Software Design

Syntax

- $\alpha|\beta$: either α or β
- $[\alpha]$: α is optonal
- $\{\alpha\}$: α can occout zero or multiple times.
- ... : Further arguments, options, ... are possible.

What is Java (software platform)

- It is a high level, robust, secured and object-oriented programming language.
- Java comes with its own runtime enviroment (JRE) and API. Thus every hardware or software platform that supports this enviroment can run java programs.

What does it consist of?

- **Java Language**: specification of the programming language.
- **Java Virtual Machine (JVM)**: interpreting bytecode.
- **Java Library**: rich collection of standard APIs
  - **Java Standard Edition (SE)**: is the core Java programming platform java.lang, java.io, java.math, java.net, java.util, etc.
  - **Java Enterprise Edition (SE)**: large scale, distributed system built on top of Java SE e.g. libraries for database access, remote method invocation (RMI), web services, XML,...
  - **Java Micro Edition (SE)**: libraries for developing applications for mobile devices and embedded systems.

Features of Java

- Object Oriented Programing (OOP) language.
- Platform independent.
- Interpreted.
- Multithreaded.
- Secured: programs run inside virtual enviroment.
- Automatic Garbage Collection.

Why do we need yet another programming language?

The problem with C/C++/... is that they are designed to be compiled for a specific target.
Eventhough its possible to compile a C++ program for just any type of CPU, to do so requires a full C++ compiler targeted for that CPU.
Problem writing compilers is expensive and time-consuming.
Thus the goal was to create a **platform-independent language** that could be used to produce code that would run on a variety of CPUs under different enviroments.

Types of Java Applications

1. **Standalone Applications**: Desktop/window-based applications.
2. **Web Applications**: applications that run on the server side and create dynamic pages.
3. **Enterprise Applications**: are usually distributed, such as banking applications etc.
4. **Mobile Applications**: applications that are created for mobile devices e.g. Android.

Building a Java Program

Definition 2.1 Compiler: Is a computer program (or set of programs) that translates source code of a high-level programming language, e.g. C++ into a low level language (e.g. assembly language or direct into machine language).

Definition 2.2 Virtual Machine (VM): Is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with the VM.

- **Source code** `file.java`: is first written in plain text files ending with a `.java` extension.
  Requirements
  1. Each source file can contain at most one public class.
  2. If there is a public class, then the class name and file name must match.
- **Bytecode** `file.class`: are created by the **java compiler** `javac.exec` from source code.
  Compiling source code:

```
javac [options] file.java
```

Options

- `-d destination_folder`: compiles file into the give destination folder.

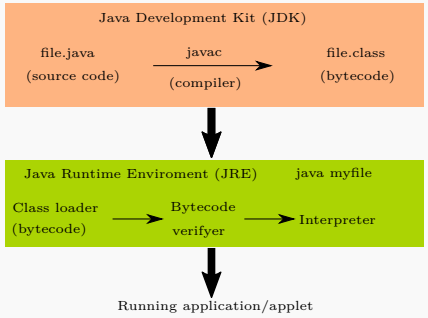
Notes

- Bytecode files are not files that can be read by processor of your platform yet.
- Bytecoes are platform-independent instructions. Thus Java's bytecode is highly portable and can run on any platform containg a JVM that supports the Java version of the bytecode.

Definition 2.3 Java Virtual Machine (JVM)/Interpreter: Is a platform independent runtime enviroment that reads and interprets the the bytecode `file.class` line by line in order to execute java programs.
Its main tasks are: Loading the bytecode, verifying the bytecode, executing the bytecode, garbage collection, thread synchronization,...
Running Java bytecode: `java file`

Definition 2.4 Java Runtime Environment (JRE): **JVM + Libraries**: Provides the libraries, the Java Virtual Machine, and other components to run applets and applications written Java. As the JVM is just an virtual enviroment, the JRE is also known as the implementation of the JVM.
It is the minimum requirement to run (not creating) java programs.

Definition 2.5 Java Development Kit (JDK): **JRE + Development Tools**: It consits of the JRE plus tools such as compilers or debuggers for developing applets and applications.
Thus is necessay in order to develope and running code.



Apllets vs. Applications

All Java programs can be classified as Applications and Applets. The striking differences are that applications contain main() method where as applets do not. One more is, applications can be executed at DOS prompt and applets in a browser. We can say, an applet is an Internet application.

Basics

In Java, every variable, constant, and function (*including main*) must be inside some class.
A Java Program: is a class which contains a main method:

```
public static void main(String[] args) { ... }
```

- `pulic`: accessible from everywhere.
- `static`: defined on the class level (not bound to instances).
- `void`: does not return a result (⇒ procedure)
- `args`: argument, array of command-line arguments

1. Packages

Programmers can define their own packages to bundle group of classes/interfaces, etc.
Packages create a new namespace thus there won't be any name conflicts with similar identifiers from other packages.

Definition 3.1 Package Statement: Identifies the package that a Java program belongs to. The package statement should be the first line in the source file (there can be only one package statement per source file).
Package Statement: `package package-name;`

Compiling source files with package statements

Using the `-d destination_fodler` option will create a folder with the given package name *in the package statement* will be created in the given destination folder (if not existing), and place the complied source file into it.

```
javac -d destination_folder file.java
```

Using Packages

- `package_name.identifier`: use fully qualified names.
- `import package_name.*`: import the whole package.
- `import package_name.identifier`: import certain identifiers.

The default package

- If a program does not include a package statement it belongs to the so called default package, which is basically an default, unnamed package.
  When developing small or temporary applications e.g. for testing purposes, its ok not to include a package statement.
  But in order to avoid name conflicts, all java source files belonging to a program should contain a package statement.
- **Convention**: use your transposed internet domain name for uniqueness, if you have one.
- **Lower Cases**: use lower case letters for packages in orde to avoid any conflicts with the names of classes and interfaces.

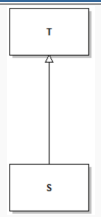
Java

Definition 4.1 Liskov Substitution Principle: If S is a subtype of T, then objects of type T may be replaced with objects of type S **without** altering the correctness of the program

In other words:

- Whenever you work with an instance of type T, you should not be surprised if you effectively work with an instance of type S
- An instance of type S can be used at all places where an instance of type T is expected

Consequences: Overriding methods need to satisfy (at least) the rules specified by the base class



Definition 4.2 Variance: Is a term applied to the expected behavior of subtypes in a class hierarchy containing complex types.

Basics

Definition 5.1 Resolution: Is a rule of inference.

Definition 5.2 Type: defines a behavior but no implementation.
(Java: Types are defined by classes)

Definition 5.3 Subtyping: Are specializations of Types and define a is-a relationship.
(Java: Subtyping is defined by subclassing, i.e. each subclass also defines a subtype)

Definition 5.4 Java Method Signature: Is the method name and the number, type and order of its parameters:

```
methodName(Type1, Type2,...)
```

Note

Return types, name of the arguments and thrown exceptions are not considered to be a part of the method signature.

Definition 5.5 Method Declaration: Is a declaration of a function i.e. declares an identifier and its types,...

```
visibility [static] returnType methodName (args);
```

Definition 5.6 Private: Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
A *subclass* does not inherit the private members of its parent class. Private means you can Only access it in that class an no where else.
However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

Consequence Methods declared private are not inherited at all.

Note: nested classes

A nested class has access to all the private members of its enclosing class—both fields and methods.

Definition 5.7 Public: A class, method, constructor, interface, etc. declared public can be accessed from any class belonging to the Java universe.

Consequence

- Methods declared public in a superclass also must be public in all subclasses

Definition 5.8 Protected: Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

Consequence

- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.

Consequence Only `protected` and `public` methods/variables can be inherited and/or overridden ⟺ Subclasses can only invoke or override `protected` or `public` methods.

Note: methods without access modifiers

Can be overridden, if the superclass is in the same package.

Definition 5.9 `final` variables: A final variable can only be assigned once (final assignment) and will always have the same reference hence:

```
final var1 = 5; // or
final var2;
var = 5;
```

- If a final variable is assigned to a primitive, then the value of the variable will be constant.
- If a final variable holds a reference to an object, then the state of the object may still change, but the viable will always refer to the same object = **non-transitivity**

Note

Non-transitivity also applies to arrays, as arrays are mutable i.e. we can change the elements of the array.

Definition 5.10 `final` **methods:** A final method cannot be overridden or hidden by subclasses This is used to prevent unexpected behavior from a subclass e.g. altering a method that may be crucial to the function or consistency of the class.

Definition 5.11 `final` **Classes:** A final class cannot be subclassed/extended.

Note

Doing this can confer security and efficiency benefits thus many of the Java standard library classes are final.

1. Mutable Types

Definition 5.12 Mutable Types: Are types that can be "mutated"/changed/modified, while maintaining its identity/reference
Arrays are mutable ⇒ we may not change its reference **but** its contents!
Examples:
- Lists
- Every class whose members can be altered (⇒ a lot)

Mutable types&security

If we simply return a mutable type, clients may alter e.g. signers (cryptographically signed objects) of objects ⇒ Return copies by:
- Return deep copies using clone:
```
public Rectangle getBounds(){ return (Rectangle)
↪  bounds.clone();}
```
- Use copy constructor to return new instance of given type:
```
public Object[] getSigners(){ return new Signers(this) }
```

Notes

- Deepcopy and copyconstructor Need to be implemented!
- If we do not make a deep copy, mutable types may still be altered even if they are private and do not get returned!

```
public class RectangleFigure implements Figure {
   private Rectangle bounds;
   public Rectangle getBounds() { return (Rectangle)bounds.clone(); }

   public void move(int dx, int dy) { // controlled state change
      bounds.translate(dx, dy);
      notifyChanges(new FigureChangedEvent(this));
   }

   public void setBounds(Rectangle r) { // controlled state change
      this.bounds = r;
      notifyChanges(new FigureChangedEvent(this));
   }
   ...
}
```

Question:
Is an uncontrolled change of the bounds still possible?
(uncontrolled = change without notification)

Figure 1: Uncontrolled changed if clone does not deep copy

2. Immutable Types

Definition 5.13 Immutable Types: Are types are types whose objects states cannot be modified after creation (they can only be in one state).
Examples: (Blue are not `final`)
- java.lang.String
- java.awt.Color
- BigInteger, BigDecimal, MathContext,
- Character, Boolean, Byte, Double, Float, Integer, Long, Short, Enum
- URI, URL
- InetAddress, Inet4Address, Inet6Address
- File
- UUID
- Pattern

Benefits

- **Consistency**: Immutables can freely be shared.
  E.g. *mutables* define their hash code *depending* on their state
  ⇒ problem when using mutables a key for hash maps/sets:
```
HashSet<Date> set = HashSet<Date>();
Date key = new Date();  // Problem this is a mutable
set.add(key);
// Modifying key e.g.key.setTime(...)
// set.contains(key) => false
```
  Thus: immutables are the better map keys
- **Thread Safety**: Immutables can be freely cached, as they cannot be changed.
  ⇒ for parallel code we may obtain race conditions when using mutables
- **Safe in the presence of ill-behaved code**: if we pass parameters to methods, they should not change their state, but this is an act of face.

