OOP

Object-Oriented Programming

Object-Oriented Programming

Nadeschda Nikitina

University of Oxford

November 2014

Part 1

Course Aims and Objectives

1.1 Course Aims

The techniques taught this week aim to help you design your programs to

- make your code more abstract—applicable to a wide range of inputs
- keep your code as simple as possible in order to reduce error-proneness
- make it easily extendable

1.1 Plan for the Week

- DAY 1: Revision of Java
- DAY 2: Iterators; pros and cons of inheritance
- DAY 3: Generics; typing rules
- DAY 4: Complex use case: binary search trees
- DAY 5: Immutable objects

Class schedule: 9:00-12:20 and 14:00-17:00

1.1 Literature

“Program Development in Java” by Liskov and Guttag is a course text book. Further reading:

- Best practices for using Java:

  “Effective Java” by Joshua Bloch. 2nd edition.

- More details on Generics:

  “Java Generics and Collections” by Maurice Naftalin and Philip Wadler.

- More on design:

  “Design patterns : elements of reusable object-oriented software” by Erich Gamma and Richard Helm.

- Latest Java features:

  “ Java SE8 for the Really Impatient” by Cay S. Horstmann.

Part 2

Java Basics

2.0 Outline

2.1 Types in Java

Java is a strictly, statically typed language.

All variable declarations must contain type information:

```
int i = 10, j = 2;
String s = "Hello";
```

2.1 Types in Java

There are two basic categories of types in Java:

- **Primitive types** represent the eight fundamental types of values in Java: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. Their elements are simple values, such as numbers, and *not* objects. The value of a variable of a primitive type is directly the numeric value.
- **Reference types** include everything else, and their elements are objects of different kinds: defined by the user, defined in the standard library, or arrays. Variables of reference types are "pointers" to objects, i.e., their value is a reference to a place on the heap where the object's data is stored.

Java provides wrapper classes for primitive types (`Boolean` for `boolean`, `Integer` for `int` etc).

2.1 The Null Value

- The expression `null` is an element of any reference type. It means that the variable does not reference any object at the moment.
- An attempt to access an object when the value of a variable is `null` generates a `NullPointerException` at runtime.
  For example:

```
String s;
if (s.length() != 0) { ... } // NullPointerException
s = "";
if (s.length() != 0) { ... } // fine now
```

2.2 Defining Classes

- Classes are the fundamental building blocks of Java.
- Class declarations consist of a class name, e.g., `Secretary`, followed by a block, contained between { and }, containing the class definition:

```
class Secretary {
  String name;

  int getSalary () { return 25000; }
  String getName () { return name; }
}
```

- A class can contain *attributes* and *methods* (collectively called *features*), initialisation code, and other classes.

2.2 Method Declarations

Methods always have specified return values, and a (possibly empty) list of typed parameters.

```
class Secretary {
  boolean broadcast(String message) {
    int number;
    ...
  }
}
```

Local variables, such as `number`, exist only within the scope of the method. If declared within a code block, e.g., loops or conditional statements, they exist only within that block.

Objects created within methods may persist after return.

Local variables *must* be explicitly initialised before use.

2.3 Constructors

Class declarations can contain a special kind of method called *constructor*—method used to set up a new instance of the class. It is a method with a same name as the class:

```
class Secretary {
  ...

  Secretary(String s) {
    name = s;
  }
}
```

Constructors do not have a return type; by definition, they return an object of that class.

If a class does not have any explicit constructors, the constructor with the empty argument list is provided implicitly (*default constructor*). It disappears when other constructors are declared.

2.3 Instances of Classes

- Once we have declared a `Secretary` class, we can create a `Secretary` object using the `new` keyword.
- Many individual *instances* of a class can exist simultaneously at runtime, and each can hold different concrete values for the attributes:

```
Secretary sec = new Secretary("Jane");
...
Secretary anotherSec = new Secretary("James");
```

- The attributes and methods of objects can be accessed using the dot-operator (.):

```
sec.name = "Jake";
sec.getName();
```

2.3 Shadowing of Variables

Normally the reference to the current object is implicit.

If an instance variable and a local variable have the same name, the local variable “shadows” or hides the instance variable within the scope of the method:

```
class Secretary {
  String name;
  public Secretary(String name) {
    name = name;                   // does not work!
    ...
  }
}
```

Both `name` references in the constructor refer to the constructor argument.

2.3 Reference This

We could refer explicitly to the member field `name` by using the special reference `this`:

```
class Secretary {
  String name;
  public Secretary(String name) {
    this.name = name;
    ...
  }
}
```

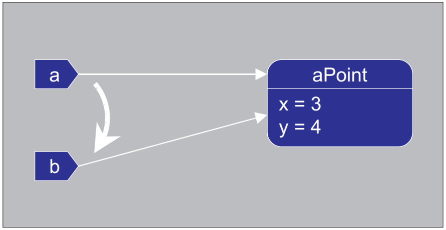
The `this` reference always refers to the current object.

Shadowing of local variables by other local variables is not allowed.

2.3 Assignments

When = is used, the value of the variable on the right-hand side is copied to the variable on the left-hand side.

- For primitives, the actual value is copied and exists twice in memory.
- For reference types, it makes both variables refer to the same address on the heap (only the identity of the object is copied):



Now, the assignment `a.x` = 5 sets `b.x` to 5 as well.

2.3 What does `a == b` mean?

- Primitive types: true if the value of the variable is the same

```
int a=12;
int b=12;
//prints true
System.out.println(a==b);
```

- Reference types: true if the value of the reference (”pointer”) is the same

```
Integer k=new Integer(12);
Integer m=new Integer(12);
//prints false
System.out.println(k==m);
```

2.3 Argument Passing

All arguments to methods are *passed by value*—the (address/numeric) value of arguments is copied.

Assume that we have the following method:

```
 void updateAge(Person p, int newAge) {
    p.setAge(newAge);
// test: try to change the argument
    newAge = 0;
  }
```

What does this program print?

```
  Person joe = new Person("Joe", 35);
  int newAge = 36;
  updateAge(joe, newAge);
  System.out.println(newAge);
  System.out.println(joe.getAge());
```

2.3 Argument Passing

All arguments to methods are *passed by value*—the (address/numeric) value of arguments is copied.

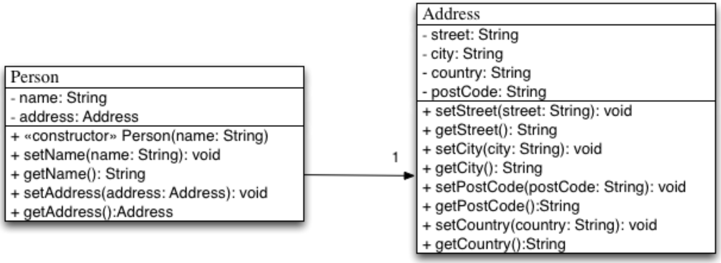
```
Person joe = new Person("Joe", 35);
int newAge = 36;
updateAge(joe, newAge);
System.out.println(newAge);
// still prints 36
System.out.println(joe.getAge());
// prints the new age 36
```

The direct value of reference type variables is a reference to a place on the heap. When a method is called, only the direct value is copied over into another variable, not the object's data. As a consequences, we can modify the state of the object.

The direct value of primitive type variables is the numeric value, so the actual numeric value is copied over into another variable.

2.3 Pen & Paper Exercise

Sketch an implementation of the class Person on paper assuming that the class Address already exists:


Person
- name: String
- address: Address
+ «constructor» Person(name: String)
+ setName(name: String): void
+ getName(): String
+ setAddress(address: Address): void
+ getAddress():Address
1
Address
- street: String
- city: String
- country: String
- postCode: String
+ setStreet(street: String): void
+ getStreet(): String
+ setCity(city: String): void
+ getCity(): String
+ setPostCode(postCode: String): void
+ getPostCode():String
+ setCountry(country: String): void
+ getCountry():String

2.4 Instance vs. Class Variables

Two kinds of features can be declared in a class:

- instance variables and methods
- static or class variables and methods

Every object has its own instance variables. The static variables are shared among all instances of the class:

```
class Secretary {
  String name;
  int employeeID;
  static int nextID;

  Secretary (String s) {
    name = s;
    employeeID = nextID++;
  }
}
```

2.4 Accessing Static Features

- Instance variables and methods are associated with and accessed through an object. Static variables and methods live in the class and are accessed through the class:

```
System.out.println( ‘‘ Maximum secretary  ID:’’ +
            Secretary. nextID−1);
```

- Inside a class, both instance and static members can be accessed directly by their names.

2.4 Static Members

The most common use of static variables is for declaration of constants. For example:

```
class Math {
  ...
  public static final double PI = 3.14;
  ...
  public static double sqrt (double arg) {...}
}
```

Static methods can only access other static members of the class.

2.4 Method Main

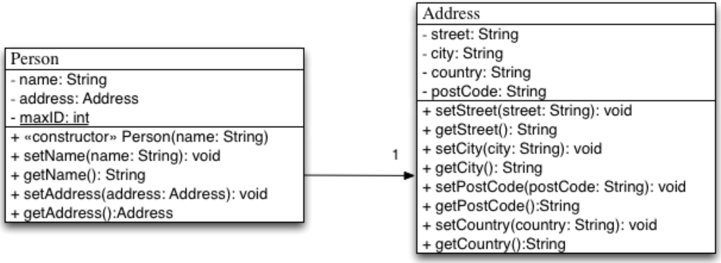
The `main` method is a special static method that can serve as the entry point for a Java application:

```
public class HelloJava {
  ...
  public static void main(String[] args) {
    System.out.println("Hello, Java!");
    ...
  }
}
```

When the program runs, printing “Hello, Java!” will be the first line of code that is executed.

2.4 Pen & Paper Exercise

Extend your implementation of the class Person so that every person will be assigned a unique ID upon creation.



2.5 Arrays

- An *array* is a special type of object that can hold an ordered collection of elements.
- Both following lines are acceptable declarations of an array variable `strings`:

```
String [ ] strings;
String strings [ ];
```

- The type of the elements of the array (e.g., String) is called the *base type* of the array.
- Java supports arrays of all primitive and reference types (exception: Generics).

2.5 Array Creation and Initialisation

- To create the array object, we use the `new` operator and specify the length of the array in brackets:

```
strings = new String[3];
```

- Each array object has one fixed attribute, called `length`.
- Once created, we can access the elements of an array using the index operator `[ ]` and integers between 0 and `strings.length − 1`:

```
System.out.println(strings[0]);
```

2.5 Array Access

- It is important to note that the elements of the above array still only refer to `null`:

```
assert strings != null;
assert strings[0] == null;
```

- The default values for elements of an array are the same as the default values for instance members: zero for numeric values, `false` for booleans, and `null` for reference types.
- The following construct combines declaration and initialisation of an array and its elements:

```
String [] strings = { "first", "second", "third" };
```

Combine declarations with initialisations whenever possible!

2.5 Multidimensional Arrays

- We can use arrays of arrays of some base type to simulate multidimensional arrays in Java.
- They are declared using multiple bracket pairs, one for each dimension:

```
boolean board [] [];
board = new boolean [8] [8];
```

- The elements of this array can be accessed using a pair of brackets. For example:

```
for (int i = 0; i < 8; i++)
  for (int j = 0; j < 8; j++)
    board[i][j] = true;
```

- Arrays may have an arbitrary number of dimensions.

2.5 Pen & Paper Exercise

Arrays do not need to be rectangular. We may also specify only the initial index of a multidimensional array to get an array type with fewer dimensions:

```
int [][] triangle = new int [4][];
```

This involves using several `new` operations.

Exercise: Write code that creates an array that is triangular in shape:

```
0
1 2
2 3 4
3 4 5 6
```

2.5 Pen & Paper Exercise: Answer

```
int [][] triangle = new int [4][];
for (int i = 0; i < triangle.length; i++) {
  triangle [i] = new int [i + 1];
  for (int j = 0; j < i + 1; j++)
    triangle [i][j] = i + j;
}
```

The code above gives:

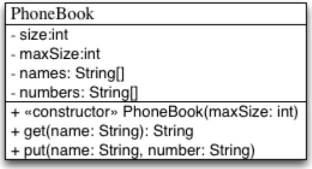
```
0
1 2
2 3 4
3 4 5 6
```

2.5 Practical 1.1

Implement a class `PhoneBook` that can be used to store phone numbers for names:

```
PhoneBook pb = new PhoneBook(2);
pb.put("nadeschda", "6753423");
pb.put("michael", "5436453");
System.out.println(pb.get( "nadeschda" ));
```

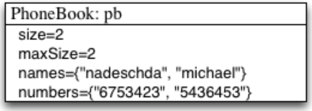
The idea is to store names and numbers in two arrays so that the position of the name and the associated number coincide.


