(String message)
{
Super(message)
}
}
```

Throwing a custom exception

```
public class ProductDDB {

public static Product getProduct(String code)

throws ProductDataException {

try {

Product p;

//code that gets the product from a file

// and might throw an IOException

p = new Product();

return p;

}

catch (IOException e)

{

throw new ProductDataException( "An IO error occurred.");

}

try {

Product p = ProductDB.getProduct(productCode);

} catch

(ProductDataException e) { System.out.println(e.getMessage());

}

}

}
```