False sense of security

Sometimes mutable classes that work correctly can give us an false sense of security, especially if the declare private fields. All of a sudden they can start to work incorrect see:
- Complex Class
- Fraction Class

Listening 5.1 : *Complex* *java*

```
Complex1 extends Complex{
  public void square(){
    multiply(this);
  }
}
```

Problem: even though we used temporaries `re` and `im` we still get an incorrect value as we change the value of the immutable *y* if we pass the special `this` reference to multiply!

Listening 5.2 : *Complex* *java*

```
Complex1 extends Complex{
  public void square(){
    multiply(this);
  }
}
```

Problem: even though we used temporaries `re` and `im` we still get an incorrect value as we change the value of the immutable *y* if we pass the special `this` reference to multiply!

Note

Again seems to work correct but we will have a problem if y is this.

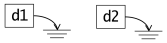
Writing Immutable Classes

- All non-private fields have to be declared `final` and have to be of primitive or immutable type.
- No setter methods (that is any method that can modify members)
- Class has to be declared `final` so that subclasses cannot declare mutable fields.
- Do not inherit from a base class which has non-static mutable fields
- References returned by getter methods:
  - have to be of primitive or immutable type or
  - have to be deeply cloned e.g. lists
```
public ArrayList<Double> getPrimitiveList() {
  return (ArrayList<Double>) this.mylist.clone();
}
```
- References passed to the object with a constructor, again
  - have to be of primitive or immutable type or
  - have to be deeply cloned e.g. lists
- The `this` reference is not allowed to escape during construction!

Definition 5.14 Primitive Types: Are immutable as there value cannot be changed and keep there identity:
```
int i = 1;
    i = 2;
```
We change the reference, but we cannot change the internals of 1, 1 will always stay 1.
Examples: int, long, short, double, float, boolean, char, byte.

Definition 5.15 Variable/Reference Declaration: In java the only way to access an object is through a reference variable.
```
StaticType reference;
```
A reference is not an object, thus no memory is allocated for an object of the Type `StaticType`. of the reference.

```
Date d1, d2;
```



Note

- Do not confuse C++ references with java references in Java variables are called (by convention) references.
- Reference variables are sort of C++ pointers but we can not do pointer arithmetic's on it e.g. `ptr++`
- Reference Variables are of size:
  - 32-bit on a 32-bit JVM
  - 32-bit or 64-bit on a 64-bit JVM, depending on the configuration

Definition 5.16 Static Type: A reference variable is declared to be of a specific type and that type, known as static type can never be changed.
Static type = type of reference variable

Corollary 5.1 Guarantees of Static Type: When a variable is declared as being of a particular type, then we have a language-enforced guarantee that any *object* referenced by that reference variable will have (at least) all the features of that *type*.
⇒ *dynamic type* needs to provide methods of *static type*.

Definition 5.17 Instanciation `new`**:** The new operator instantiates a class by dynamically allocating memory (=allocation at run time on the heap) for a new object and returns a reference to that memory.
```
new MyClass();
```
This reference can then be stored in/assigned to a (reference [def. 7.15]) variable.
```
Type ref = new MyClass();
```

Note

In Java, all class objects must be dynamically allocated.

Object Oriented Programming (OOP)

1. Principles

Definition 5.18 Encapsulation:
- Methods and data are combined in classes
- Not unique to OOP

2. Class Basics

1. Static

Definition 5.19 Static Fields: Per class fields that exist only per class (≠ per object), accessible via class (or instance)
- Bound to class (available even if no instance has been created)
- Only one copy of the attribute (identical for all instances)
- Initialization per default with zero (0 / 0.0 / false / null)
```
static Type ref;
```

Constants Declare constants as `static final` fields.

Definition 5.20 Static Methods: Per Class methods, accessible via class (or instance):
```
static ResultType Name(args){ Body }
```
- No `this`
- No access to instance attributes & methods

Do not use instances to access static fields or methods.

2. Class Initialization

Upon loading o the class

3. Inheritance

Definition 5.21 (Implementation Aspect)
Code Inheritance/Extension/Subclassing:
- Subclasses add new fields & methods
- Subclasses have more (specialized) attributes & operations
- Implementation aspect

Class: defines a type **and** implementation

Definition 5.22 (Design Aspect)
Interface Inheritance/Specialization/Subtyping:
- Types define a set of objects
- Subtypes are specializations of this Type
- Inheritance of behavioral aspects

Interface: defines type

Definition 5.23 `extends`**:**
Define new class by extending existing classes.
In Java only one base class can be specified to inherit from. Extension inherits all fields and methods from the base class.

1. Polymorphism

Definition 5.24 Type Checking: The process of verifying and enforcing the constraints of types

Definition 5.25 Statically Typed Languages: Is a language where the type of a variable is known at compile time. For some languages this means that we as programmer must specify of what type each variable is.
Advantage: can do type-checking during compile time by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.
Examples: C, C++, Java

Definition 5.26 Dynamically Typed Languages: If the type is associated with/depends on run-time **values**, and not named variables/fields/etc
Advantage: we do not have to specify types every time.
Drawback: need to do type checking at run-time

Definition 5.27 Polymorphism: Polymorphism allows operations to be performed on objects without needing to know which class the object belongs to, provided that we can guarantee that the class implements the specified type.

Definition 5.28 Polymorphic Assignment: Instances of an extended class can be assigned to references of type base class:
```
BaseClass ref = new extendedClass();
```

Definition 5.29 Dynamic Type: Is the type of the object assigned to a reference variable.
Dynamic types of reference variables may change with every assignment.
Dynamic Type ⊆ Static Type as the dynamic type must full fill at least the guarantees of the static type.

Definition 5.30 Static Binding: Is type resolution based on the static type of a variable reference.

Definition 5.31 Static Binding and Overloading: If we overload a method in Java, the compiler will produce a version for each overloaded function signature.
The resolution of the method signature (not to the actual implementation) of the method is done during compile time and does hence depend on the static type passed to the method ⇒ static binding
```
StaticTypeOfa a = new DynamicTypeOfa();
a.methodToResolveTo(StaticTypeOfb b);
```
`staticTypeOfb` decides which overloaded method to call.

Definition 5.32
Dynamic Type Binding and Overloading: The runtime chooses a function implementation
1. Based on the function signature chosen at compile time (static binding)
2. Depending on the dynamic type of the object referenced by a in order to chose an actual implementation of

```
DynamicTypeOfa.methodToResolveTo(StaticTypeOfb b)
```

Note

For non-overloaded methods only the dynamic type of the reference variable decides which method to call.

Static Class methods

Cannot be overriden because they are not dispatched on the object instance at runtime. The compiler decides which method gets called.
Thus even if we overwrite them (and the compiler does not complain), it will always be called the method of the static type and not the dynamic.

Listening 5.1 : *Dynamic Type Inference*

```
if( ref instanceof Type)
```

Note

A runtime error is thrown if the dynamic type of `ref` is not a Type or extension thereof. (Type) ref;

Definition 5.33 Covariant Typing: Covariant return, means that when one overrides a method, the return type of the overriding method is allowed to be a subtype of the overridden method's return type.
This allows to narrow down return type of an overridden method without any need to cast the type or check the return type.
Thus: Co-variant return types follow the Liskov substitution principle.

Note: Invariant Types

Before Java 5.0 it was not possible change the return type of an overriden method, the return type was said to be invariant.

Listening 5.2 : *Covariant Example*

```
public class Animal {
    protected Food seekFood() {
        return new Food();
    }
}

public class Dog extends Animal {
  @Override
  protected Food seekFood() {
      // covariant return type
      return new DogFood();
  }
}
```

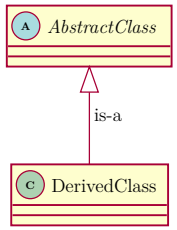
4. Abstract Classes

Definition 5.34 Abstract Methods: Are methods that only define a signature/are only a declaration:

```
visibility abstract retrunType methodName();
```

- Define methods to be implemented in subclasses
- Can only be declared in `abstract` classes
- Cannot be `private` as private methods cannot be overridden
- Cannot be `static` as static methods cannot be overridden

Definition 5.35 Abstract Classes: Are classes that stand in an "is-a" relationship with their subclasses:



- Cannot be instantiated
- Usually have one or more abstract methods
- May have attributes, constructors, non-abstract methods

```
visibility abstract class Name{ Body }
```

Note

An abstract class must not necessarily have an abstract method.
Useful to declare classes abstract that cannot/may not be initiated but are supposed to be extended.

Note: Derived Classes

- Have to override (implement) all `abstract` methods
- Or have to be declared abstract as well, if not all abstract methods are overridden.
- Can only extend one Class.

Note

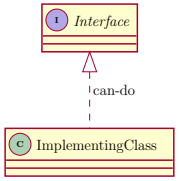
Arrays of an abstract base type may be instantiated sine no instances are created (only reference variables)

```
ContainerType[] name = new ContainerType[Number]
```

Usage
- Want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

5. Interfaces

Definition 5.36 Interfaces=pure abstract class: Have to be implemented by the class that uses the interface and represent a "can-do" relationship:



- Have only `public` and `abstract` methods
- Attributes are by default `public`,`static` and `final`
- No constructors ⇒ instantiated (≠ declared)

Definition:

```
public intreface ClassName{ Body; }
```

Implementation:

```
class ClassName implements IntefaceName{ Body; }
```

- All methods defined in the declared interface have to be implemented unless its another interface adding more functionality
- A class may implement multiple interfaces

Note: Java 8 default methods

Can be implemented inside interfaces and are able to access other methods. Allows to extend interfaces without breaking existing classes that implement the interface.

Use `@override` to implement the methods.

Usage
- You expect that unrelated classes would implement a piece of functionality.
- Want to specify the behavior of a particular data type, but not concerned about who implements its behavior.

Interfaces as function arguments

Methods with interfaces as function argument can be called with any class that implements the interface.