PhoneBook
- size:int
- maxSize:int
- names: String[]
- numbers: String[]
+ «constructor» PhoneBook(maxSize: int)
+ get(name: String): String
+ put(name: String, number: String)

2.5 Practical 1.1

```
PhoneBook pb = new PhoneBook(2);
pb.put("nadeschda", "6753423");
pb.put("michael", "5436453");
```

The above statements change the state of the phone book instance as follows:


PhoneBook: pb
size=2
maxSize=2
names={"nadeschda", "michael"}
numbers={"6753423", "5436453"}

Part 3

Inheritance and Subtyping

3.0 Outline

3.1 Subtyping

- Java supports *subtyping*.
- If a type `B` is a *subtype* of `A`, then it supports all the features of `A`, and possibly more.
- If the type `Mobile` is a subtype of `Phone`, it supports phoning behaviour in addition to, say, MP3-playing behaviour.
- Subtyping is reflexive—a type is a subtype of itself—and transitive:
- If type `Smart` is a subtype of type `Mobile`, and type `Mobile` is in turn a subtype of type `Phone`, then `Smart` is a subtype of type `Phone`. We will use the following notation for brevity:

$$\texttt{Smart} \preccurlyeq \texttt{Mobile} \preccurlyeq \texttt{Phone}$$

3.1 Subtype and Substitutability

- The point of type hierarchies is the ability to apply the same code to a set of types rather than a single type in accordance to the type hierarchy.
- *Substitution principle:* if `B` is a subtype of `A`, it should be possible to substitute instances of `B` for instances of `A` in *any situation* with *no observable effect*.
- An instance of `Phone` can be substituted by an instance of `Mobile` or `Smart`.
- An instance of `Smart` cannot be substituted by an instance of `Mobile` or `Phone`.

3.1 Types in Variable Declarations

Variable declarations specify an interval of admissible types for the referenced object:

```
Phone  o;
```

The above variable can reference objects of type `Phone` or any more specific type: admissible types interval is [`Phone` ... T^{∞})

```
// all assignments are valid:
o = new Phone ();
o = new Mobile();
o = new Smart ();
```

`Object` is the most general reference type in Java. It is a supertype of any reference type.

3.1 Pen & Paper Exercise

The table contains the static variable type in the left-most column and the types of the object assigned to the variable in the remaining columns. Mark all cells that represent a valid assignment.

Variable Type	Object	Phone	Mobile	Smart
Object				
Phone				
Mobile				
Smart				

3.2 Subclassing

One way to establish a subtype relationship in Java is *subclassing*.

- A class in Java can be declared as a subclass of another class using the `extends` keyword. A subclass can be further subclassed, so classes exist in a hierarchy:

```
class Phone {}
class Mobile extends Phone  {}
class Smart  extends Mobile {}
```

- A subclass inherits all members (fields and methods) from its superclass. The subclass may use these members as if they were declared within the subclass itself.
- A class may only inherit from one class; that is, Java allows only single inheritance.
- Classes with the `final` modifier can not be subclassed.

3.3 Back to Running Example

So far, we have defined the following class:

```
class Secretary {
  String name;

  int getSalary () { return 25000; }
  String getName () { return name; }
}
```

3.3 Extending the Example

Let us define a different kind of an employee:

```
class Manager {
  String name;
  Secretary secretary;

  Manager(String n, Secretary s) {
    name = n; secretary = s; }

  String getName() { return name; }
  int getSalary() { return 65000; }
  Secretary getSecretary() { return secretary; }
}
```

3.3 Need for a Common Supertype

Assume that a part of our system, e.g., EmployeeRegister, only uses `getName()` in both `Secretary` and `Manager`. In that case, introducing a superclass including those common features would enable us to make our code more general.
Code without common supertype:

```
Object[] employees = new Object[10];
Secretary s1 = new Secretary("bert jones");
employees[0] = s1;
employees[1] = new Manager("anne smith", s1);
...
// does not compile:
for(int i=0;i<employees.length;i++){
  System.out.println(employees[i].getName());
}
```

We lose the information about s1 being a secretary and the other instance being a manager and cannot access `getName()`.

3.3 Need for a Common Supertype

Second attempt without a common supertype:

```
 Secretary[] secretaries = new Secretary[10];
 Secretary s1 = new Secretary("bert jones");
 secretaries[0] = s1;
 Manager[] managers = new Manager[10];
 managers[0] = new Manager("anne smith", s1);
 ...
 // compiles, but requires two loops:
for(int i=0;i<secretaries.length;i++){
   System.out.println(secretaries[i].getName());
 }
 for(int j=0;j<managers.length;j++){
   System.out.println(managers[j].getName());
 }
```

We can access `getName()`, but implement the same functionality for each type separately.

3.3 Introducing a Common Supertype

We can define a common superclass, and extract the common parts of these classes:

```
class Employee {
  String name;

  Employee () {}
  Employee (String s) { name = s; }

  String getName() { return name; }
}
```

3.3 Reusing Common Members

Now we may define `Secretary` and `Manager` to be subclasses of this class:

```
class Secretary extends Employee {
  Secretary (String s) { name = s; }
  int getSalary() { return 25000; }
}

class Manager extends Employee {
  Secretary secretary;
  Manager (String n, Secretary s) {
    name = n; secretary = s;}
  int getSalary() { return 65000; }
  Secretary getSecretary() { return secretary; }
}
```

3.3 Advantages of Common Supertypes

- Now, we may always use an instance of `Secretary` or `Manager` wherever we use an instance of `Employee` .

```
Employee[] employees = new Employee[10];
Secretary s1 = new Secretary("bert jones");
employees[0] = s1;
employees[1] = new Manager("anne smith", s1);
employees[2] = new Employee("charlie wood");
...
// now it compiles and works for all employees:
for(int i=0;i<employees.length;i++){
  System.out.println(employees[i].getName());
}
```

3.4 Shadowed Variables in Subclasses

An instance variable in a subclass shadows an instance variable of the same name in the parent class.

Assume that Employee has a field salary of type int with a default value 15000. Then, the corresponding field salary in Manager shadows it:

```
class Manager extends Employee {
  int salary;  // shadows salary field of Employee
  Manager (String n, Secretary s) {
    name = n; secretary = s; salary = 65000;
  }
  int getSalary( ) { return salary; }
}
```

When variables are shadowed, they are not replaced - just masked! Both variables still exist. Methods of the superclass see the original variable, and methods of the subclass see the new version.

3.4 More on Shadowing

The variable in the subclass can also have a different type from the variable it shadows. We could, for example, have:

```
class Manager extends Employee {
  double salary;
  Manager (String n, Secretary s) {
    name = n; secretary = s; salary = 3.00e+5;
  }
  int getSalary( ) { return new Double(salary).intValue();}
}
```

If we need to access the original `salary`, we can use the `super` reference (or a reference of the supertype):

```
 int s = super.salary;
```

Avoid shadowing of variables!

3.5 Casting

Why couldn't we say:

```
Employee[] employees = new Employee[10];
employees[0] = new Secretary("bert jones");
employees[1] = new Manager("anne smith", employees[0]);
```

And why doesn't this compile:

```
System.out.println(employees[0].getSalary());
```

After all, the `employees[0]` reference points to a `Secretary` object, so both the last two lines seem fine?

3.5 Casting

A cast explicitly tells the compiler to change the apparent type of an object reference. Casts only affect the treatment of references; they never change the form of the actual object, just the compiler's notion of it.

```
Employee[] employees = new Employee[10];
employees[0] = new Secretary("bert jones");
employees[1] = new Manager("anne smith",
                           (Secretary)employees[0]);

System.out.println(((Secretary)employees[0]).getSalary());
```

What happens if we were wrong about the type of `employees[0]`?

3.5 Operator Instanceof

The operator `instanceof` is used to compare an object against a particular type, and can be used to determine the type of an object at runtime.

```
assert employees[0] instanceof Secretary;
assert employees[0] instanceof Object;
assert ! employees[0] instanceof Manager;
```

If you aren't sure if the `Employee` reference points to a `Secretary` or `Manager`, you can check with the `instanceof` operator:

```
if (employees[0] instanceof Secretary)
  ((Secretary)employees[0]).getSalary();
else if (employees[0] instanceof Manager)
  ((Manager)employees[0]).getSalary();
```

Avoid casting!

3.6 Access Modifiers

- By default, attributes and methods of a class are accessible within the class itself and to other classes in the same package. This default level of visibility is called *package-private*.
- The other levels of visibility are defined explicitly, using the keywords `public`, `protected` and `private`.

Exercise: Order the rows in the following table according to their restrictiveness:

private	
public	
package-private	
protected	

3.6 Access Modifiers

- By default, attributes and methods of a class are accessible within the class itself and to other classes in the same package. This default level of visibility is called *package-private*.
- The other levels of visibility are defined explicitly, using the keywords `public`, `protected` and `private`. The table below summarises their effect:

private	no visibility outside of the class
package-private	visible to the classes in the package
protected	visible to the classes in the package and to subclasses inside or outside the package
public	visible to all classes

3.6 Encapsulating Attributes with Accessors

- It has several advantages to *encapsulate* attributes — make them private and provide methods for accessing them (*accessors*).
- Accessor methods come in two flavours: *setters* and *getters*. A setter modifies the value of an attribute, whereas a getter retrieves its value.
- Accessors help to hide implementation details so that we can later on revise those details (e.g., choice of data structure).
- By having, at most, two control points from which an attribute is accessed, it is easier to enforce invariants.

The getter and setter for the new `name` attribute of the `Employee` class:

```
private String name;

// Gets the Employee's name
public String getName()
{
  return name;
}

// Sets the Employee's name
public void setName(String name)
{
  this.name = name;
}
```

An alternative implementation of the setter method might verify that the name is valid:

```
// Sets the Employee's name
public void setName(String name)
       throws InvalidArgumentException
{
  if (validName(name)) {
      this.name = name; }
  else {
      throw new InvalidArgumentException(); }
}
```

3.6 Visibility of Accessors

- You should always try to keep the level of visibility of features as low as possible. This also applies to accessors.
- You should make accessors private if you only need to access them within the class.
- You should make accessors protected, if only subclasses need to access the attributes.
- Only when an external class needs to access an attribute should you make the appropriate getter or setter public.

3.6 Pen & Paper Exercise

Find all errors:

```
        class Point {
            int x, y;
            private Point() { reset(); }
            Point(int x, int y) { this.x = x; this.y = y; }
            private void reset() { this.x = 0; this.y = 0; }
        }
        class ColoredPoint extends Point {
            int color;
            void clear() { reset(); }
        }
        class Test {
            public static void main(String[] args) {
                ColoredPoint c = new ColoredPoint(0, 0);
                c.reset();
            }
}
```

3.6 Pen & Paper Exercise

Find all errors:

```
    class Point {
        int x, y;
        private Point() { reset(); }
        Point(int x, int y) { this.x = x; this.y = y; }
        private void reset() { this.x = 0; this.y = 0; }
    }
    class ColoredPoint extends Point {
        int color;
        void clear() { reset(); }  // error
    } // error
    class Test {
        public static void main(String[] args) {
            ColoredPoint c = new ColoredPoint(0, 0);  // error
            c.reset();  // error
        }
}
```

3.7 Abstract Methods and Classes

In Java, it is possible to declare a method without implementing it. Such methods are declared using the abstract modifier:

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

A class containing such a method must be explicitly declared as abstract and cannot be instantiated:

```
// does not compile:
Employee e = new Employee();
```

Subclasses must provide an implementation of all abstract methods in order to be non-abstract.

3.7 Interfaces

An *interface* can be thought of as a purely abstract class that contains only abstract methods.

```
interface FireOfficer {
  public abstract boolean evacuate( ) ;
}
```

Analogous to classes, every interface introduces a new type. To become a subtype of an interface, a class can declare that it `implements` an interface and provide an implementation of the interface methods:

```
class Secretary extends Employee implements FireOfficer {
  ...
  boolean evacuate( ) { ... }
}
```

3.7 Using Interface Types

- Types introduced by interfaces are used in the same way as those introduced by classes. Any class that implements the `FireOfficer` interface has an extra type: `FireOfficer`.

```
FireOfficer fo = new Secretary();
fo. evacuate( );
//does not compile:
FireOfficer fo = new Manager();
```

- Arbitrary many interfaces can be implemented by a single class in Java, which is one of the advantages of using interfaces.
- The methods of an interface are always considered `public` and they are always `abstract`; it is optional to declare them so.

3.7 Interface Variables

Interfaces are normally created to contain a list of abstract methods.
Interfaces may also contain `static final` variables, that is, constants.
These constants appear in any class that implement the interface.
Any interface variables are implicitly `static final`, so these modifiers are optional.
Interfaces are sometimes empty. These interfaces are used to mark that a class has some special property. For example, the interface

```
java.io.Serializable
```

is used to mark that a class allows serialisation.