Interface Reference

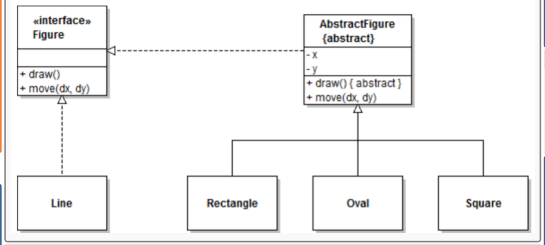
Interfaces can be used as reference variable for all subclasses **but** be care-full if the reference object implements another interface we will not be able to call its functions unless we use an implicit cast.
Check if referenced object implements an interface: `ref instanceof aInterface`

Listening 5.3 : *Interface Variable*

```
TypeImplementingAandB ref = new TypeImplementingAandB()
InterfaceTypeA refA = ref;
InterfaceTypeB refB = ref;
refA instanceof InterfaceTypeB // True as ref implements B
refA.methodOfTypeImplementingAandB() // works
refA.methodOfB() // does not work
((InterfaceTypeB)refA).methodOfB() // works
```

Intrefaces and Abstract Classes

- `Interface`: provides a type
- `abstract Class` semi finished component that contains default implementations
  - which can be used in subclasses
  - which can be overriden in subclasses



Abstract Classes vs. Interfaces

- Abstract Classes
  - A class may extend only one abstract class
  - Abstract classes may contain attributes & concrete implementations
- Interfaces
  - A class may implement several interfaces
  - Interfaces may contain no implementations

Definition 5.37 Anonymous Class: Is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.
The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

```
interface/class MyClass{
  // definition
}

// Anonymous object with extras
MyClass obj = new MyClass()
{
  // data members and methods
  @Override
  public my_func typ(typ)
  {
    ...
  }

  public new_func typ(typ)
  {
    ...
  }
};
```

Definition 5.38 Marker Interface: Is an empty interface, that can be used to add a certain attribute/characteristic to a class that can be checked with `instanceof` e.g. RandomAcess

Definition 5.39
Functional Interface `@FunctionalInterface`: Is an interface with a single abstract method:

```
@FunctionalInterface
public interface InterfaceName{
  visibility returnType methodName(args);
}
```

Notes
- The annotation `@FunctionalInterface` allows compilers to generate an error if the interface does not satisfy the conditions of a functional interface.
- Default methods are not abstract and do not count.

Definition 5.40 Lambda Expressions: Is a block of code that you can pass around so it can be executed later.

```
[var = ](Typ1 arg1, Typ2 arg2,...) -> single expressions;
// or
[var = ](args) -> {expressions}
```

You never specify the result type of a lambda expression. It is always inferred from context.

Note
- If the argument types the lambda expression can be inferred, we can omit them.
- For no arguments we still need to write empty parentheses ().
- If a method has a single parameter with inferred type, you can even omit the round parentheses:

```
ActionListener listener = event ->
System.out.println("The time is " + new Date());
```

Definition 5.41
Functional Interfaces Lambda Expressions:
We can use Lambda expressions to define the anonymous functions at a point.
This is really convenient as we do not have to define a subclass or anonymous class but can simply supply a lambda expression whenever an object of an interface with a single abstract method is expected.

```
subj_obj.addActionListener((ActionEvent e) -> {
  ...
  });
  // or even just
subj_obj.addActionListener(e -> {
  ...
  });
```

Note: the provided lambda function is than basically an observer (section 1) getting triggered when the subject notifies this function.

Testing

Definition 5.42 Unit Test: Is a level of software testing where individual units/ components of a software are tested i.e. we do not run `main` but test individual pieces of software.

Is a framework for running unit tests

```
import org.junit.Test
import static org.junit.Assert.*;

public class MyTest{

  @Test
  public void testName1(){

  }
  @Test
  public void testName2(){

  }
}
```

Figure 2: Basic JunitTest

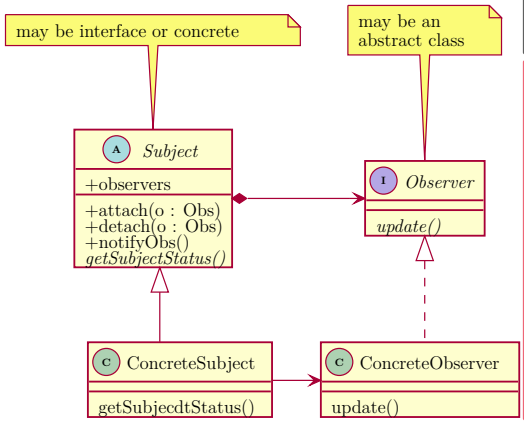
Test Class Annotations

- `@Test`: indicates that a method contains unit tests
  ```
  @Test(expected=IllegalException)
  ```
- `@Before`/`@After`: mark methods that are called before and after each unit test.
  Note: can be used to set up and tear down thins needed for all tests.
- `@BeforeClass`/`@BeforeClass`: similar to `@Before` and `@After` but only run once per class.

Behavioral Patters

Definition 6.1 Behavioral Patterns: These design patterns are specifically concerned with communication between objects.

1. Observer/Listener/Publish-Subscriber



Intent
- Consistency assurance between cooperating objects without connecting them too much.
- One-to-many relation between objects which allows to inform the dependent objects about state changes

Listening 6.1 : *Observer*
```
interface Observer{
  void update ();
}
```

Listening 6.2 : *Subject*
```
class Subject {
  private List <Observer > observers = new ArrayList <Observer >();

  public void addObserver(Observer o) {
    // method of List
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    // method of List
    observers.remove(o);
  }

  protected void notifyObservers () {
    // inform all Observers that we have changed
    for(Observer obs : observers) {
      // Let (concrete) observers handle it
      obs.update ();
    }
  }
}
```

Listening 6.3 : *Concrete Subject*
```
class ConcSubject extends Subject {
  private int state;
  public int getState (){
    return state;
  }

  public void setState(int val){
    state = val;
    notifyObservers();
  }
}
```

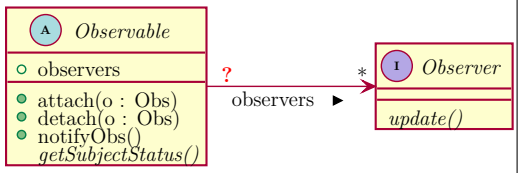
Listening 6.4 : *Concrete Observer*
```
class ConcObserver implements Observer {
  private ConcSubject s;
  ConcObserver (ConcSubject s){
    this.s = s;
    s.addObserver(this);
  }

  public void update (){
    // take appropriate steps
  }
}
```

Participants
- **Observer**: typically an interface that declares methods to handle notifications.
- **Subject**: maybe concrete or interface knows its observers over the observer interface (guarantee that the method update exists).
- **Concrete Observer**:
  - Handles state changes of its observable
  - Implements the observer interface to keep its state consistent with the subject
  - May maintain a reference to a concrete subject (pull-model)
- **Concrete Subject**: Implements/extends Subject and hence can:
  - attach observers
  - detach observers
  - call the method notify if its state changes (e.g. in a set method)

Consequence
- Subject does not care about the number of observers ⇒ notifications are broadcast.
- A simple state change/operation may lead to a (unwanted) cascade of updates.
  → beware of cyclic dependencies see

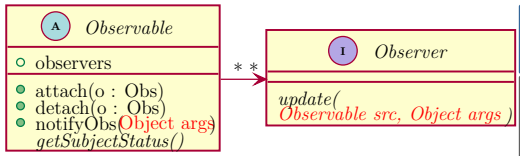
Note
- Thus it is up to the observer to handle or **ignore** notifications.
- When should `notifyObservers` be called?
  - **Automatically**: at every state change
    ⇒ many updates
  - **Explicitly**: after a set of changes
    ⇒ responsibility e.g.
    - Asynchronous at idle times
    - Use a boolean flag `changed`, that can be set in order to test if object has changed see section 3

1. One or Many Observables



0...1 : *
- Subject/Observable is usually passed to constructor of the Concrete Observer (⇒ reference).

*** : ***
- If we want that observers can observe multiple subjects and if these observers may be able to ignore certain events from those subjects, then the observer need to know the source of an update event.
  Moreover if arguments are passed to the Observables, then the observer needs to know:
  - The arguments
  - The source of the event



Listening 6.5 : *Many vs Many*
```
protected void notifyObservers(Object args){
    for(Observer obs : observers){
        obs.update(this, args);
    }
}
```

2. Push vs Pull Model

Push-Model
- The subject (i.e. the Observable) sends the observer on notification all the data it will need. The observer doesn't need to query the subject for information.
- Observable must know which information is needed by the observers in order for them to act appropriately (or Observable sends all data)

Pull-Model
- In the pull model, the subject merely notifies the observer that something happened, and the observer queries the subject based on that, to get the information it needs.
- Observable send only information about what has changed
- Observer has to access the Observable state (via reference)

Pitfalls

3. Java.util.Observable

Provides an implementation of an Observable that can be used.
But it is never used!
Problem: multiple inheritance is in java not possible, thus `Java.util.Observable` can only be implemented if extension is not defined in its own inheritance hierarchy.
Solution: composite twin class (**??**).
⇒ code duplication ⇒ `Java.util.Observable` is not used.

Listening 6.6 : *Java.util.Observable*
```
public interface Observer {
  void update(Observable o, Object arg);
}

public class Observable {
  private boolean changed = false;

  // Marks this Observable as having been changed
  protected void setChanged() { changed = true; }

  // Indicates that this object has no longer changed,
  // or that it has already notified all of its observers
  // of its most recent change.
  protected void clearChanged() { changed = false; }

  // Tests if this object has changed.
  public boolean hasChanged() { return changed; }

  // If this object has changed, as indicated by the
  // hasChanged method, then notify all of its observers
  // and then call the clearChanged method to indicate
  // that this object has no longer changed.
  public void notifyObservers(Object arg) {
    if (!changed) return;
    // See section on ConcurrentModificationException
    Object[] copy = obs.toArray();
    clearChanged(); // before notification!
    for (int i = copy.length-1; i>=0; i--)
      ((Observer)copy[i]).update(this, arg);
  }
}
```

4. ConcurrentModificationException

Definition 6.2 ConcurrentModificationException: We cannot change a list e.g. add/remove while iterating over it.
Example: Once Observer and notify.

Definition 6.3 OnceObserver: Is an observer that only observers for a single notification.
⇒ removes itself afterwards.