3.7 Interface Inheritance

- An interface may `extend` another interface with additional methods or constants:

```
interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput out)
                  throws IOException
  public void readExternal(ObjectInput in)
                  throws IOException,
                         ClassNotFoundException
}
```

- Such an interface is called subinterface.
- An interface may extend an arbitrary number of other interfaces.

Part 4

Exceptions

4.1 What are Exceptions?

- Programmers should always be prepared for the worst: errors occur, and unexpected events happen.

```
public void saveToFile(String filename) {
  FileOutputStream fd = new FileOutputStream(filename);
  processFile(fd);
  fd.close();
}
```

- Java provides a mechanism to deal with this — *Exceptions.*
- An *exception* is an event that occurs during the execution of a program that disrupts the flow of instructions and prevents the program from continuing along its normal course.

4.1 Declaring Methods with Exceptions

In Java, we can declare that a particular exception can occur during the execution of a method by adding a throws declaration to the method signature:

```
public void saveToFile(String filename)
            throws FileNotFoundException, SecurityException {
  FileOutputStream fd = new FileOutputStream(filename);
  processFile(fd);
  fd.close();
}
```

4.1 Exception Handling

When we call a method, we wrap the corresponding lines in a `try` block, and test for possible exceptions in respective `catch` branches.

```
try {
  saveToFile(filename);

} catch (FileNotFoundException e) {
  System.out.println("Cannot find file " + filename);

} catch (SecurityException e) {
  System.out.println("Cannot write to file due to " +
                     " security vialations: " + filename);
}
```

When an exception is thrown, control transfers directly to the `catch` block that handles the error.

4.1 Clause Finally

The try/catch statement permits an optional last clause, labelled `finally`. Statements contained in the `finally` block will be executed regardless of whether or not an exception is raised.

```
FileOutputStream fd = new FileOutputStream(filename);
try {
  processFile(fd);
} catch (IOException e) {
  System.out.println("Something wrong with " + filename);
} finally {
  System.out.println("Closing file " + filename);
  fd.close();
}
```

The `finally` block is executed even if a `return` statement is called within the try/catch block.

4.2 Exceptions in the Class Hierarchy

Exceptions are a subtype of `Throwable`.

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Error
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
```

Another kind of throwable in Java is `java.lang. Error`. Instances of `java.lang. Error` are meant to indicate serious problems that a reasonable application should not try to catch, e.g., OutOfMemoryError, StackOverflowError, ThreadDeath. It is a strong convention that Errors are reserved for the use by the JVM.

4.2 Two Types of Exception

Checked Exceptions (`java.lang.Exception`):

- To be used for recoverable errors
- Require a try/catch block or propagation
- Examples: FileNotFoundException, InterruptedIOException

Unchecked Exceptions (`java.lang.RuntimeException`):

- To be used for programming errors, e.g., to indicate that a the user of a method did not adhere to the methods's contract
- Does not require a try/catch block or propagation
- Examples: ArrayIndexOutOfBoundsException, ClassCastException, ArithmeticException

Use checked exceptions for recoverable conditions and unchecked ones for programming errors.

4.2 Defining Exception Classes

- Defining new Exception classes works in the same way as defining classes in general:

```
class ParseException extends Exception {
  ParseException() { super(); }
  ParseException(String s) { super(s); }
}
```

  Of course, you may extend this definition with new fields or methods as well.

- To make the new exception unchecked, extend RuntimeException instead of Exception.

4.3 Throwing Exceptions

We can now write:

```
public void parseFile(File f)
  throws ParseException{
  if (!f.canRead())
    throw new ParseException(
      "Could not read file " + f);
}
```

Now all calls to `parseFile(File f)` have to be guarded in a try/catch block against this exception.

If we do not want to handle the `ParseException` at these places, we can propagate the exception back to the caller, since he may have more information about what corrective action to take.

4.3 Exceptions Summary

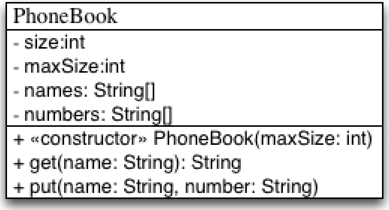
The exception mechanism in Java is better than returning an arbitrary value, such as `null`, because it:

- gives the programmers a type-safe and precise way to return information about different kinds of errors;
- provides means to force a method caller to take notice of the possible exceptional situations;
- allows us to pass the information to the right place in code and handle it there.

Part 5

Practical 1.2

5.0 Practical 1.2


PhoneBook
- size:int
- maxSize:int
- names: String[]
- numbers: String[]
+ «constructor» PhoneBook(maxSize: int)
+ get(name: String): String
+ put(name: String, number: String)

Assessment:

- As the number of entries grows, the access time becomes unacceptable.
- Fixed number of entries.

For an industrial version, we need a better data structure behind the same abstract interface.

5.0 Practical 1.2

- There are many ways to represent the mapping:
    - Use an array of keys and an array of values, with linear search.
    - The same, but keep the keys in order and use binary search.
    - Use a hash table.
    - ...

5.0 Hashing

- The technique called **hashing** works by transforming the key object k into a number, called a **hash value**, in the range $[0..N-1]$.
- It then uses this hash value as an array index to determine where in an N-element array $table[]$, called a **hash table**, we should store the key and the corresponding value objects.
- That is, given a (k, v) tuple, we apply a function f, called a **hash function**, to the key object k to obtain an index $i = f(k)$ and then store (k, v) into array location $table[i]$.

5.0 HashMap Class in the Java Library

Package `java.util` provides this class:

```
public class HashMap extends AbstractMap
                  implements Serializable, Cloneable, Map {...}
```

- This class implements the Map interface using a hashtable, which maps keys to values. Any object can be used as a key or as a value.
- To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the `hashCode()` method and the `equals()` method. The class `String` does this.

5.0 Using Map and HashMap

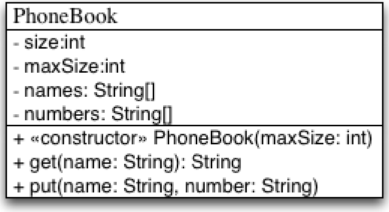
To store a number, use the following code:

```
Map names2numbers = new HashMap();
names2numbers.put("one", new Integer(1));
names2numbers.put("two", new Integer(2));
names2numbers.put("three", new Integer(3));
```

To retrieve a number, use the following code:

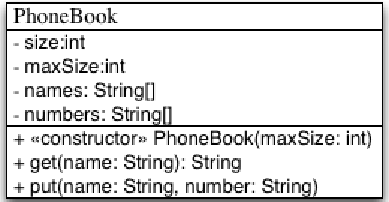
```
Integer n = (Integer) names2numbers.get("two");
System.out.println("two = " + n);
```

5.0 Practical 1.2


PhoneBook
- size:int
- maxSize:int
- names: String[]
- numbers: String[]
+ «constructor» PhoneBook(maxSize: int)
+ get(name: String): String
+ put(name: String, number: String)

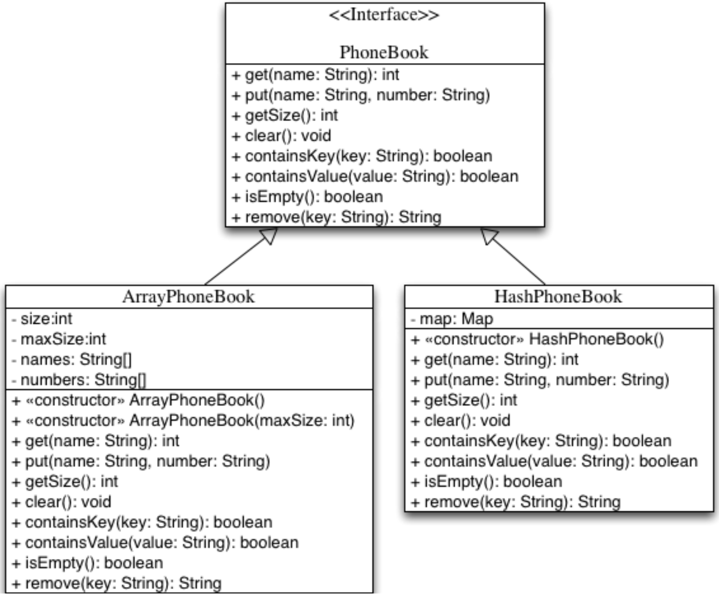
- How to introduce the new alternative implementation with a minimum effort?

5.0 Practical 1.2


PhoneBook
- size:int
- maxSize:int
- names: String[]
- numbers: String[]
+ «constructor» PhoneBook(maxSize: int)
+ get(name: String): String
+ put(name: String, number: String)

- We could extract an interface from the current class. If we call the interface `PhoneBook` and rename the class into `ArrayPhoneBook`, we only need to modify the instantiation of `PhoneBook` in our existing code.
- What does the interface `PhoneBook` look like?

5.0 Practical 1.2



<<Interface>>
PhoneBook
+ get(name: String): int
+ put(name: String, number: String)
+ getSize(): int
+ clear(): void
+ containsKey(key: String): boolean
+ containsValue(value: String): boolean
+ isEmpty(): boolean
+ remove(key: String): String
ArrayPhoneBook
- size:int
- maxSize:int
- names: String[]
- numbers: String[]
+ «constructor» ArrayPhoneBook()
+ «constructor» ArrayPhoneBook(maxSize: int)
+ get(name: String): int
+ put(name: String, number: String)
+ getSize(): int
+ clear(): void
+ containsKey(key: String): boolean
+ containsValue(value: String): boolean
+ isEmpty(): boolean
+ remove(key: String): String
HashPhoneBook
- map: Map
+ «constructor» HashPhoneBook()
+ get(name: String): int
+ put(name: String, number: String)
+ getSize(): int
+ clear(): void
+ containsKey(key: String): boolean
+ containsValue(value: String): boolean
+ isEmpty(): boolean
+ remove(key: String): String

Part 6

Nested Classes

6.0 Nested Classes

Nested classes are classes declared in a scope of another class or interface, also called *member classes*.

```
public class PhoneBook{
    class Entry{
        private String name;
        private String number;
        // getters and setters:
          ...
    }
    ...
    private Entry[] entries;
}
```

6.0 GUI Example without Inner Classes

The following implementation can be made more succinct with an inner mouse listener class:

```
public class HelloJava3
    extends JComponent implements MouseInputListener {
  ...
  public HelloJava3(String message) {
    this.message = message;
    addMouseListener(this);
  }
  public void mouseReleased(MouseEvent e) { ... }
  public void mouseMoved(MouseEvent e) { }
  ...
}
```

6.0 GUI Example with Inner Classes

Rather than implementing the MouseInputListener interface, we can extend the class MouseInputAdapter that implements all methods:

```
public class HelloJava3 extends JComponent {
  public HelloJava3(String message) {
    ...
    MouseInputListener mml = new MyMouseListener( );
    addMouseListener(mml);
  }

  class MyMouseListener extends MouseInputAdapter {
    public void mouseReleased(MouseEvent e) { ... }
  }
  ...
}
```

6.0 Example Summary

- Within the body of *MyMouseListener* we have access to all members of *HelloJava3*.
- Conversely, we have also access to *MyMouseListener* from any method of *HelloJava3*.
- A *MyMouseListener* always lives within a single instance of *HelloJava3*, referred to as the *enclosing instance*.
- Outside of the class, the inner class can be referred to via *HelloJava3.MyMouseListener*.
- Modifiers can be applied to restrict the visibility of the inner class.

6.0 Scoping of the `this` Reference

An inner class has multiple **`this`** references, because it has one or more enclosing instances. The appropriate one may be specified by prepending the name of the class.
This is useful when we need to pass a reference to the parent, or to refer to a shadowed variable:

```
public class HelloJava3 extends JComponent {
  int x;
  class MyMouseListener extends MouseInputAdapter {
    int x;
    public void mouseDragged(MouseEvent e) {
      HelloJava3.this.x = 5;    //refers to the outer x
    }
  }
}
```

6.0 Inner Classes in Methods

- Inner classes may also be defined within method bodies.
- In that case, the body of the inner class can also see all the local variables in the method.
- Issues with scoping and lifetime: objects can live as long as they are referenced, while methods have statically limited lifetimes.
- Solution: any of the methods local variables that are referenced by the inner class must be declared `final`. This means that they are constant once assigned.
- Inner classes in methods cannot be declared `static`.

6.0 Anonymous Inner Classes

When a class definition is needed only once, we can avoid naming it, that is, we can leave it anonymous. With anonymous inner classes, the class's declaration is combined with the allocation of an instance of that class.

```
public class HelloJava3 extends JComponent {
  public HelloJava3(String message) {
    ...
    addMouseListener(new MouseInputAdapter() {
      public void mouseReleased(MouseEvent e) { ... }
    });
  }
  ...
}
```

Use anonymous inner classes when you only have a few lines of code.

6.0 Lambda Expressions

For anonymous classes created from functional interfaces (interfaces with exactly one abstract method) there is a shortcut syntax inspired by functional programming languages.

```
addComparator(new Comparator() {
   public int compare(Object o1, Object o2){
      ...
 });
```

The syntax is referred to as *lambda expressions*:

```
addComparator(  (o1, o2) -> { ... }  );
```

The argument of addComparator looks like a function, but is only a shortcut syntax for creating an anonymous class.

6.0 Static Member Classes

- Member classes can be declared as static (default in interfaces):

```
public class ArrayPhoneBook{
    int size;
    public static class Entry{
        private String name;
        private String number;
        public String getName(){
          // size is not visible here
          ...
        } ...
    } ...
}
```

- Just as a static methods, static member classes also have no current instance of the enclosing class. Therefore, a static member class cannot reference a non-static feature of the enclosing class.

Part 7

Iterators

7.1 Saving Phone Books

Suppose we also want to implement a `save()` method that writes the phone data to a file.

- We need to work through the keys one by one.
- We'd like to separate I/O details from details of the data structure, but without exposing our data representation details.

This problem goes beyond the interface we have so far.

7.1 Design Choice A

What happens if we mix the concerns?

We could provide a method

```
public void writeTable(PrintStream out);
```

in the `PhoneBook` interface.

This solution mixes I/O with data structure. It results in a disastrous explosion in complexity when multiple file formats meet multiple internal representations!

7.1 Design Choice B

Add a method

```
public String[] getKeys();
```

that returns a new array containing all the keys (i.e. it explicitly copies all the strings into a new array).

Pros and cons:

- It gives perfect separation of concerns.
- It also gives a huge transient consumption of storage.

7.1 Design Choice C: Iterators

Iterators constitute a uniform framework for providing access to a collection of values, without compromising the encapsulation of the container.

Standard Java interface `Iterator` is in package java.util:

```
 interface Iterator {
  boolean hasNext();
  Object next();
  void remove();
}
```

We add a new method to the mapping interface:

```
    Iterator iterator();
```

We can then implement the `save` method in a different class using the iterator:

```
  private void save() {
    Iterator it = book.iterator();
    while (it.hasNext()) {
      String name = (String) it.next();
      String number = book.get(name);
      writeLineToFile(name + "," + number);
    }
  }
```

7.1 Implementing the Interfaces

We need a new class `ArrayIterator` which implements the `Iterator` interface.

```
class ArrayIterator implements Iterator {
  ...
}

public Iterator iterator() {
     return new ArrayIterator (names, size);
}
...
```

Pros and cons:

- Solution C achieves good separation of concerns.
- It supports multiple simultaneous traversals.
- It requires new classes.

7.1 Foreach Loops and Iterable

Foreach loops in Java are shortcuts for loops involving iterators:

```
for (Object n : phonebook) { System.out.println((String)n); }
```

The foreach loop above is translated to

```
for (Iterator it = phonebook.iterator(); it.hasNext(); ){
    Object n = it.next(); System.out.println((String)n);
}
```

Foreach loops can be applied to any object that implements the interface `Iterable`:

```
interface Iterable {
  public Iterator iterator();
}
```

7.1 Pen&Paper: Lambda Expressions

The following function `wrap` packages an `Iterator` as an `Iterable`:

```
static Iterable wrap(final Iterator i) {
    return new Iterable() {
        public Iterator iterator() {
            return i;
        }
    };
}
```

Rewrite the function using a lambda expression.

7.1 Pen&Paper: Lambda Expressions

Previous implementation:

```
static Iterable wrap(final Iterator i) {
    return new Iterable() {
        public Iterator iterator() {
            return i;
        }
    };
}
```

New implementation:

```
static Iterable wrap(final Iterator i) {
    return () -> i;
}
```

7.1 Concurrent Modification

- While an iterator is in existence, new keys are added to the mapping.
- What happens?
- The `Iterators` returned by the `iterator` and `listIterator` methods in the Java library are fail-fast: if the `HashMap` is structurally modified at any time after the `Iterator` is created, in any way except through the `Iterator`'s own remove method, the `Iterator` will throw a `ConcurrentModificationException`.
- In the face of concurrent modification, the Iterator fails quickly and cleanly using a `ConcurrentModificationException`; rather than risking arbitrary, non-deterministic behaviour at an undetermined time in the future.

7.1 Benefits

Benefits brought by iterators:

- Iterators make it easy to support and switch between different traversal strategies for complex aggregates. We can implement different iterator classes and switch between them in a transparent way.
- Iterators simplify the interface of the aggregate.
- Iterators allow us to have several parallel traversals on the same aggregate.

7.1 Practical 1.3

Add the following method to the interface `PhoneBook`:

```
public Iterator iterator();
```

For this, you may need to introduce a nested class that implements the `Iterator` interface in the array-based implementation of `PhoneBook`.

Part 8

Overloading and Overriding

8.1 Method Overloading

- In java, it is possible to *overload* a method—define another method with the same name but different parameters:

```
public class PrintStream{
    public void println(){...}
    public void println(String input){...}
    public void println(boolean input){...}
    // 10 different implementations in total  ...
}
```

- When the method is invoked, the compiler picks the correct one based on the arguments passed to the method **at compile time based on the static type of objects**.
- If there is no exact type match, the compiler searches for an assignable match. If there is more than one assignable match, the compiler chooses the most specific match.

8.1 Overloaded Constructors

Constructors, like methods, can be overloaded.

```
class Secretary {
  String name;
  public Secretary(String name) {
    this.name = name; ...
  }
  public Secretary() {    // the default constructor
    this("jane");
  }
}
```

8.1 Statements `this()` and `super()`

The special statements **`this()`** and **`super()`** are used, respectively, to invoke another constructor of the same class, and to invoke a constructor of a superclass, upon entry to another constructor. Usage:

- **`this(...)`** is used with overloaded constructors.
- **`super(...)`** is used when we want to invoke a superclass constructor that takes arguments (rather than use the implicit call to the default constructor).

```
public class Manager extends Employee {
  Secretary secretary;
  public Manager (String n, Secretary secretary) {
    super(n);
    this.secretary = secretary;
  }
}
```

8.1 Constructor Chaining

Upon a call to a constructor (eg. `new Manager("fred")`), the following chain of events happens:

1. Superclass constructor is invoked, either implicitly (the default constructor), or explicitly. This causes the superclass's superclass constructor to be invoked, etc, up to `Object`.
2. Instance variables of the class are initialised upon return from the superclass constructor.
3. The statements of the current constructor are executed.

Constructors are thus *chained* together (ie. successively invoked, according to the inheritance hierarchy) rather than inherited.

8.2 Redefinition or Overriding

- In Java, it is possible to re-implement a method of the superclass in a subclass. We say that the second method *overrides* the first.

```
class Employee {
  ...
  int getSalary() { return 10000; }
}
class Secretary extends Employee {
  ...
  int getSalary() { return 25000; }
}
```

- The new method needs to have the same signature as a method in a parent class except that the return type can be made more specific.

8.2 Dynamic Binding

- Overridden methods are selected dynamically at runtime. This is called *dynamic binding*.
- What is printed here?

```
Employee[] employees = new Employee[10];
employees[0] = new Secretary("bert jones");
...
System.out.println(employees[0].getSalary());
```

- Due to dynamic binding, the method defined in the `Secretary` class will be executed.

8.2 Overloading vs Overriding Methods

- Overloaded methods have the same name but different arguments. Both implementations are available on the same instance. Most suitable method is chosen at compile time.
- Overriding of methods happens when the new method replaces the corresponding method in the parent class. The old method is not available to the external caller on the instance of the subclass. Most suitable method is chosen at runtime.

8.2 Restrictions on Overridden Methods

You will need to remember the following when overriding methods:

- You can't override a `static` method.
- The overriding method must adhere to the `throws` clause of the overridden method.
- The overriding method must be at least as visible as the overridden method (eg. you can't override a `public` method with a `private` method).
- Methods with the `final` modifier can not be overridden.

8.3 Statements `this` and `super`

- Sometimes we still need to refer to the initial implementation of the overridden method.
- `super` is used to access shadowed variables from the superclass, and to access overridden methods from the superclass.

```
class Manager extends Employee {
  int getSalary( ) {
    int bonus = ...; // do some bonus calculation
    return bonus + super.getSalary();
  }
}
```

8.3 Pen & Paper Exercise

Which methods are called when?

```
public class C {
 public void op (C c) { ... } // 1
}
```

```
public class SC extends C {
  public void op (C c) { ... } // 2, overriding
  public void op (SC c) { ... } // 3, overloading
}
```

```
public static void main(String[] args) {
  C c = new C(); C c2 = new SC(); SC sc = new SC();
  sc.op(c); sc.op(c2); sc.op(sc);
  c2.op(c); c2.op(c2); c2.op(sc);
  c.op(c);  c.op(c2);  c.op(sc);
 }
```

8.3 Pen & Paper Exercise Answer

Which methods are called when?

```
public class C {
 public void op (C c) { ... } // 1
}
```

```
public class SC extends C {
  public void op (C c) { ... } // 2, overriding
  public void op (SC c) { ... } // 3, overloading
}
```

```
public static void main(String[] args) {
  C c = new C(); C c2 = new SC(); SC sc = new SC();
  sc.op(c); sc.op(c2); sc.op(sc);  // 2 2 3
  c2.op(c); c2.op(c2); c2.op(sc);  // 2 2 2
  c.op(c);  c.op(c2);  c.op(sc);   // 1 1 1
}
```

8.4 Pen & Paper Exercise

Given is the following interface:

```
public interface IMap {
    boolean put(String key, String value);
    String get(String key);
    Iterator getKeys();
    int putAll(IMap s);
}
```

Assume that the class ArrayMap implements exactly this interface.

Task: On paper, implement a more specific class CountingArrayMap by extending ArrayMap so that the new class keeps record of its size in an attribute and has a getter getSize().

8.4 Pen & Paper Exercise: Alternatives

1. Override both, use super implementation in both and increase size in both
2. Override both, but use only super implementation of put and re-implement putAll
3. Override both, but use only super implementation of putAll and re-implement put
4. Override only put and increase size there
5. Override only putAll and increase size there

Which alternative works depends on the implementation of ArrayMap.

8.4 Inheritance Breaks Encapsulation

- The correct function of a subclass depends on the implementation details of its superclass.
- The superclass's implementation may change and break the subclass even if its code has not been touched (fragile subclass problem).
- As a consequence, a subclass must evolve in tandem with its superclass.
- We have a tight coupling between the subclass and the superclass.

8.4 Exercise: Group Discussion

In a small group, discuss your answers and agree on a common answer to the following questions:

- Does it solve the problem to override the methods but not make use of the super-implementation?
- Is it safe to use inheritance without overriding?
- Should inheritance be avoided in general or are there examples where inheritance is an appropriate choice?
- Can we re-design the class ArrayMap to make inheritance and overriding safer? Is there a way to make sure that ArrayMap does not access any methods from its subclasses?

Find examples to justify your answers. Subsequently, present your answers to the class.

8.4 Exercise: Group Discussion

1. We would fail to reuse code. Subclass still can break with the next release (see next point).
2. It is not gueranteed that we will not be overriding methods in the next release. If that method is final, our class is broken.
3. It is safe if the superclass's authors have designed and documented it specifically for the purpose of being extended. Good examples are MouseInputAdaper, Object.
4. We can use private helper methods internally. We should document our class for extension.

8.4 Summary

- (Class) Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job.
- It is particularly critical to use inheritance where information hiding is essencial.
- It is safe if the superclass's authors have designed and documented it specifically for the purpose of being extended.

Design your classes for extension or forbid it!
Design your methods for overriding or forbid it!

8.4 Delegation Example

ContingArrayMap with delegation:

```
public class ContingArrayMap implements IMap
{
    private IMap map = new ArrayMap();
    private int size;

    public boolean put (String key, String value){
       if( map.put(key,value)){
           size++;
           return true;
       }
       return false;
     }
     ...
}
```

8.4 Practical 1.4

Implement two more subtypes of `PhoneBook`:

1. `CountingArrayPhoneBook` which has a public `getModificationsCount()` method and reuses code from `ArrayPhoneBook` by extending it; refactor the class `ArrayPhoneBook` accordingly to make inheritance safe;
2. `DelegatingPhoneBook` with the same functionality, but using *delegation*. In order to reuse the implementation from `ArrayPhoneBook`, `DelegatingPhoneBook` shall have an instance variable of type `ArrayPhoneBook` and forward all method calls to this instance while keeping track of modifications. To establish an appropriate subtyping relation, this class shall implement the `PhoneBook` interface.

8.4 Delegation versus (Class) Inheritance

Inheritance cons:

- Single inheritance in Java can be too limiting in a particular case.
- Inheritance results in a tight coupling between the superclass and the subclass.
- Designing a class for extension requires substantial additional documentation and results in a more complex usage contract. It is not always possible to design a class for extension as it would require to guarantee the persistence of the exposed details for backwards-compatibility.
- If the superclass has not been designed for extension, relying on inheritance is error-prone.

Delegation cons:

- Implementation with delegation tends to be more verbose due to a larger number of forwarding methods.

Part 9

Overriding Methods of Object

9.1 Overriding Methods of `Object`

The class `java.lang.Object` is the ancestor of all objects. It is often required to override the following methods:

- `boolean equals(Object o)` is used to determine whether two objects are equivalent. Equivalence is different from equality for reference types.
- `int hashCode()` returns a unique integer to be used as a hashcode for the object. Hashcodes are used in the process of storing objects in `HashMap`s. If you override `equals`, you should override `hashCode`.
- `String toString()` is called implicitly when the object needs to be represented as a `String`.

9.1 What does `a.equals(b)` mean?

- Reference types: comparison with ==

```
String k=new String ("12");
//prints false
System.out.println(k=="12");
```

- Reference types: comparison with `equals`

```
String k=new String ("12");
//prints true
System.out.println(k.equals("12"));
```

`Equals` is meant to compare the values of objects, not the value of the reference. It has to be overriden in every class to work properly.

9.1 Equals: Properties

Properties of `equals`:

- Reflexive: for any non-null reference value x, x.equals(x) should return true.
- Symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- Transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- For any non-null reference value x, x.equals(null) should return false.

Default implementation: for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object.

9.1 Equals: Example

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    ArrayPhoneBook other = (ArrayPhoneBook) obj;
    if (maxSize != other.maxSize)
        return false;
    ...
    return true; }
```

For a proper function, it is important that equals is overriden for all attributes taken into account during comparison.

9.1 HashCode: Example

```
public int hashCode(){
    final int prime = 31;
    int result = 1;
    result = prime * result + maxSize;
    ...
    return result;
}
```

A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance:

$$31 * i == (i << 5) - i$$

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

9.1 ToString: Example

```
public String toString() {
    return "ArrayPhoneBook ["+
    " size= + size +
    ","+ ...
    + "]";
```

For a proper function, it is important that toString is overriden for all attributes that need to be printed.

9.1 Practical 1.5

Override the following methods in your subtypes of `PhoneBook`:

```
public boolean equals(Object o);
public int hashCode();
public String toString();
```

Part 10

Co-, Contra- and Invariance

10.1 Co-, Contra- and Invariance

Terms *covariance*, *contravariance* and *invariance* describe relation between a simple type A and a type $X(A)$ constructed using A.

- The behaviour of the type construct X is *covariant*, if:
  $A_1 \preccurlyeq A_2 \quad \Rightarrow \quad X(A_1) \preccurlyeq X(A_2)$
- The behaviour of the type construct X is *contravariant*, if
  $A_1 \preccurlyeq A_2 \quad \Rightarrow \quad X(A_2) \preccurlyeq X(A_1)$
- The behaviour of the type construct X is *invariant*, if $X(A_1)$ and $X(A_2)$ are unrelated unless $A_1 = A_2$. In the latter case, we have $X(A_1) = X(A_2)$.

10.1 Array Construct

We know that

$$\texttt{Smart} \preccurlyeq \texttt{Mobile} \preccurlyeq \texttt{Phone}\,!$$

Does this imply that

$$\texttt{Smart}[\,] \preccurlyeq \texttt{Mobile}[\,] \preccurlyeq \texttt{Phone}[\,]\ ?$$

10.1 Example

```
public static void main(String[] args) {
  Integer[] is = new Integer[10];
  Object[] os = is;
  os[0] = new Long(Long.MAX_VALUE);
  Integer myInt = is[0];
  System.out.println(myInt);
}
```

10.1 Covariance of Arrays

In Java, arrays behave covariantly:

$$\texttt{Smart}[\,] \preccurlyeq \texttt{Mobile}[\,] \preccurlyeq \texttt{Phone}[\,]$$

- However, convariant use of arrays is not entirely type-safe.
- Reading from an up-casted array is safe, but when writing into an up-casted array, make sure that the new element has a compatible type.
- Beware of ArrayStoreExceptions!

10.2 Typing Rules for Method Signatures

- In type theory, a class type can be seen as a constructed type defined by its public interface.
- The types used in the public interface include all types involved in method signatures. In Java, those are return types, parameter types and types of exceptions.
- We loosely refer to the corresponding typing rules as covariant (contravariant) return types, parameter types or types of exceptions, if it is acceptable to make those types more specific (general) in an overridden method of a subclass.
- We say that those are invariant, if it is not admissible to vary them in an overridden method of a subclass.

10.2 Running Example

We use the following class as a running example:

```
public class Phone {
    public Contact getContact(Name name)
                throws PhoneBookException {
        ...
    }
}
```

Assume that we have some existing code that uses `Phone`:

```
// we are given a Phone p and a Name name
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

10.2 Introducing New Subtype

Now we extend the class hierarchy and introduce a class SmartPhone.

```
public class SmartPhone extends Phone {
    public Contact getContact(Name name)
                throws PhoneBookException {
        ...
    }
}
```

What are the admissible modifications of the method's signature? Assume that we want to make it applicable in a wider range of contexts, e.g., so that SmartPhone is also usable in parts of the system specific to smartphones? But we also want SmartPhone to be a valid substitute for Phone.

10.2 More Specific Return

Returning more specific objects:

```
public class SmartPhone {
    public Contact getContact(Name name)
                throws PhoneBookException {
        SmartPhoneContact result;
        ...
        return result;
    }
}
```

$$\texttt{SmartPhoneContact} \preccurlyeq \texttt{Contact}$$

Will the existing code work with an instance of the new class?

10.2 More Specific Return

```
// we are given a Phone p and a Name name
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

$\texttt{SmartPhoneContact} \preccurlyeq \texttt{Contact}$

It will work, because existing code expects at least the functionality of a `Contact`.

Using `getContact` in a smartphone-specific context would require a down-cast to `SmartPhoneContact`. Can we avoid it?

10.2 More Specific Return

More specific return type:

```
public class SmartPhone {
    public SmartPhoneContact getContact(Name name)
                throws PhoneBookException {
        SmartPhoneContact result;
        ...
        return result;
    }
}
```

Now, `SmartPhone` is also usable in a smartphone-specific context without a down-cast to `SmartPhoneContact`.

Will existing code still work?

10.2 More Specific Return

```
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

Existing code still works.

⇒ Substitutability is preserved, if the return type of a method in a subclass is more specific.

Can we return a more general type, e.g., `Object`?

10.2 More General Return

More general return type:

```
    public Object getContact(Name name)
                throws PhoneBookException { ... }
```

Existing code:

```
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

This doesn't work, because the caller expects at least `Contact`.

10.2 More Specific Parameters

Can we make parameters more specific while preserving substitutability?

```
public class SmartPhone {
    public Contact getContact(ExtendedName name)
                throws PhoneBookException { ...
    }
}
```

$$\texttt{ExtendedName} \preccurlyeq \texttt{Name}$$

10.2 More Specific Parameters

Can we make parameters more specific while preserving substitutability?

```
// we are given a Phone p and a Name name
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

This doesn’t work, because the existing code only guarantees `name` to be an instance of `Name`, not `ExtendedName`. We cannot demand more specific parameters in a subtype.

10.2 More General Parameters

Can we make parameters more general while preserving the substitutability?

```
public class SmartPhone {
    public Contact getContact(IName name)
                throws PhoneBookException { ...
    }
}
```

$$\texttt{Name} \preccurlyeq \texttt{IName}$$

10.2 More General Parameters

Can we make parameters more general while preserving the substitutability?

```
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

It could work. The existing code makes stronger guarantees for `name` than needed in the new method of `SmartPhone`. In principle, more general parameter types would be acceptable for an overridden method. (However, Java does not support more general parameter types.)

10.2 More General Exceptions

Can we make exceptions more general while preserving the substitutability?

```
public class SmartPhone {
    public Contact getContact(Name name)
                throws GeneralPhoneException { ...
    }
}
```

$\texttt{PhoneBookException} \preccurlyeq \texttt{GeneralPhoneException}$

10.2 More General Exceptions

Can we make exceptions more general while preserving the substitutability?

```
try {
  Contact c = p.getContact(name);
  System.out.println(c.toString());
} catch (PhoneBookException e){
  e.printStackTrace();
}
```

$$\texttt{GeneralPhoneException} \preccurlyeq \texttt{PhoneBookException}$$

Current catch block covers only subtypes of `PhoneBookException`, not its supertypes or unrelated types. `GeneralPhoneExceptions` is not covered. Therefore, substitutability would no longer be preserved.

10.2 More Specific Exceptions

Can we make exceptions more specific while preserving the substitutability?

```
public class SmartPhone {
    public Contact getContact(Name name)
                throws SmartPhoneBookException { ...
    }
}
```

$$\texttt{SmartPhoneBookException} \preccurlyeq \texttt{PhoneBookException}$$

Current catch block covers all subtypes of `PhoneBookException` including `SmartPhoneBookException`. Therefore, substitutability is preserved.

10.2 Co-, Contra- and Invariance

Substitutability principle implies the following typing rules:

Variable method signature part	Behaviour
Parameter types	Contravariant
Exception types	Covariant
Return type	Covariant

In Java, the following typing rules hold:

Variable method signature part	Behaviour
Parameter types	Invariant
Exception types	Covariant
Return type	Covariant

10.2 Pen & Paper Exercise

- Reflect on covariant exception types. Think of an example where the restriction to more specific exception types would be a significant limitation and suggest a strategy to deal with it.

Part 11

Linked Lists and Stacks

11.1 Linked Lists

- Like arrays, linked lists are used to store a sequential list of objects.
- The primary difference between these data structures is that arrays are better at referencing elements via a numeric index, whereas linked lists are more efficient when it comes to inserting and removing elements.
- In other words, arrays are better suited for random access via a numeric index, while linked lists are better suited for accessing data in a purely sequential manner.

11.1 Integer Lists

```
public class IntegerList {
   private int head;
   private IntegerList tail;

   public void setHead(int item) { head = item; }
   public int getHead() { return head; }

   public void setTail(IntegerList next) { tail = next; }
   public IntegerList getTail() { return tail; }
}
```

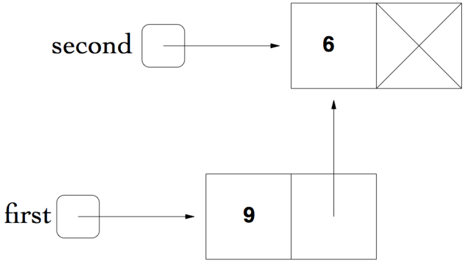
11.1 Using Integer Lists

```
IntegerList num1 = new IntegerList ();
IntegerList num2 = new IntegerList ();

num1.setHead(10);   // set head in the first link
num2.setHead(20);   // set head in the second link
num1.setTail(num2); // connect the links
```

11.1 A Linked List of Objects

```
public class LinkedList {
   private Object head;
   private LinkedList tail;

   public LinkedList (Object item) {
          head = item;
          tail = null; }

   public LinkedList (Object item, LinkedList next) {
          head = item;
          tail = next; }

   public void setHead(Object item) { head = item; }
   public Object getHead() { return head; }

   public void setTail(LinkedList next) { tail = next; }
   public LinkedList getTail() { return tail; } }
```

11.1 Using a Linked List



```
LinkedList second  = new LinkedList (new Integer(6));
LinkedList first = new LinkedList (new Integer(9), second);
```

11.1 Exercise

Write a method that iterates through a list and prints its elements.

11.1 Exercise

Printing the elements of a linked list:

```
public void printList(LinkedList list){
  for (LinkedList cur = list; cur != null; cur = cur.getTail()) {
    System.out.println(cur.getHead());
  }
}
```

11.2 Stacks

- A **stack** is a last-in, first-out data structure in which the only object that can be accessed is the last one placed on the stack. This element is called the **top** of the stack.
- When a new value is inserted into the stack, it becomes the new top element. The process of adding a new element to the stack is called **pushing**.
- A detailed interface for a stack data structure is provided on the following slides.

11.2 Stack Interface

```
public interface IStack {
  public void push(Object data);
  /* Places its argument on the top of the stack
     Pre:
     Post: stack size has increased by 1
           data is on top of the stack
           the rest of the stack is unchanged    */

  public Object pop ();
  /* Returns the top element of the stack
     Pre:  stack is not empty
     Post: stack size is decreased by 1
           top item of stack is returned
           the rest of the stack is unchanged    */
  public boolean isEmpty();
  /* Determines if the stack is empty
     Pre:
     Post: returns true iff the stack is empty
           stack is unchanged                  */
}
```

Which data structure is preferable for implementing a stack? An array or a linked list?