Listening 6.7 : *OnceObserver*
```
public class OnceObserver implements Observer {
                                Conc. Observerable
  public void update(Observable source, Object arg) {
        // take appropriate steps once
        source.removeObserver(this);
  }
}
```

Listening 6.1 : *Problem* *java*
```
protected void notifyObservers(Object arg){
  for(Observer obs : observers){
    obs.update(this, arg);
  }
}
```
Problem: we cannot remove an observer from obs while iterating over it

Solutions
1. Perform the notifications on a copy of the observer list.
2. Delay add/remove Observer calls.
3. Copy the observer list upon modification.

Definition 6.4 Java list to array: Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`java.util.ArrayList.toArray(a T[])`
a: is the array into which the elements of the list are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.

Listening 6.8 : *Copy Solution*
```
protected void notifyObservers(){
  Observer[] copy;
  copy = observers.toArray(new Observer[observers.size()]);
  for (Observer obs : copy){
    obs.update(this);
  }
}
```

Listening 6.9 : *Alternative*
```
protected void notifyObservers(){
  for(Observer o : new ArrayList<>(observables)){
    o.update(this);
  }
}
```

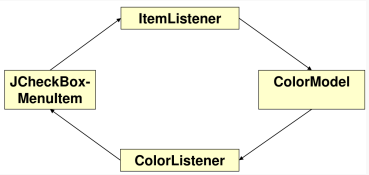
Definition 6.5 Copy on write list: Thread-safe variant of ArrayList in which all mutative operations are implemented by making a fresh copy of the underlying array.

Listening 6.10 : *Best Solution*
```
private List<Observer> observers = new CopyOnWriteArrayList<>();
public void addObserver(Observer o) { observers.add(o); }
public void removeObserver(Observer o) { observers.remove(o); }

protected void notifyObservers(Object arg) {
  for(Observer obs : observers) {
    obs.update(this, arg);
  }
}
```

5. Cyclic dependencies

Problem

If we have a cyclic dependency we will end up in an endless loop of notifying and updating, unless we take appropriate measures.



General Rule: Propagate changes only if the model really changed!

Solution 1: Color Listener

Break recursion inside the listener: e.g. we check inside the color listener if the color really changed, or is still the same ⇒ recursion or nothing happens anyway.

```
// update method of listener
public void colorValueChanged(Color c){
  if (isSelected() OR_OPRATOR !c.equals(this.color))
  // If the color really changed and isSelected is true
  // select the checkbox menu item
    setSelected(c.equals(this.color));
}
```

Note: not recommended to check inside the listener, as we may have many different kinds of listener.

Solution 2: ItemListener flag

Same as Solution 1 (but now for the ItemListener) but we use a flag instead of checking the actual value and:

```
private boolean updating = false;
public void itemstateChanged(ItemEvent e){
  if (!updating){
    updating = true;
    if(e.getStateChanged() == ItemEvent.SELECTED)
        model.setColor(color);
  }
  updating = false;
}
```

Note: not recommended to check inside the listener, as we may have many different kinds of listener.

Solution 3: Concrete Model

Break recursion inside the Model: e.g. we check inside a subclass of JCheckBoxMenueItem if the performed action has already been performed and if not we can call the super method:

```
public void setSelected(boolean b){
  if (b != isSelected())
    super.setSelected(b);
}
```

Note: recommended as we usually have one model and lots of different listeners.

Solution 4: Color Model

```
public class ColorModel{
  private Color color;
  public void setColor(Color color){
    if (c.equals(this.color)){
      this.color = color;
      notify(color);
    }
  }
}
```

Note: recommended as we usually have one model and lots of different listeners.

Note

isSelected() OR_OPRATOR !c.equals(this.color) can be written as isSelected() != c.equals(this.color)

Note: Solution 5

Use ActionListener instead of an ItemListener: Actions are only triggered by keyboard or mouse events.

6. Causality of Changes

Problem

How can we make sure that notifications of the model will be handled by the observers in the same order as they were applied to the models.

Solution

- Queuing of state changes: model becomes asynchronous... difficult to program
- Queuing of notifications
- Prohibit state changes during notification

7. Memory Management

Is automatic in java if no more reference to an object exists. ⇒ need to detach all observers/listeners for clean up.

2. Actions

Definition 6.6 Action: Is the measure to take onto and actionEvent.

1. ActionListener and performed actions

Definition 6.7 Adapter Class:
Java adapter classes provide the default implementation of ActionListener interfaces and implement the `public abstract void actionPerformed(ActionEvent e)` method.

There exist multiple ways to specify how an action Listener should react to an an (action) event.
E.g. java.awt.event.ActionListener [def. 7.39] is a functional interface with a single abstract method
`public abstract void actionPerformed(ActionEvent e);`
which defines what action to take onto an event.
In order to write an Action Listener, we can:

1. **Use the subject directly**: the subject must implement the ActionListener and hence implement the method actionPerdormed or
2. **We use an Adapter Class** ([def. 8.7]): that implements the action Listerner Interface and register it inside the subject or
3. We use anonymous classes of the ActionListener and register it inside the subject.
4. **We use lambda expressions** ([def. 7.41]): of the ActionListener and register it inside the subject.

Direct Implementation

```
public class Subject implements ActionListener {
  ...
  public void actionPerformed(ActionEvent e) {
    if(e.getSource() == push) handlePush();
    else if(e.getSource() == pop) handlePop();
  }
```

Adapter Implementation

```
class PushListener implements ActionListener {
  public void actionPerformed(ActionEvent e) { ... }
}
subj_obj.addActionListener(instanceOfMyClass);
```

Anonymous Implementation

```
subj_obj.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {...}
  });
```

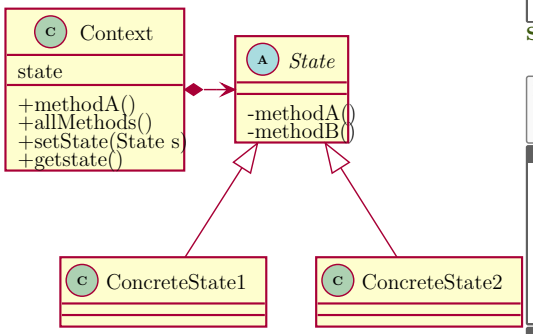
Lambda Implementation

```
subj_obj.addActionListener((ActionEvent e) -> {
  ...
  });
  // or even just
subj_obj.addActionListener(e -> {
  ...
  });
```

Note: multi-method listeners

Anonymous interfaces still need to be used for implementing multi-method interfaces.

3. State Patterns



Intent Context related behavior depending on the current state

Participants

- **Context**: is the concrete thing (e.g. Vending Machine, Ventilator) that has a reference/stores the concrete state and acts depending on it by calling the `request` or `doAction` method of its current state reference.
- **Abstract/Interface State**: is
  - Is an interface that will be implemented by the *concrete states* and defines all the possible actions that each *concrete state* may be able to perform or
  - An abstract base class if
    - we want to define default behaviors of the shared methods:
      ```
      privat void doAction1(args){
        throw new IllegalStateException();
      }
      ```
    - the concrete states store shared similar attributes/methods that we do not want to define inside the context.
- **Concrete State**: Implements the methods of the *state interface*

Note

- Contexts may hold multiple different states e.g. light state as well as temperature state.
- The possible actions declared inside the state interface/abstract class may take a context object as argument in order to change the current state, if they are not defined inside the context (see State changes section).
- Interfaces allow only public methods before Java9, may be another reason for using abstract classes.

Listening 6.11 : *Context*

```
class Context{
  State state;

  public void setState(State s){
    this.state = s;
  }

  public State getState(){
    return state;
  }

  public doAction1(){
    state.doAction1(this);
  }

}
```

Listening 6.12 : *State*

```
abstract class State {
  privat void doAction1(Context cur,args){
    throw new IllegalStateException();
  }
}
```

State Changes

1. Decentralized

May be initiated by the concrete state method ⇒

- Concrete state object needs a reference to the *context* in order to set its state:

Listening 6.13 : *Context*

```
class Context{
  State state;
  ...
  public doActions(){
    state.doAction1(this);
    state.doAction2(this);
  }

}
```

Listening 6.14 : *State*

```
interface State {
  public retType doAction1(Context cur,args);
  public retType doAction2(Context cur,args);
}
```

Listening 6.15 : *Concrete State*

```
concStateA implementes State {
  public state doAction1(Context cur, args);
      // do somthing
      cur.setState(new StateB());
}
```

- **Or** the state returns the new state when running an action which gets then set by the context

Listening 6.16 : *Context*

```
class Context{
  State s;
  ...
  public doActions(){
    s = state.doAction1();
    s = state.doAction2();
  }
}
```

- **Or** the state and Concrete States are defined inside context class, thus concrete states can directly set state of the context:

Listening 6.17 : *Concrete State*

```
class Context{
  State s;
  ...
  concStateA implementes State {
    public state doAction1(args);
      // do somthing
      s = setState(new StateB());
  }
  ...
}
```

2. Centralized

Transitions are initiated by the context, state should be informed that it gets activated or deactivated.

Listening 6.18 : *DrawTool (State)*

```
public interface DrawTool{
  void activate();
  void deactivate();
}
```

Listening 6.19 : *DrawView (Context)*

```
DrawTool tool; // current state

public void setTool(DrawTool tool){ // set conc state;
  if(tool == null) throw new IllegalArgumentException();
  if(this.tool != null) this.tool.deactivate();
  this.tool = tool;
  this.tool.activate();
}
```

Note

If the initial state is set in the constructor of the context we can omit: `if(this.tool != null) this.tool.deactivate();`

Creation of State Objects

3. States are created on demand

```
public void stateMethod1(Context cur){
  cur.setState(new StateB());
}
```

4. States are stored

States are stored in variable or list that is accessible by all concrete states ⇒ thus states are for example stored inside context.

Context

```
class Context{
  State s;
  ...
  private final State CONC_STATE_A = new CStateA();
  private final State CONC_STATE_B = new CStateB();
  private final State INIT = new InitState();
}
```

Abstract State Class

```
public void stateMethod1(Context cur){
  cur.setState(c.INIT);
}
```

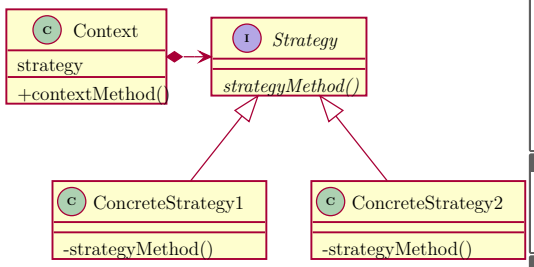
Abstract State Class

```
public void stateMethod2(Context cur){
  throw new IllegalStateException();
}
```

Examples

- DrawTool (drawing state)
- DrawGrid (constrain the mouse coordinates)
- Handles (CTRL/SHIFT pressed)

4. Strategy Patterns



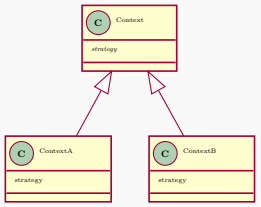
Intent

- Allow to change algorithms independently of the clients
- Support extensibility with new algorithms
- Make algorithms interchangeable

Which problem does it solve

Strategy pattern is an alternative to specialization of context class which:

- violates the Single Responsibility Principle
- prevents interchanging the strategy of a context dynamically, as subclassing statically defines the binding between context and strategy



Participants

- **Context**: is the real thing e.g.
  - A secure channel
  - Container
  - TransportationToAirport

  that uses a concrete strategy.
  Thus it must have a reference/stores an instance of the concrete strategy.
  The name of the context method(s) depends on the context e.g.
  - encrypt, decrypt
  - addLayoutComponent, minimumLayoutSize
  - transport

  and acts depending on it by calling the `request` or `doAction` method of its current state reference.
- **Strategy/Algorithm Interface**: is usually an interface as we implement some functionality and do not have an "is-a" relationship.
  The strategy interface specifies which methods the algorithms may be able to perform.
- **Concrete Algorithm**: Implements the methods of the *Strategy interface*

Note

- Strategy methods may contain a reference to the context as a parameter, in order to use a state full algorithm see section 1.
- Strategy Interface may be defined inside the context class.

Listening 6.20 : *Context*

```
class Context{
  private Strategy algorithm;

  public void setAlgorithm(Strategy a){
    if(a == 0){ // See also Null Object Pattern
    ↪  |\cref{subsubsec:NullObjectPattern}|
      throw new IllegalArgumentException();
    }
    this.algorithm = a;
  }

  public State contextMethod1(){
    algorithm.doAction1()
  }

  public State contextMethod1(){
    algorithm.doAction1()
    algorithm.doAction2([this],args)
    algorithm.doAction1()
  }

}
```

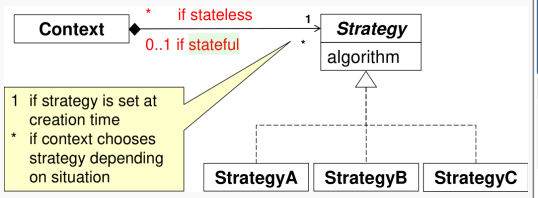
Listening 6.21 : *Strategy Interface*

```
interface class Strategy {
  privat void doAction1();
  privat void doAction2([Context cur],args);
}
```

Listening 6.22 : *Concrete Strategy*

```
class Algorithm1 implements Strategy {
  privat void doAction1(){...};
  privat void doAction2([Context cur],args){...};
}
```

1. Statefull vs Stateless Strategys



- **Stateless** Strategy: does not store/have any state/context specific parameters ⇒ strategy can be used from different contexts.
  Note: in order to use parameters, strategy may access state specific parameters of *context* (but not itself!).
- **Statefull** Strategy: may store/have any state/context specific parameters ⇒ cannot be exchanged between different contexts.

Strategy vs. State

Strategy	State
• Strategy is typically not aware of other concrete strategies	• A concrete state may be aware of others ⇒ transitions.
• Typically set only once	• Defines state-specific behavior
• Usually only one *public* method	• Typically state changes at runtime
• Strategy may contain algorithm specific state	• State usually contains no state but accesses state in context

5. Null Object Pattern

IntentMotivation: We do not want to litter the whole code with statements of the form `if(strategy!=null)` in order to check if a strategy is set.

- Encapsulate the absence of an object/a strategy by providing a substitutable alternative that offers suitable default "*do noting*" behavior
  - ◆ empty methods {}
  - ◆ no strategy set error

Listening 6.23 : *-NoValue-*

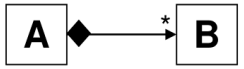
```
public class NullLayout implements LayoutManager {
  public void addLayoutComponent(String name, Component comp) { }
  public void removeLayoutComponent(Component comp) { }

  public Dimension minimumLayoutSize(Container parent) {
    return parent.getSize();
  }
  public Dimension preferredLayoutSize(Container parent) {
    return parent.getSize();
  }
}
```

6. Composite Pattern

Definition 6.8 Composite:

B is a fixed part of A that is:

- B may be part of only one A
- If A is deleted, then also all its part
- Whole acts substitutions for all its parts
  ⇒ operations are propagate to its parts



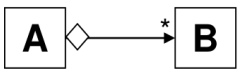
Example

House (parent) and a room (child), if we delete the house the room will be gone as well.

Definition 6.9 Aggregate:

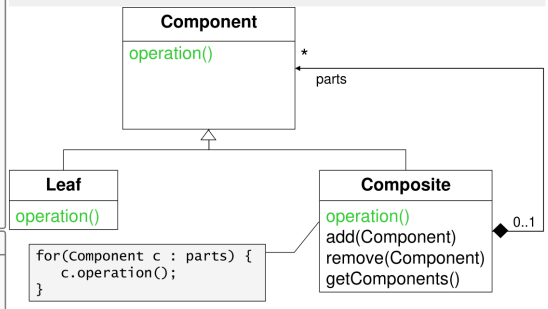
B is a variable part of A that is:

- B may be part of several A's
- B may exist individually



Example

Class (parent) and Student (child), if we delete the class the students wills still exists, plus a student can be part of multiple classes.



IntentRepresent single/primitive components as hierarchical group such that we can treat the group object/composite the same way as the single objects.
Examples of operations applied uniformly on composites and single objects:

- Jdraw: move, draw, resize, copy
- File sizes: getSize, setProperties, delet

Paticipants

- **Abstract Class/Interface Component**: declares the base class and uniform methods implemented by the leaf primitives
  E.g. Figure, Graphic, Shape, Room, ArithmeticOperation
- **Leaves/primitives**: implements the Component interface/base class and specifies the behavior of the concrete primitive types
  E.g. Rectangle, Text, Circle, PlayRoom, Multiply
- **Composite**: implements the Component interface/base class and
  - ◆ Stores references to children which may be
    - Leaves/Primitives
    - Again composites
  - ◆ Implements methods to manage children add, remove, get
  - ◆ Defines behavior of components containing children and must be concurrent with behavior of single elements

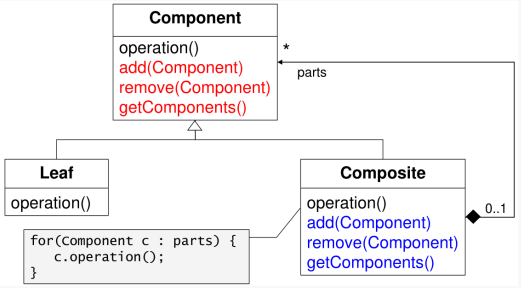
  E.g. FigureGroup, GraphicsComposite, House, ArithmeticExpression
- Client: manipulates objects through the Component interface

Transparent vs. Safe

Question

Where should we declare the methods for managing the children, add, remove,...?

- **Transparent Approach**: declare them inside the Component interface
- **Safe Approach**: declare them only inside the Composite



1. Safe Approach (Composite)

Clients cannot do stupid/meaningless things s.a. adding, removing,... from primitives/leaf objects because they do not even implement those methods ⇒ compile time warning.

2. Transparent Approach (Component)

Is the desire to treat composites and primitives uniformly/the same way/atomically/transparent and requires that we declare the child managing function inside the Component ⇒ also the primitives must implement them ⇒ clients may possibly do stupid things like adding to primitives.

Solution 1

Child managing methods inside the primitive throw an UnsupportedOperationException.

Drawbacks:

- Program flow should not be controlled with exceptions
- Only transparent on the interface level

Listening 6.24 : *-NoValue-*

```
public void add/remove(Component c ){
  throw new UnsupportedOperationException();
}

public void getComponents(){
  throw new UnsupportedOperationException();
}
```

Solution 2

Methods return a `boolean` result which informs the client whether the operation was successful.
For `getComponents()` we can use `getComponents().getSize()` to check whether the composite contains components or not.

Listening 6.25 : *-NoValue-*

```
public boolean add(Component c) { return false; }
public boolean remove(Component c) { return false; }
public List getComponents() {return Collections.EMPTY_LIST;}
```

Solution 3

Method `isComposite` can be used to check whether a component is a composite or not.
⇒ `isComposite()` should be called as precondition to check if component is a composite, before add/remove,...

Listening 6.26 : *-NoValue-*

```
public boolean isComposite() { return false; }
public void add(Component c) { throw new AssertionError(); }
public void remove(Component c){throw new AssertionError();}
public List getComponents() {return Collections.EMPTY_LIST;}
```

Solution 4

Default behavior is that nothing is done

Listening 6.27 : *-NoValue-*

```
public void add(Component c) {    /* do nothing */ }
public void remove(Component c) { /* do nothing */ }
public List getComponents() {return Collections.EMTPY_LIST;}
```

Solution 5

Implement full functionality, that primitives can add and remove,...

7. Issues

Listening 6.28 : *0..1 Constraint*

```
Figure f1 = new RectangleFigure(...);
Figure f2 = new RectangleFigure(...);
Figure f3 = new OvalFigure(...);

GroupFigure g1 = new GroupFigure(f1, f2);
GroupFigure g2 = new GroupFigure(f3, f1);
```

Listening 6.29 : *0..1 Constraint*

```
Figure f1 = new RectangleFigure(...);
Figure f2 = new RectangleFigure(...);
Figure f3 = new OvalFigure(...);

GroupFigure g1 = new GroupFigure(f1, f2);
GroupFigure g2 = new GroupFigure(g1, f3);

g1.addFigure(g2);  // recursion
```

Solutions

There exists to solutions:

- **Variant 1**: Only add component if it does not belong to anything:
  Check first for 0..1 constraint and then for recursion
- **Variant 2**: If component belongs already to a composite remove it from current parent and add it to this composite.
  Check first for recursion and then for 0..1 constraint

in order to do this components need to have a parent field in order to check for recursions:

```
Component parent;
```

Listening 6.30 : *Variant 1*

```
void add(Component c) {
  // 1. Check 0..1 constraint
  if(c.parent != null) {
    throw new IllegalArgumentException("constraint violation");
  }
  // => c is for sure not part of anything
  // But c maybe composite itself

  // 2. Check for recursion
  // c has no parents => only possibility for recursion if
  // c is the same as root of this
  if(c instanceof Composite) {
    Composite comp = this;
    while (comp.parent != null) { comp = comp.parent; }
    if(comp == c) {
      throw new IllegalArgumentException("recursion");
    }
  }

  c.parent = this;
  this.children.add(c);
}
```

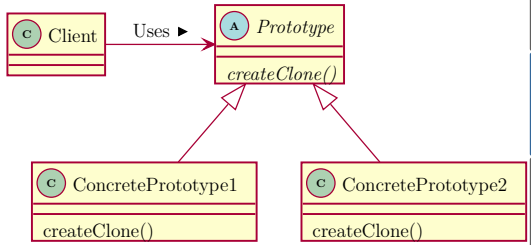
Note

Recursion only possible if `c` is the same as root of `this`. Root because we already know that `c` has not parent.