11.2 Implementation of the Stack Interface

```
public class Stack implements IStack {
  private LinkedList top;   // top of stack
  public Stack() { top =  null; }
  public void push (Object item) {
    top = new LinkedList (item, top);
  }
  public Object pop() {
    Object head = top.getHead();
    top = top.getTail();
    return head;
  }
  public boolean isEmpty() { return top == null; }
}
```

Part 12

Java Generics

12.0 Outline

12.1 Old-Style Generics

So far, in order to make our container types applicable to arbitrary element types, we had to sacrifice type-safety. Consider a stack storing integers:

```
Stack stack = new Stack();
stack.push(4711);
stack.push(815);
Integer n = (Integer)stack.pop();
```

Since `stack` is formally a stack of `Objects`, getting an element out of the stack involves a *down-cast*.

The programmer has to keep track of the fact that `stack` is a stack of integers.

12.1 New-Style Generics

Java Generics allow you to abstract over types. A generic interface or class takes a type parameter at initialisation:

```
Stack<Integer> stack = new Stack<Integer>();

stack.push(4711);
stack.push(815);
Integer n = stack.pop();
```

In this way, Generics allow a programmer to mark a stack as being restricted to contain a particular type of elements. Now, the compiler keeps track of the fact that `stack` is a stack of integers. Down-casts are no longer necessary.

Generics enable a type-safe use of containers. At the same time, they allow us to make our code more general.

12.1 A Simple Stack Interface

Here is a generic version of the stack interface:

```
interface IStack<E>
{
    public push(E... arr);
    public E pop();
    public boolean isEmpty() ;
}
```

Type parameters can be used throughout the generic declaration, pretty much where you would use ordinary types.

12.1 Implementing Stack

```
class Stack<E> implements IStack<E>
{
    private class LinkedList {
        private E head;
        private LinkedList tail;

        public LinkedList (E head, LinkedList tail) {
            this.head = head;
            this.tail = tail;
        }
    }

    private LinkedList top = null;
...
```

`LinkedList` is an inner class and can make use of `Stack`'s type parameter `E`.

```
    public void push(E... arr){
        for(E x: arr){
          top = new LinkedList (x, top);
        }
    }

    public E pop() {
        if (top==null)
            throw new NoSuchElementException();
        else {
            E x = top.head;
            top = top.tail;
            return x;
        }
    }

    public boolean isEmpty() { return top == null; }
}
```

12.1 Exercise

Define a generic versions of interfaces List and Iterator that would enable the following use:

```
List<Integer> myIntList = new MyList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

12.1 Exercise

The definitions of the interfaces `List` and `Iterator` are as follows:

```
public interface List<E>{
  void add(E x);
  Iterator<E> iterator();
}

public interface Iterator<E>{
  E next();
  boolean hasNext();
}
```

12.1 Multiple Type Parameters

A class can also have two, or more type parameters:

```
class Pair<A, B>
{
    public A fst;
    public B snd;

    public Pair(A fst, B snd)
    {
        this.fst = fst;
        this.snd = snd;
    }
}
Pair<String, Integer> p =
    new Pair<String, Integer>("nadeschda", 44796583223);
```

12.2 Generics and Subtyping

We know that

$$\texttt{Smart} \preccurlyeq \texttt{Mobile} \preccurlyeq \texttt{Phone}\ !$$

Does this imply that

$$\texttt{Map<Smart>} \preccurlyeq \texttt{Map<Mobile>} \preccurlyeq \texttt{Map<Phone>}\ ?$$

12.2 Exercise

Given are the following objects:

```
Map<Phone>  mp = new Map<Phone>(phone);
Map<Mobile> mm = new Map<Mobile>(mobile);
Map<Smart>  ms = new Map<Smart>(smart);
Mobile freds = new Mobile();
```

Which of the following statements make sense?

```
freds = mp.get();
freds = mm.get();
freds = ms.get();
```

```
mp.put(freds);
mm.put(freds);
ms.put(freds);
```

12.2 Exercise

Reading from a map (freds = □.get();):

```
freds = mp.get(); // NOT ok
freds = mm.get(); // ok
freds = ms.get(); // ok
```

Writing to a map (□.put(freds);):

```
mp.put(freds);     // ok
mm.put(freds);     // ok
ms.put(freds);     // NOT ok
```

12.2 Example with Generics

```
public static void main(String[] args) {
  Stack<Integer> is = new Stack<Integer>();
  Stack<Object> os = is;
 os.push( new Long(Long.MAX_VALUE));
  Integer myInt = is.pop();
  System.out.println(myInt);
}
```

`Stack<Object>` is not substitutable by a `Stack<Integer>` !

12.2 Example with Generics

Reading from an upcast Stack:

```
public static void main(String[] args) {
  Stack<Object> os = new Stack<Object>();
  Stack<Integer> is = os;
 os.push( new Long(Long.MAX_VALUE));
  Integer myInt = is.pop();
  System.out.println(myInt);
}
```

`Stack<Integer>` is not substitutable by a `Stack<Object>` !

Generic types are *invariant*!

12.2 Typing Rules in Java

Typing rules for methods:

Variable method signature part	Behaviour
Return type	Covariant
Exception type	Covariant
Parameter type	Invariant

Typing rules for contructed types:

Contructed type	Behaviour
Arrays	Covariant
Generic types	Invariant

12.2 Practical 2.1

Generalise the current implementation and extract the functionality of associative arrays so that it can be easily reused in other parts of the system:

1. Generalise the interface `PhoneBook` into a generic interface `Map<K,V>`.
2. Generalise the class `ArrayPhoneBook` into a generic class `ArrayMap` that implements the generic interface `Map`.
3. Generalise the class `HashPhoneBook` into a generic class `HashMap` that implements the generic interface `Map`.

12.3 Restrictions in Generic Classes

No access to type parameters in static class members:

```
class Wrapper<E>
{
    E element; // works fine
    static LinkedList<E> allElements; // error
    static LinkedList<E> getAllElements(){...} // error
}
```

Each instance will have its own value of the type parameter. Subsequently, type parameters belong to an instance rather than the class as a whole.

12.3 Restrictions in Generic Classes

The following code would compile:

```
class Wrapper<E>
{
    E element; // works fine
    static LinkedList<Object> allElements; // works fine
    static LinkedList<Object> getAllElements(){...} // works fine
    ...
    Wrapper(E element){
      this.element = element;
      allElements = new LinkedList<Object> (element, allElements);
}
```

The type information gets lost when we store the element in the static container.

12.3 Generic Methods

What if we need the type information?

```
final class StackUtil
{
    static void copy(IStack src, IStack dst) {
            for (Object x : src)
                dst.push(x);
    }
}
```

The above code uses non-parametrised (raw) version of the generic type `IStack`. Compiler will produce a warning and leave it up to the programmer to check the type compatibility of the two `IStack`s.

12.3 Generic Methods

Methods can have their own type parameters:

```
final class StackUtil
{
    static <E>
        void copy(IStack<E> src, IStack<E> dst) {
            for (E x : src)
                dst.push(x);
    }
}
```

- Method `copy` is *generic*, even though the class `StackUtil` isn't.
- The expression <E> after `static` means that the method `copy` works *for all* types `E`.
- The only functionality that we assume is that of `Object`.

12.3 Invoking a Generic Method

Method copy may be invoked as follows:

```
IStack<Integer> s = new ListStack<>();
s.push(1, 2, 3);
IStack<Integer> yas = new ListStack<>();
yas.push(4, 5, 6);

StackUtil.copy(yas, s);
// does not compile:
StackUtil.copy(yas, new ListStack<String>());
```

We can also make the type argument explicit:

```
StackUtil.<Integer>copy(yas, s);
```

12.3 Exercise

- Write a method that accepts an array of any reference type and converts it to a stack.
- Write a second version of the same method that accepts a variable number of arguments and stores them in a stack.

12.3 Exercise: Part 1

```
final class StackUtil {
   public static <T> IStack<T> toStack(T[] arr){
      IStack<T> stack = new ListStack<>();
      for(T elt:arr)
         stack.push(elt);
      return stack;
}
```

12.3 Exercise: Part 2

```
final class StackUtil {
   public static <T> IStack<T> toStack(T... arr){
      IStack<T> stack = new ListStack<>();
      for(T elt:arr)
         stack.push(elt);
      return stack;
}
```

12.3 Exercise

What is the value of T in the following examples?

```
StackUtil. toStack(1,2,3);
StackUtil. toStack("1","2","3");
StackUtil. toStack(new ListStack<Integer>());
StackUtil. toStack();
StackUtil. toStack(1,"2");
```

12.3 Exercise Part 3

What is the value of T in the following examples?

```
// Integer:
StackUtil. toStack(1,2,3);
// String:
StackUtil. toStack("1","2","3");
// ListStack<Integer>:
 StackUtil. toStack(new ListStack<Integer>());
// Object:
StackUtil. toStack();
// Object & Comparable<?> & Serializable:
StackUtil. toStack(1,"2");
```

12.4 Bounded Type Parameters: Motivation

Purely generic function `copy` works *for all* types `E`:

```
static <E> void copy(...);
```

This is quite limiting: we can invoke only `Object` methods on the `E` elements within `copy`.

Sometimes, it is of interest to assume a more specific value of the type parameter. For instance, if we assume that `E` implements the interface `Comparable<E>`, we can invoke the method `compareTo` on instances of `E` within our method.

12.4 Comparable

```
interface Comparable<E> {
  public int compareTo(E that);
}
```

The method `compareTo` provides the so-called *natural ordering*. It returns a value that is negative, zero, or positive depending upon whether the object `this` is less than, equal to, or greater than the argument `that`.

For instance, if we would like to determine the maximum element of a stack, we would need to use `compareTo`.

12.4 Bounds

Java allows us to specify bounds on the type parameter:

```
static <E extends Comparable<E>> E maxElement(IStack<E> stack) {
    if(stack.isEmpty())
        return null;
    E max = stack.iterator.next();
    for (E x : stack)
        if (x.compareTo(max) > 0)
            max = x;
    return max;
}
```

The expression `<E extends Comparable<E>>` means that `E` must be a subtype of `Comparable<E>`, i.e., `E` must support the method `compareTo`. Consequently,

- we can do more with instances of `E`, but
- `maxElement` is less general than without the bound.

12.5 Generic Methods Revisited

Which calls of `copy` are valid?

```
IStack<Person> s1;
IStack<Employee> s2;
IStack<Secretary> s3;
IStack<Person> s;
...
StackUtil.copy(s, s);
StackUtil.copy(s1, s);
StackUtil.copy(s2, s);
StackUtil.copy(s3, s);
```

12.5 Generic Methods Revisited

Which calls of `copy` are valid?