Listening 6.31 : *Variant 2*

```
void add(Component c) {
  // 1. Check for recursion
  // recursion not possible if c is primitive
  if(c instanceof Composite) {
    Composite comp = this;
    while (comp != null && comp != c) { comp = comp.parent; }
    // |go up in the hierarchy check for recursion
      if(comp == c) {
        throw new IllegalArgumentException("recursion");
      }
    }

  // 2. Establish 0..1 constraint
  if(c.parent != null) {
    c.parent.remove(c);
  }

  c.parent = this;
  this.children.add(c);
}
```

8. Prototype Pattern



9. Comparing Objects

Definition 6.10 `obj1==obj2`: Compares references/addresses of the two operands.

Definition 6.11 `obj1.equals(obj2)`: Compares the content of two objects.

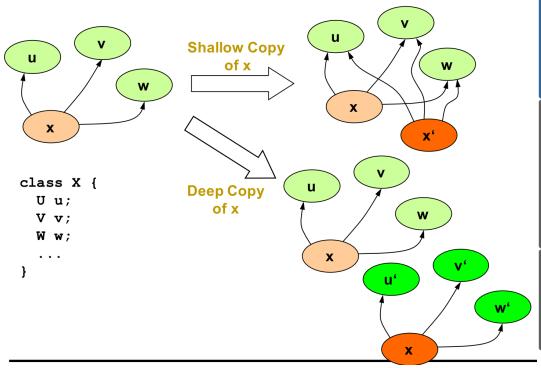
Definition 6.12 `obj1.getClass()==obj2.getClass()`: Checks if two objects are of the same dynamic type.

Definition 6.13 `obj is instanceof`: Checks if an object is an instance of a class or interface.

Intent of cloning Provide a clone that satisfies:

- New instance: `x.clone()!=x`
- Same dynamic type: `x.clone().getClass()==x.getClass()`
- Equal: `x.clone().equals(x)`

10. Deep vs. Shallow Copies



Definition 6.14 Shallow Copy: Is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object.
Thus: If any of the fields of the object are references to other mutable types/objects, just the reference addresses are copied i.e., only the memory address is copied.

```
myFoo.childObj == sFoo.childObj // ture
myFoo.childObj.equals(sFoo.childObj) // ture
```

Hence manipulating child mutable types/objects of the copy will also modify the child mutable types/objects of the original object ⇒ not a real copy.

Attention

Lists are complex types!

Listening 6.32 : *Shallow list copy*

```
public class Ex {
  private int[] data;
  // makes a shallow copy of values
  public Ex(int[] values) {
      data = values;
  }
}
```

Definition 6.15 Deep copy: Recursively copies objects, and hence is a real copy.

```
myFoo.childObj == sFoo.childObj // false
myFoo.childObj.equals(sFoo.childObj) // ture
```

Listening 6.33 : *-NoValue-*

```
class Foo {
  private Bar myBar;
  ...
  public Foo shallowCopy() {
    Foo newFoo = new Foo();
    newFoo.myBar = myBar;
    return newFoo;
  }

  public Foo deepCopy() {
    Foo newFoo = new Foo();
    newFoo.myBar = myBar.clone();
    return newFoo;
  }
}

Foo myFoo = new Foo();
Foo sFoo = myFoo.shallowCopy();
Foo dFoo = myFoo.deepCopy();

myFoo.myBar == sFoo.myBar => true
myFoo.myBar.equals(sFoo.myBar) => true
myFoo.myBar == dFoo.myBar => **false**
myFoo.myBar.equals(dFoo.myBar) => true
```

1. Java.lang.Clonable

Definition 6.16 Clonable: Is a marker interface [def. 7.38], that is special in that it is empty and does not even declare the clone method:

```
public interface Cloneable {
  // Marker interface
}
```

It is simply there to indicate that a class is clonable.

Listening 6.34 : *java.lang.Object*

```
protected Object clone() throws CloneNotSupportedException {
  if (!(this instanceof Cloneable)) {
    throw new CloneNotSupportedException(
      "Doesn't implement Cloneable interface!");
  }
  return VMMemoryManager.clone(this);
}
```

Listening 6.35 : *VMMemoryManager*

```
class VMMemoryManager {
  static native Object clone(Object object);
  ...
}
```

Definition 6.17 nativ: Enables java code running JVM to call& be called by nativ applications (programs specific to hardware&OS platform) and libraries written in other languages e.g. C.
Here: it calls C-code that uses memcpy to copy the given object bitwise efficiently (⇒ shallow copy).

2. Java.lang.Object and Clonable

Definition 6.18 Java.lang.Object: Is the base class of all Classes and provides a protected default implementation of the clone method which:

- Throws an CloneNotSupportedException if an object of a class calls the clone method without implementing the Cloneable marker interface
- Will lead to a compile time error if we try to copy an object not from within itself, subclasses, or classes of the same package.
  This is because clone is protected

How to use: java.lang.Object.clone the right way:

- If the Class we want to clone only contains primitive fields or references to immutable objects, then we can simply:
  Shallow Copy:
  - override the clone method as a public method
  - call super.clone() to make a shallow copy using memcpy

```
visibility class MyClass implements Cloneable{
  ...
  public Object clone(){
    try{
      // go up in hierarchy until
      // java.lang.Object.clone
      MyClass c = (MyClass) super.clone();
      return c;
    } catch (CloneNotSupportedException e){
      throw new InternalError(e.getMessage());
    }
  }
}
```

Notes

- Type Conversion: MyClass may be a subclass of another class i.e. the parent class will call again super.clone until we reach java.lang.object.
  Hence we need to make sure that we will obtain the right dynamic type (x.clone().getClass()==x.getClass()).
- try: again if MyClass is a subclass of other, the super/-parent classes need to implement at least the clonable marker interface (do not necessarily need to implement clone method).

- If the Class we want to clone does contains complex fields or references to mutable types then we need to:
  Deeply Copy:
  - Recursively clone all elements
  - Replace lists with new ones

Listening 6.36 : *Company: Deep Copy*

```
public class Company implements Cloneable {
  private String name;
  // Java lists are mutable data types!
  private ArrayList<Employee> employees = new ArrayList<>();
  public Object clone() {
    try {
      Company c = (Company) super.clone();
      c.employees = new ArrayList<>();
      for(Employee e : employees)
        c.employees.add(e.clone());
      return c;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.getMessage());
    }
  }
}
```

Listening 6.37 : *Employee: Shallow Copy*

```
public class Employee implements Cloneable {
  // Only primitive types
  private String name;
  private int yearOfBirth;
  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e.getMessage());
    }
  }
}
```

Why is clone protected?

- Clone is declared protected so that if all we do is implement Cloneable, only subclasses and members of the same package will be able to invoke clone() on the object.
  To enable any class in any package to access the clone() method, you'll have to override it and declare it public!
- Clone cannot be invoked on objects of static type Object.

Listening 6.38 : *Compile Time Error*

```
import myClass
// Implementing (empty) marker interface
// But does not implement clone method

class Client{
  public static void main(String[] args){
    myClass obj1 = new myClass();
    // Compile time error => calling protected method
    myClass obj2 = obj1.clone();

  }
}
```

Notes

- Java allows covariant typing
- Alias references and cycles have to be handled manually

3. Clone with copy Constructor

Java does not provide a default copy constructor as in C++ but we may write one and then use it for our clone method.

Listening 6.39 : *Clone with copy constructor*

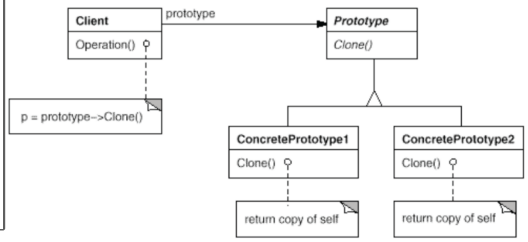
```
class Company {
  public Company(Company c) {
    if(c==null) throw new IllegalArgumentException();
      // initialize this with attributes of c
    }
    ...
    public Company clone() {
      return new Company(this);
    }
}
```

Notes

- It is possible that the state of an immutable type changes after creation, as long as this change is not visible to the client.
- Companion mutables: provide conversion classes e.g. immutable string and mutable stringBuffer

Provide conversionhh

4. Prototype Design Pattern



Problem

New object creation MyClass obj = new MyClass() is costly e.g. an object is created by calling a costly database operation inside the constructor.
Thus if we need many similar types it is not a good idea to always construct a new object using new.

Intent Avoid costly creation of an object using new by *cloning it from some similar prototype* and simply adapt it to our needs.
Thus: we specify the kinds of objects to create using a prototypical instance, and create new objects by cloning the right prototype.

Participants

- **Prototype**:
  - ◆ Is in its easiest form the java.lang.Clonable interface
  - ◆ An interface that implements the clonable interface
    - Implements the clonable marker interface
    - declares a public clone method
    - may declare methods for other classes
  - ◆ An abstract class e.g. Shape implementing the java.lang.Clonable interface
- **Concrete Prototype**: is an concrete class implementing the prototype interface that we may want to clone
- **Client**: Produces new instances by calling clone on concrete prototypical prototypes.

Note: Why not just use clonable interface

Many consider clonable as broken/bad design as it does not even declare clone thus checking anything with the clonable marker interface does not bring us anything e.g.

```
concretePrototype a = new A();
if(a instanceof Cloneable)
{
  // This check is worthless and
  // the method may fail
    A copied = a.clone();
}
```

Listening 6.40 : *Abstract Prototype*

```
abstract class Prototype implements Cloneable {

  public Prototype clone() throws CloneNotSupportedException {
    return (Prototype) super.clone();
  }
  public abstract void setX(final shot X);
  public abstract short getX();
}
```

Listening 6.41 : *Concrete Prototype*

```
class ConcretePrototype implements Prototype {
  private int x;

  public ConcretePrototype extends Prototype {
    return (Prototype) super.clone();
  }

  public abstract void setX(int X){
    x = X;
  }

  public abstract short getX(){
    return x;
  }
}
```

Listening 6.42 : *Client*

```
public class Client{
  public static void main(String[] args) {
  Prototype origin =  new ConcretePrototype(10);
  Prototype clone =  origin.clone();
  clone.setX(4);
}
```

5. Exmple

Listening 6.43 : *Abstract Prototype*

```
public abstract class Figure implements Cloneable {
  ...
  public Object clone() {
      Object clone = null;
      try {
         clone = super.clone();
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }
      return clone;
      // Or do not let clone abstract base class
    }
}
```

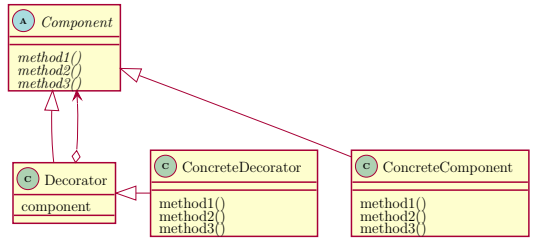
Listening 6.44 : *Concrete Prototype*

```
public class Rectangle extends Figure {
  ...
  public Object clone() {
      Object clone = null;
      try {
         clone = super.clone();
         // or provide deep copy
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }
      return clone;
    }
}
```

Listening 6.45 : *Client*

```
public class Client{
  ...
  Rectangle greenRec = new Rectangle(green);
  Circle blueCircle = new Circle(blue);
  Circle yellowCircle = new Circle(yellow);

  public static void main(String[] args) {
    Rectangle myRecClone = greenRec.clone();
    Circle myCircleClone = yellowCircle.clone();
  }
```

11. Decorator Pattern



Intent We have some basic building blocks that share common method declarations but may have lots of different functionality.
⇒ attach dynamically additional responsibility to an object by passing it to a decorator.

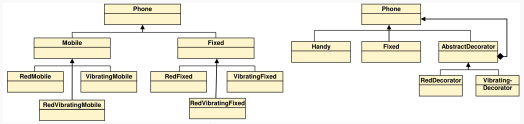


Figure 3: Inheritance vs. Decoration

n : number of properties

# of Subclasses with inheritance&combination	$2^n - 1$
# of additoinal Classes with decoration	$n + 1$

Participants

- **Basic building blocks**: is, or are they things that we want to decorate/add functionality to.
- **Component Interface**: declares the shared methods of the basic building blocks that may be decorated by the decorator.
  The component interface is also implemented by the decorator.
  ⇒ Leads to consistency between the decorator and the basic building blocks by declaring the methods that may be decorated.
- **Decorator**: Is an *optional*
  - abstract wrapper class that defines the shared methods and fields of the concrete decorators i.e. stores a reference to a basic building block or more commonly
  - a wrapper class that
    - stores a reference to the basic building block
    - simply calls the to be decorated methods of the reference to the building block
- **Client**: creates decorated classed by handing/decorating basic building blocks by decorators.

Note

A concrete Decorator class is useful if concreteDecorators do not decorate all methods that may possibly decorated!

Listening 6.46 : *Component e.g. Figure*

```
interface Component{
  public type op1(args);     // e.g. draw
  public type op2(args);     // e.g. setBounds
}
```

Listening 6.47 : *Concrete Component/Basic Building Block e.g. Rectangle*

```
class concreteCompA implements Component{
  public type op1(args){

optcdo something1
  };
  public type op2(args){
      do something2
  }
}
```

Listening 6.48 : *Decorator*

```
class Decorator implements Component{
  Component inner;
  public Decorator(Component inner){
    this.inner = inner;
  }
  // No decoration yet
  public type op1(args){
    inner.op1(args);
  }

  public type op2(args){
    inner.op2(args);
  }
}
```

Listening 6.49 : *ConcreteDecorator e.g. ColorDecorator*

```
class op1Decorator extends Decorator{
  @override
  public type op1(args){
    // Decorate before
    inner.op1(args);
    // Decorate after
    addFunctionality(args);
  }

  public type addFunctionality(args){
    // Do something
  }
}
```

Note

`inner` may very well be another `concreteDecorator` and not a basic building block.

Listening 6.50 : *Client*

```
Oval ov = new Oval();
RedColorDecorator redRec = new RedColorDecorator(ov);
```

Listening 6.51 : *Client Example 2*

```
// MobilePhone implements Phone and has methods vibration and color
MobilePhone m = new MobilePhone();
RedColorDecorator redRec = new RedColorDecorator(ov);
Phone redVibratingPhone =
        new RedColorDecorator(new VibrationDecorator(m));
```

Notes

- May apply decorator before or after e.g. may use decorator for some type safety checking
- May also make sense to pass inner to setter method.

1. Fields

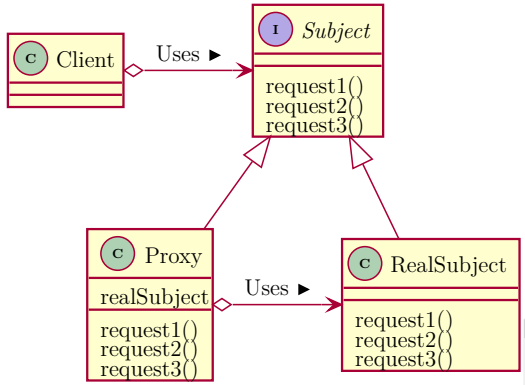
Fields cannot be decorated explicitly **but** if the basic building blocks use `getters` and `setters` then we may decorate them implicitly via those methods.

Decorator vs State vs Strategy

- A Decorator changes an object's skin: that is it usually adds functionality (decorates) e.g. DrawTools may be decorators of the view.
- A Strategy changes an object's guts: that is it is likely to change the internal strategy
- State changes an object's guts: that is it is likely to changed its functionality completely

12. Proxy Pattern

Definition 6.19 Proxy: Latin word meaning "the authority to represent someone else".



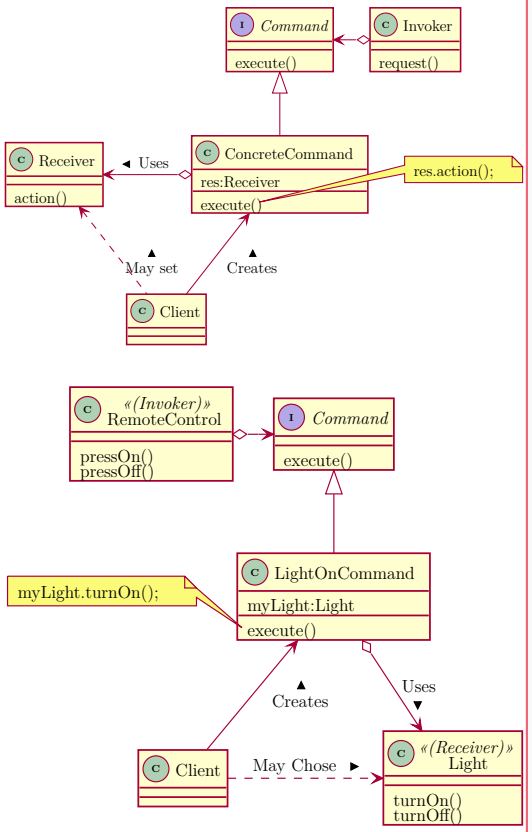
Intents

- Use a surrogate or placeholder to control the access of original object ⇒ protection of the original object from the outside world.
- Lazy initialization of resource hungry objects

Participants

- **Proxy interface**: An interface that is implemented by the concrete object and the proxy object
- **Dynamic proxy**
  - Instance of a dynamic proxy class
  - Dynamic proxy class implements a list of interfaces specified at runtime when dynamic proxy instance is generated (not at compile time)
- **Invocation Handler Object**
  - Each proxy instance has an associated invocation handler object which implements the interface InvocationHandler
  - A method invocation on a proxy instance trough one of its proxy interfaces will be dispatched to the invoke() method of the instance's invocation handler
- **Concrete Object**: Object which is used in invocation handler

13. Command Pattern



Note: UML dotted line

- Is only a weak dependency and not a full association. Client does not necessarily need to know anything about the Receiver.
- here in this example we use <<()>> for comments

IntentionEncapsulates an action or request into an object such that it can be invoked at a later time.
Examples:

- Client wants to perform action at later time: undo/redo
- We want to queue requests s.t. the can be handled by a receiver in a timely manner.

encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command

Participation

- **Command Interface**: declares an interface for executing operations. In its simplest form, this interface includes an abstract execute operation.
- **Concrete Command**: Each concrete Command class specifies a receiver-action pair by storing the Receiver as an instance variable.
  The concrete command overrides the `execute` method such that it fulfills the required action/request with the help of the stored `receiver` object.
- **Receiver**: the Receiver has the knowledge required to carry out a given request/action.
- **Invoker**: Decides when the methods is called and then asks the command to perform the request. The invoker is an `aggregation` (≠ implementation) of command objects and is called invoker because it can invoke the `execute` method of its stored `commands`.
  Thus: it can for example:
  - Have a list of commands to process

```
class Invoker{
  ArrayList<Command> l = new ArrayList<Command>();
  ...
  void invoke Command{
    for(Command c : l) {
      // if resources available
      c.execute();
    }
  }
```

  - Has references to the concrete commands, which can be invoked by the client e.g.:

```
class Switch{
  private Command onCommand;
  private Command offCommand;

  public Invoker (Command on, Command off){
    this.onCommand = on;
    this.offCommand = off;
  }
  void on(){
    onCommand.execute();
  }
  void off(){
    offCommand.execute();
  }
}
```

    hence the client may a pair of on/off commands to the invoker e.g. Light/Fan,... and the simply use the switch.
- **Client**: The client instantiates the Invoker, the Receiver, and the concrete command objects.

Problems

If we store every movement of 1 pixel, we will have way to many move commands ⇒ store only periodically.