```
IStack<Person> s1;
IStack<Employee> s2;
IStack<Secretary> s3;
IStack<Person> s;
...
StackUtil.copy(s, s); //OK
StackUtil.copy(s1, s);//OK
StackUtil.copy(s2, s);//error
StackUtil.copy(s3, s);//error
```

In principle, it should be possible to also copy elements from an `IStack<Secretary>` into an `IStack<Person>`. What does the implementation look like?

12.5 Generic Methods Revisited

The new method `copy` does exactly the same:

```
final class StackUtil
{
    static
        void copy(IStack<Secretary> src, IStack<Person> dst) {
            for (Secretary x : src)
                dst.push(x);
    }
}
```

Is there a way to make the old method more generic so that it covers this case as well?

12.5 Wildcards with Extends

Java allows us to express our intent using a wildcard:

```
static <E>
    void copy(IStack<? extends E> src, IStack<E> dst) {
        for (E x : src)
            dst.push(x);
}
```

The expression `IStack<? extends E>` means that we invoke `copy` with a stack whose element type is a *subtype* of `E`.

However, the type also means that it is not possible any more to *write* elements of type `E` into `src`, since we do not know how specific `? extends E` really is. Consequently, we cannot make sure that the new element is sufficiently specific.

12.5 Wildcards with Super

We can use a similar wildcard for the destination stack:

```
static <E>
    void copy(IStack<E> src,
              IStack<? super E> dst) {
        for (E x : src)
            dst.push(x);
}
```

The expression `IStack<? super E>` means that we invoke `copy` with a stack whose element type is a *supertype* of `E`.

However, the type also means that we cannot *read* elements of type `E` from `dst`, since we do not know how general `? super E` really is. We only know the upper bound.

12.5 Summary Wildcards

- By using an `extends` wildcard we explicitly express our intent to use our aggregate in a read-only fashion:

$$\texttt{Map<A>} \preccurlyeq \texttt{Map<? extends B>} \quad \text{if} \quad \texttt{A} \preccurlyeq \texttt{B}$$

- By using a `super` wildcard we explicitly express our intent to use our aggregate in a write-only fashion:

$$\texttt{Map<A>} \preccurlyeq \texttt{Map<? super B>} \quad \text{if} \quad \texttt{B} \preccurlyeq \texttt{A}$$

- If you need both reads and writes, do not use wildcards.

12.5 Variance of Generic Types with Wildcards

We know that

$$\texttt{Smart} \preccurlyeq \texttt{Phone}\ !$$

Does this imply that

$$\texttt{Map<? extends Smart>} \preccurlyeq \texttt{Map<? extends Phone>}\ ?$$

What about

$$\texttt{Map<? super Smart>} \preccurlyeq \texttt{Map<? super Phone>}\ ?$$

12.5 Summary Typing Rules

- Generic type constructors with `extends` wildcards are covariant:

$$\texttt{A} \preccurlyeq \texttt{B} \;\Rightarrow\; \texttt{Map<? extends A>} \preccurlyeq \texttt{Map<? extends B>}$$

- Generic type constructors with `super` wildcards are contravariant:

$$\texttt{A} \preccurlyeq \texttt{B} \;\Rightarrow\; \texttt{Map<? super B>} \preccurlyeq \texttt{Map<? super A>}$$

- Generic type constructors without wildcards are invariant:

$$\texttt{A} \preccurlyeq \texttt{B} \;\not\Rightarrow\; \texttt{Map<B>} \preccurlyeq \texttt{Map<A>}$$
$$\texttt{A} \preccurlyeq \texttt{B} \;\not\Rightarrow\; \texttt{Map<A>} \preccurlyeq \texttt{Map<B>}$$

12.5 Exercise

Make the following methods more general using wildcards:

- Method within the class `Map<E>`:

```
public void addAll(Map<E> other){...}
```

- Static method within a class `CollectionUtil`:

```
public static <E> void addAll (Map<E> src, Map<E> dst){...}
```

- Static method within a class `NumberUtil` computing the sum:

```
public static Number sum(Collection<Number> numbers) {...}
```

- Static method within a class `NumberUtil` pushing ten random numbers onto the given stack:

```
  public static void randomFill(IStack<Number> stack){...}
```

12.5 Exercise

- Method within the class `Map<E>`:

```
public void addAll(Map<? extends E> other){...}
```

- Static method within a class `CollectionUtil`:

```
public static <E> void addAll (Map<? extends E> src,
            Map<? super E> dst){...}
```

- Static method within a class `NumberUtil` computing the sum:

```
public static Number sum(Collection<? extends Number> numbers)
{...}
```

- Static method within a class `NumberUtil` pushing ten random numbers onto the given stack:

```
public static void randomFill(IStack<? super Number> stack)
{...}
```

12.5 Summary

Java generics can boost your productivity by making your code both

- more generic (applicable to a larger range of inputs) and
- more type-safe (less work for a programmer).

12.5 Practical 2.2

Complete the implementation of the class `LinkedStack`:

1. Rewrite the given implementation to make use of generics, so that we have a generic data structure.
2. Write a method that appends two `LinkedStack`s; the method signature is currently commented out in the `Stack` interface and the `LinkedStack` class. Bear in mind that the append operation is intended to preserve the order of elements from the second stack.

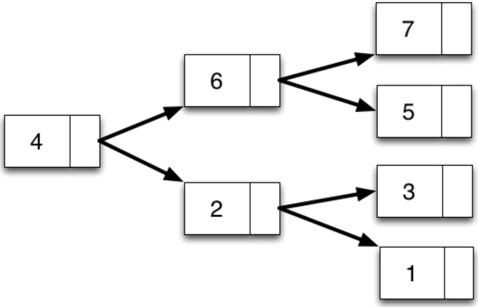
Part 13

More Complex Example: Binary Trees

13.0 Binary trees

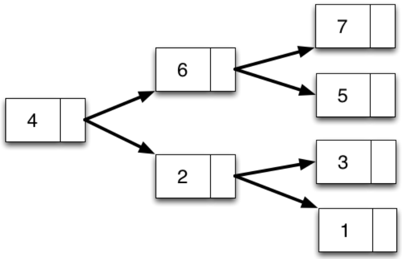
Binary trees can be used for a variety of purposes:

- to organise hierarchical information (pedigrees, expression trees etc) and
- to implement collection types, e.g., ordered sets.



13.0 Binary trees

A *binary tree* is a more complex linked data structure than a linked list. Instead of a single successor, each binary tree node has two childnodes.



A *binary tree* is either

- empty or
- a node that consists of an element, a left tree and a right tree. The left and right tree are also called subtrees.

13.0 Binary search trees

On the following slides we use *binary search trees* to implement *ordered multisets*—ordered collections of elements that possibly contain duplicates.

A *binary search tree* is a binary tree such that

- the left subtree of every node only contains elements less than the element in the node;
- the right subtree of every node only contains elements greater than or equal to the element in the node.

If you visit the nodes in a left-to-right fashion (first the left subtree, then the element in the node itself, then the right subtree), you obtain a non-decreasing sequence of elements.

13.0 Tree Interface

Given is the following minimalistic tree interface (which will be extended later on):

```
public interface Tree<E extends Comparable<E>>
{
  int size();
  int depth();
  boolean member(E elem);
  Tree<E> insert(E elem);
}
```

How do we implement it?

13.0 Tree Implementation

Possible approach:

```
public class MyTree<E extends Comparable<E>> implements Tree<E>
{
    public Tree<E> left;
    public E       item;
    public Tree<E> right;
    ...
}
```

How do we represent leaf nodes?

1. Leaf nodes do not cary data: Subtrees in leaf nodes are null and so is the data element.
2. Leaf nodes cary data: E is never null. Subtrees are null.

Which solution is more preferable and why?

13.0 Tree Implementation

Assume that leaves do not cary data:

```
public class MyTree<E extends Comparable<E>> implements Tree<E>{
    ...
    public int size(){
        int result=0;
        if(null != item){
            result ++;
            if (null != left)  result += left.size();
            if (null != right)  result += right.size();
        }
        return result;
    }
}
```

Too complicated! Can you think of a simpler alternative?

13.0 Tree Implementation

Assume that leaves cary data:

```
public class MyTree<E extends Comparable<E>> implements Tree<E>{
    ...
    public int size(){
        int result=1;
        if (null != left)
            result += left.size();
        if (null != right)
            result += right.size();
        return result;
    }
}
```

Can you think of an alternative design so that the implementation of size would be a simple line? Hint: take advantage of polymorphism.

13.0 Pen & Paper Exercise

On a piece of paper, sketch an implementation of the method `depth`. You can use the method `Math.max(int a, int b)` in your implementation.

13.0 Pen & Paper Exercise

A possible approach to implement the method **depth**:

```
public class MyTree<E extends Comparable<E>> implements Tree<E>{
    ...
    public int depth(){
        int result=1;
        if (null != left)
            result = Math.max(left.depth()+1, result);
        if (null != right)
            result = Math.max(right.depth()+1, result);
        return result;
    }
}
```

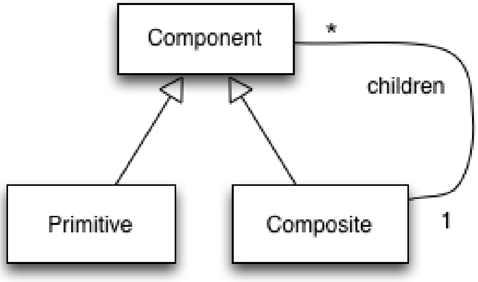
We notice the same kind of conditional statements as in the method **size**. Also here, it would be possible to re-design our implementation so that the method would fit into a simple line.

13.0 Pen & Paper Exercise

In a group of three or four people, design a collection of classes that capture the static aspects of binary trees. The goal is to keep the implementation of the corresponding methods as simple as possible. Ideally, your design would enable you to reduce the implementation of `size` and `depth` to a simple line of code. Keep in mind that you will need to make use of the `Comparable<T>` interface in your implementation. You should use generics to abstract away from the type of the elements contained in a tree.

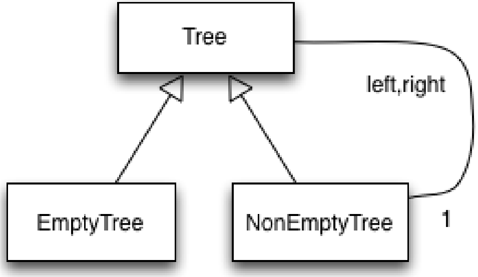
13.0 Composite Pattern

We can take advantage of polymorphism by utilising the composite pattern:



13.0 Applying Composite Pattern

We can take advantage of polymorphism by utilising the composite pattern:



Now we have a separate implementation of the methods for an empty tree, which can be used instead of a null tree. No tests for null are necessary.

13.0 Practical 3

Complete Tasks 1-5 of Practical 3.

13.0 Class NonEmptyTree

```
public class NonEmptyTree<E extends Comparable<E>>
                                          implements Tree<E>{
    ...
    public NonEmptyTree(Tree<E> left, E item, Tree<E> right)
    {
        this.left  = left;
        this.item  = item;
        this.right = right;
    }

    public int size() {
        return left.size() + 1 + right.size();
    }

    public int depth() {
        return 1 + Math.max(left.depth(), right.depth());
    }
```

```
public Tree<E> insert(E elem) {
    if (elem.compareTo(item) < 0)
        left  = left. insert(elem);
    else
        right = right.insert(elem);
    return this;
}

public boolean member(E elem) {
    if (elem.compareTo(item) < 0)
        return left.member(elem);
    else if (elem.compareTo(item) == 0)
        return true;
    else
        return right.member(elem);
}
```

```
    public void dump(int depth)
    {
        left.dump(depth + 1);
        for (int i = 0; i < depth; i++)
            System.out.print("    ");
        System.out.println(item);
        right.dump(depth + 1);
    }
}
```

The implementation of the NonEmptyTree is very succinct due to recursion. No null-checks are necessary.

13.0 Class EmptyTree

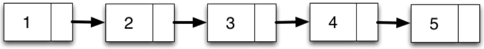
```
public class EmptyTree<E extends Comparable<E>> implements Tree<E>
{
    private static EmptyTree empty = new EmptyTree();

    private EmptyTree(){}

    @SuppressWarnings("unchecked")
    public static <T extends Comparable<T>> Tree<T> getEmptyTree() {
        return empty;
    }

    public int size() { return 0; }

    public int depth() { return 0; }

    public boolean member(E elem) { return false; }
    public void dump(int depth){}

    public Tree<E> insert(E elem)
    {
        return new NonEmptyTree<E>(EmptyTree.<E>getEmptyTree(),
                                  elem, EmptyTree.<E>getEmptyTree());
    }
}
```

- EmptyTree is immutable
- It makes use of the singleton pattern
- One instance is shared across the system

13.1 Balancing Trees

The shape of the binary search tree totally depends on the order of insertions, and it can be degenerated. As a consequence, binary search trees lose their effectiveness.

In fact, the simplest algorithms for item insertion may yield a tree with height n in rather common situations:



A tree is called *balanced* if it has the maximum average out-degree possible for the given sequence of data elements. A balanced binary tree has the minimum possible maximum height (depth), because for any given number of leaf nodes the leaf nodes are placed at the greatest height possible.

13.1 Pen & Paper Exercise

- Draw a balanced tree that contains numbers 1-10 and 1-15.
- Implement the following method:

```
    public Tree<Integer> createBalancedTree(int first, int last);
```

The method should insert the elements of the range [first ... last] in the right order so that the resulting tree is balanced.

13.1 Pen & Paper Exercise

```
public Tree<Integer> createBalancedTree (int first, int last)
{
    if (first > last)
        return EmptyTree.<Integer>getEmptyTree();
    else
    {
        int m = (first + last) / 2;
        return new NonEmptyTree<Integer>
                          (createBalancedTree (first, m-1),
                          m, createBalancedTree (m+1, last));
    }
}
```

13.2 Traversing Trees

External iterators can be difficult to implement for complex composite structures like binary trees.

If the structure spans many levels of nested aggregates, the iterator may have to store the path through the structure.

Alternatively, we can provide an *internal* iterator, which can be easily implemented using recursion.

In case of external iterators, the traversal is controlled by the external user. In contrast, internal iterators control the traversal themselves and apply a function supplied by an external user to each of the elements along the path.

13.2 Reminder: External Iterators

```
public interface ExternalIterable<T> {
    ExternalIterator<T> iterator();
}
public interface ExternalIterator<T> {
    T next();
    boolean hasNext();
}
```

External iterators provide methods `next()` and `hasNext()` to an external user who controls the progress of the traversal.

13.2 Interface: Internal Iterators

```
public interface InternalIterable<T> {
    void traverse(Visitor<T> visitor);
}

public interface Visitor<T> {
    void visit(T t);
}
```

Internal iterators provide a method `traverse` that accepts a visitor who know how to process elements of the strucutre. The iterator keeps control of the traversal and applies the method `visit` provided by the visitor to each element.

13.2 Usage: Internal Iterators

```
public interface Tree<T> extends InternalIterable<T> {...}
...
Tree<Integer> tree = createBalancedTree(1,15);
tree.traverse( new Visitor<Integer> {
    void visit(Integer t){
        System.out.println(t);
    }
};);
```

Tree provides a method `traverse` that accepts a visitor. The visitor is just a wrapper class for a function, which can be rewritten using a corresponding lambda expression.

13.2 Usage: Internal Iterators

```
public interface Tree<T> extends InternalIterable<T> {...}
...
Tree<Integer> tree = createBalancedTree(1,15);
tree.traverse(   (t) -> {System.out.println(t);}    );
```

Within the method `traverse`, we implement the traversal and apply the supplied function to each element along the path.

13.2 Pen & Paper Exercise

Implement a generic static method that can turn an external iterator into an internal one:

```
public <T> InternalIterable<T> internalize
                        (final ExternalIterable<T> ext){ ... }
```

13.2 Pen & Paper Exercise

```
public <T> InternalIterable<T> internalize
                  (final ExternalIterable<T> ext){
    return new InternalIterable<T>() {
        public void traverse(Visitor<T> visitor){
            for (T t: ext){
                visitor.visit(t);
            }
        }
    }
}
```

13.2 Inorder Traversal

To make sure that we traverse the tree in the right order, we need to apply inorder traversal.

The inorder traversal work as follows:

1. Visit all elements in the left subtree.
2. Visit the element of the root node.
3. Visit all elements in the right subtree.

As a result, elements are visited in accending order.

13.2 Practical 3

Complete Tasks 6-7 of Practical 3.

13.2 Practical 3: Traverse

NonEmptyTree implementation:

```
    public void inorder(Visitor<E> v) {
        left.inorder(v);
        v.visit(item);
        right.inorder(v);
    }
```

EmptyTree implementation:

```
    public void inorder(Visitor<E> v){}
```

13.2 Comparison of Iterators

Both kinds of iterators have their benefits and limitations:

- External iterators provide more flexibility.
- With an external iterator, the client can decide when it is a good time to process the next element. In contrast, with an internal iterator, the aggregate pushes item after item to the client.
- Internal iterators are often easier to implement.

Part 14

Immutability

14.0 Outline

14.1 Immutability

- An object is considered *immutable* if its state cannot change after it is constructed.
- Example of immutable classes in Java: String
- Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.
- Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

14.1 Pen & Paper

Given the following code:

```
public class LinkedList<E> {

private E head;
private LinkedList<E> tail;

        public LinkedList(E item) {
            head = item;
            tail = null;
        }

        public LinkedList(E item, LinkedList<E> next) {
            head = item;
            tail = next;
        }
      ...
```

```
        ...
public void setHead(E item) { head = item; }

public E getHead() { return head; }

public void setTail(LinkedList<E> next) { tail = next; }

public LinkedList<E> getTail() { return tail; }

}
```

14.1 Pen & Paper

Extend the class with the following functionality:

- Implement a method `length()` that computes the length of the list.
- Implement a method `append(LinkedList<E> other)` that appends the given list `other` at the end of the current list.

14.1 Pen & Paper

```
int length(){
    if(tail == null) {return 1; }
    return 1+ tail.length();
}

public void append(LinkedList<E> other){
    LinkedList<E> last = this;
    while (last.getTail() != null) { last = last.tail; }
    last.setTail(other);
}
```

14.1 Pen & Paper

```
public static void main(String ar[]){
  LinkedList<Integer> list1 = new LinkedList<>(3);
  LinkedList<Integer> list2 = new LinkedList<>(2, list1);
  LinkedList<Integer> list = new LinkedList<>(1, list2);
  System.out.println(list.length());

  LinkedList<Integer> list4 = new LinkedList<>(4);
  list.append(list4);
  System.out.println(list.length());
}
```

14.1 Observation

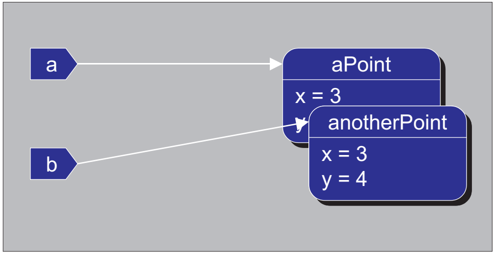
A second append of `list4` generates a cycle in our list, which causes a stack overflow when we call `length`.

Maintaining invariants is difficult if we share our internal objects with others.

How can we fix the method `append`?

We can create a copy of the appended list to make sure that our invariants are fulfilled also for the new part of the list (defensive copying).

14.2 Copying



- There's a built-in method called `clone()` that's supported by every object: it does a bitwise copy. It is not recommended to use this mechanism.
- Alternatively, we can implement a copy method. It is common to make this method a constructor.

14.2 Defining a Copy Constructor

```
class Point {
  private int x, y;
  public Point(int x, int y) {
    this.x = x;  this.y = y;
  }
  public Point copy( ) {   // ordinary copy method
    return new Point(this.x, this.y);
  }
  public Point(Point p) {  // copy constructor
    this(p.x, p.y);
  }
  ...
}
```

14.2 Shallow versus Deep Copies

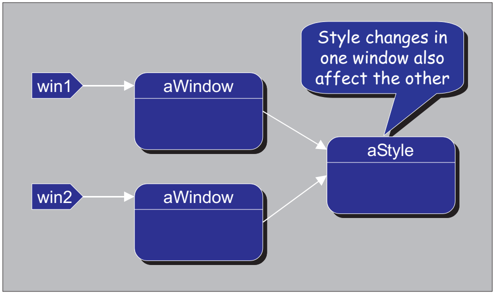
When copying objects, we need to decide how to deal with the network of outgoing references.

Assume that we have a window system where we can

- create a duplicate of a window, and
- change the colour scheme of a window.

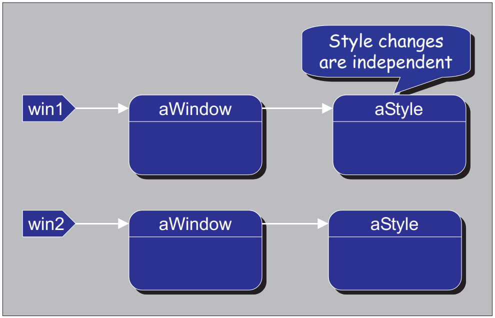
If we duplicate a window, then change the colours of one copy, should the other copy change too?

14.2 Shallow Copying



```
win2 = win1.shallowCopy();
```

14.2 Deep Copying



```
win2 = win1.deepCopy();
```

14.2 Pen & Paper Exercise

Given is the following class:

```
class ColourWindow {
  Style style;
  ColourWindow(Style style) {
    this.style = style;
  }
}
```

Extend it with methods shallowCopy() and deepCopy(). Assume that Style has a method copy().

14.2 Code for Shallow Copying

```
class ColourWindow {
  Style style;
  ColourWindow(Style style) {
    this.style = style;
  }
  ColourWindow shallowCopy() {
    return new ColourWindow(style);
  }
}
```

14.2 Code for Deep Copying

```
class ColourWindow {
  Style style;
  ColourWindow(Style style) {
    this.style = style;
  }
  ColourWindow deepCopy() {
    return new ColourWindow(style.copy());
  }
}
```

14.2 Pen & Paper

Implement a copy constructor `LinkedList (LinkedList<E> other)` that generates a deep copy of the list.

Fix the method `append(LinkedList<E> other)` using defensive copying before appending it at the end of the current list.

14.2 Deep Copy Constructor

```
public LinkedList(LinkedList<E> other) {
    this(other.getHead().copy());
    if(other.getTail()!=null){
      tail = new LinkedList<E>(other.getTail());
    }
}
```

We need to impose a bound on the type parameter `E` to make sure that it implements a `Copyable` interface that supports the method `copy`.

14.2 Code for Appending with Copying

```
public void appendFix(LinkedList<E> other){
   LinkedList<E> last = this;
   while (last.getTail() != null) { last = last.tail; }
   last.setTail( new LinkedList<E>( other));
}
```

The new method uses the copy constructor.

14.2 Pen & Paper

```
public static void main(String ar[]){
  LinkedList<Integer> list1 = new LinkedList<>(3);
  LinkedList<Integer> list2 = new LinkedList<>(2, list1);
  LinkedList<Integer> list = new LinkedList<>(1, list2);
  System.out.println(list.length());

  LinkedList<Integer> list4 = new LinkedList<>(4);
  list.appendFix(list4);
  System.out.println(list.length());
  list.appendFix(list4);
  System.out.println(list.length());
}
```

Now appending works.

14.2 How To Make LinkedList Immutable

Assume that the list other is immutable and filfills the invariants of a list. Do we need to defensively copy it? How can we make our linked list immutable?

14.3 How To Make LinkedList Immutable

Assume that the list other is immutable and filfills the invariants of a list. Do we need to defensively copy it? How can we make our linked list immutable?

- Remove setter methods for `head` and `tail`
- Make sure that the data element is either immutable or copy it defensively before storing or returning it
- Make `append` return a new list instead of modifying `this`

14.3 How To Make LinkedList Immutable

Assume that the list other is immutable and filfills the invariants of a list. Do we need to defensively copy it? How can we make our linked list immutable?

- Remove setter methods for `head` and `tail`
- Make sure that the data element is either immutable or copy it defensively before storing or returning it
- Make `append` return a new list instead of modifying `this`

Even better:

- Make `head` and `tail` final
- Make the class itself final

14.3 Defining Immutable Classes

- Don't provide any mutating methods — methods that modify fields or objects referred to by fields.
- Make all fields final and private.
- Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final.
- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't modify the mutable objects within your methods.
  - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

14.3 Pros and Cons of Immutable Classes

Pros:

- Invariants are guaranteed to hold for lifetime.
- Immutable objects are inherently thread-safe. No synchronisation is needed.
- Immutable objects are easy to share and a good foundation for object pooling. No defensive copying is needed.
- They are well suited as keys in hashtables.

Cons:

Immutability requires a new object for each distinct value. This makes immutability impractical for large objects that are likely to change often. One way to deal with this problem is to provide a mutable companion class to be used in multistep operations involving modification (e.g. StringBuilder for String).

Use immutability unless you have a strong reason not to!

Part 15

Summary

15.1 Aims of the Course

The techniques taught this week aim to help you design your programs to **make your code more general**—applicable to a wider range of inputs:

- **Type Hierarchies** enable us to write code that works for a range of types. Programming againgst the most general required type is the key to achieving a maximum level of abstraction.
- **Generics** allow us to explicitly abstract from types in aggregates and thereby make aggregates applicable to any reference type or a wide range of them.

15.1 Aims of the Course

The techniques taught this week aim to help you design your programs to **keep your code as simple as possible** and achieve an effective separation of concerns:

- **Languages features**: reduce visibility of features with modifiers, write type-safe code with Generics, use informative exceptions, use nested classes to keep interfaces of enclosing classes simple

- **Common programming idioms**: iterators, delegation instead of inheritance, immutable objects

- **Design patterns**: composite, singleton

15.2 Prefer Delegation to Inheritance

Inheritance Breaks Encapsulation: The correct function of a subclass depends on the implementation details of its superclass. As a consequence, a subclass must evolve in tandem with its superclass.

Inheritance cons:

- Single inheritance in Java
- Tight coupling between the superclass and the subclass
- Design for extension not always possible
- Without design for extension, error-prone use of inheritance

Delegation cons: Implementation with delegation tends to be more verbose due to a larger number of forwarding methods.

15.2 Generics

We learned how to use common Generics features of Java:

- Generic classes
- Generic methods
- Generic parameter bounds
- Wildcards (mnemonic `PECS`):
  - By using an `extends` wildcard we explicitly express our intent to use the aggregate in a read-only fashion.
  - By using a `super` wildcard we explicitly express our intent to use the aggregate in a write-only fashion.
  - If you need both reads and writes, do not use wildcards.

15.2 Co- Contra- and Invariance

Typing rules for methods:

Variable method signature part	Behaviour
Return type	Covariant
Exception type	Covariant
Parameter type	Invariant

Typing rules for constructed types:

Constructed type	Behaviour
Arrays	Covariant
Generic types	Invariant
Wildcard types with extends	Covariant
Wildcard types with super	Contravariant

15.2 Practical 3.8

Introduce an immutable version of binary search trees.

END

I hope you enjoyed the journey!