Listening 6.52 : *Command Interface*

```
public interface Command {
    public abstract void execute();
}
```

Listening 6.53 : *Fan (Receiver 1)*

```
class Fan {
  public void startRotate() {
          System.out.println("Fan is rotating");
  }
  public void stopRotate() {
          System.out.println("Fan is not rotating");
  }
}
```

Listening 6.54 : *Light (Receiver 2)*

```
class Light {
  public void turnOn( ) {
          System.out.println("Light is on ");
  }
  public void turnOff( ) {
          System.out.println("Light is off");
  }
}
```

Listening 6.55 : *Invoker*

```
class Switch {
   private Command UpCommand, DownCommand;
   public Switch( Command Up, Command Down) {
       UpCommand = Up;
       DownCommand = Down;
   }
   void flipUp(){
     UpCommand.xecute();
   }
   void flipDown(){
     DownCommand.execute();
   }
}
```

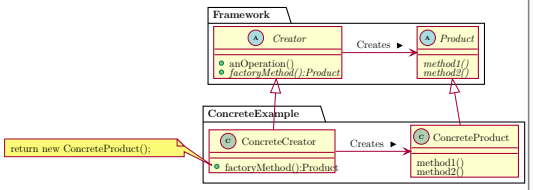
Listening 6.56 : *Concrete Commands*

```
class LightOnCommand implements Command {
  private Light myLight;
  public LightOnCommand ( Light L) {
          myLight  =  L;
  }
  public void execute( ) {
         myLight.turnOn( );
  }
}
class LightOffCommand implements Command {
  private Light myLight;
  public LightOffCommand ( Light L) {
          myLight  =  L;
  }
  public void execute( ) {
          myLight.turnOff( );
  }
}
// analoguous for fan but rotate
```

Listening 6.57 : *Client*

```
public class Client {
  public static void main(String[] args) {
    Light  testLight = new Light( );
    LightOnCommand testLOC = new LightOnCommand(testLight);
    LightOffCommand testLFC = new LightOffCommand(testLight);
    Switch testSwitch = new Switch( testLOC,testLFC);
    testSwitch.flipUp( );
    testSwitch.flipDown( );
    Fan testFan = new Fan( );
    FanOnCommand foc = new FanOnCommand(testFan);
    FanOffCommand ffc = new FanOffCommand(testFan);
    Switch ts = new Switch( foc,ffc);
    ts.flipUp( );
    ts.flipDown( );
  }
}
```

14. Factory Method



Intent

- Making program code independent of concrete classes:
- a class can't anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates

Thus: creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

Listening 6.58 : *Bad (statically typed)*

```
Interf_X ref = new X();
```

Listening 6.59 : *factoryMethod*

```
InterfX ref = cCreat.getX();
```

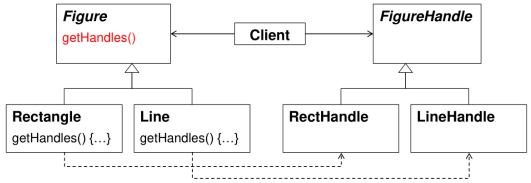


Figure 4

Definition 6.20 GOF Definition: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Note

People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes.

Note

The Factory Method pattern has commonly been misunderstood to mean any method whose sole purpose is to construct an object and return this created object. In most cases, this translates to a static method that abstracts the construction process of an object

Participants

- **Abstrac/Interface Product**: specifies type of product to be created e.g. HandleType
- **Abstrac/Interface Creator**: Object that will instantiate the product may be
  - an `interface` which includes one or more Factory Methods, as well as any number of other methods.
  - an `abstract class` with defined methods or even a default implementation of the Factory Method that returns a default instance of Product.
- **Concrete Creator**: creator class (≠ factory)s that uses factory method & returns different `ConcreteProduct` instances (that all implement the Product interface) depending on the the concrete creator.

Implementation

- **In a static method**: simplified version of factoryMethod (pattern): where the Creator hierarchy is reduced to a single class and the factory method is reduced to a single static method (rather than a method that is overridden for each of the possible products).
  The use of this `static` method allows for a single point of entry into a library. We use the class in order to obtain a concrete object.

```
public class XmlHandlerFactory {
  public static XmlHandler createHandler() {
    return new XmlPrintHandler();
  }
}
...
XmlHandler handler = XmlHandlerFactory.createHandler();
```

- **Sub classes**: as discussed above
- **In an separate object** = Abstract Factory pattern

Note: static approach

assumes that the `XmlHandler` creator has enough knowledge to know which handler implementation is appropriate.

Difference Abstrac Factroy

Factory Method is similar to Abstract Factory but without the emphasis on families.

Listening 6.60 : *Product Interface*

```
public interface EncryptionAlgorithm {
  public String encrypt(String plaintext);
}
```

Listening 6.61 : *Concrete Product*

```
public class Sha512EncryptionAlgorithm implements EncryptionAlgorithm {
  @Override
  public String encrypt(String plaintext) {
      return DigestUtils.sha512Hex(plaintext);
  }
}
```

Listening 6.62 : *Abstract Creator*

```
public abstract class Encryptor {
  public void writeToDisk(String plaintext, String filename) {
      EncryptionAlgorithm encryptionAlgorithm = getEncryptionAlgorithm();
      String cyphertext = encryptionAlgorithm.encrypt(plaintext);
      ...
  }
  public abstract EncryptionAlgorithm getEncryptionAlgorithm();
}
```

Listening 6.63 : *ConcreteCreator*

```
public class Sha256Encryptor extends Encryptor {
  @Override
  public EncryptionAlgorithm getEncryptionAlgorithm() {
      return new Sha256CEncryptionAlgorithm();
  }
}
```

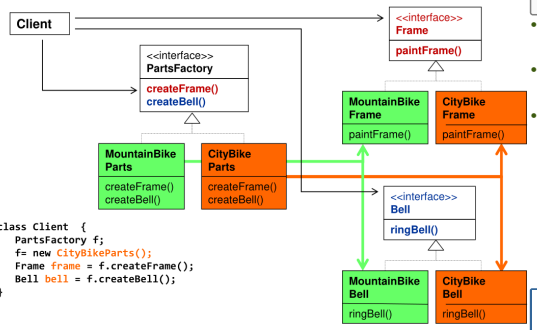
Implementation Issues

- static factory implementations method cannot be overridden (as the method is static) ⇒ can be compared to a `final` factory method
- Factory methods my be parameterized to describe the product it creates:

```
Product createProduct(ProductID id)
```

- Factory Method in abstract base class may provide default return type.

15. Abstract Factory Pattern



Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Abstract Factory=Abstract Method+Strategy Pattern

- **Abstract/Interface Factory**: declares methods in order to create products of a certain family by:
  - declaring an interface operations that create abstract products.

```
interface Factory{
  intA createA();
  intB createB();
}
```

  - declaring an abstract class that provides methods to produce abstract products

```
abstract class Factory{
  public abstract intA createA();
  public abstract intB createB();
}
```

  ⇒ methods to create product must share common interface:

```
intA createProdA();
```

- **Concrete Factories**: implements operations to create concrete products.
- **Abstract/Interface Product**: declares an interface for a type of product.
- **Concrete Products**: products to be created by the corresponding ConcreteFactory;
  Need to implement the corresponding product interface in order to be consistent with the factories.
- **Client**: Uses a factory in order to obtain products of a certain family.
  Note: this may

1. Where is the concrete Factory

- Directly used by client
- by another class that uses e.g. factory method and singleton

2. Who creates the concrete factory?

- **Externally**:

```
CurrentFactory.setFactory(new Factory1());
```

- **Internally**:

```
CurrentFactory.setFactory(new Factory1());
```

- **Automatically upon loading**:

```
class Factory1 implements Factory {
    public A createA() { ... }
    public B createB() { ... }
    static {
      CurrentFactory.setFactory(new Factory1());
    }
    private Factory1() { };
  }
```

Definition 6.21 Static Block: Will be called only once at initialization even-though if we call the constructor multiple times.

3. Difference factory Method and abstractFactory

The main difference between a "factory method" and an "abstract factory" is that the factory method is a single method, and an abstract factory is an object. I think a lot of people get these two terms confused, and start using them interchangeably. I remember that I had a hard time finding exactly what the difference was when I learnt them.

Factory Method Paradigm

. . . the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

Abstract Factory Paradigm

With the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition.
What they're saying is that there is an object A, who wants to make a Foo object. Instead of making the Foo object itself (e.g., with a factory method), it's going to get a different object (the abstract factory) to create the Foo object.

Factory vs. DI

When using a factory your code is still actually responsible for creating objects. By DI you outsource that responsibility to another class or a framework, which is separate from your code.

16. Singleton Pattern

Insure that a class has only one instance at a time.

Idea: use only static variables methods

Problems:

- Order in which static inializers are called is not determined

```
class A {
  static { System.out.println("A()"); }
  static int x = B.x + 1;
}
class B {
  static { System.out.println("B()"); }
  static int x = A.x + 1;
}

public class Initialization {
  public static void main(String[] args) throws Exception
  ↪  {
    System.out.println("A.x = " + A.x);
    System.out.println("B.x = " + B.x);
  }
}
```

- We may require run-time information

ParticipantsOnly final Singleton class that provides a private constructor.
final such that the constructor cannot be overridden in a child class.

1. Eager Initialization

Problem: singleton is instantiated when the class is first accessed.
If the class contains other `static` fields or methods we may create a costly object that we do not even need atm.

```
public final class Singleton {
private Singleton() { }
  private static Singleton instance = new Singleton();
    public static Singleton getInstance() {
    return instance;
  }
}
```

2. Lazy Initialization

Only create new instance if it does not already exist:

```
public final class Singleton {
  private Singleton() { }
  private static Singleton instance = null;

  public static synchronized Singleton getInstance() {
    if(instance == null) instance = new Singleton();
    return instance;
}
```

Problem

`synchronized`: is important for thread safety but really expensive and we need to synchronize on every `getInstance`

Solution: singleton with double checking

Tries to avoid costly synchronization by first checking synchronized.

```
public class Singleton {
    private volatile static Singleton instance;
    public static Singleton getInstance() {
      if(instance == null) {
        synchronized(Singleton.class) {
          if(instance == null) {
            instance = new Singleton();
          }
        }
      }
      return instance;
    }

    private Singleton() { /* initialization */ }
      // other methods
  }
```

Note

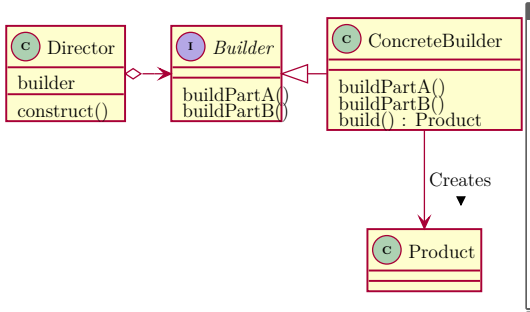
Need to volatile so that compiler does not optimize away our if.

3. Holder Initialization

Instance is created upon first access of getInstance (lazy instantiation) and not on access of singleton static fields.

```
public final class Singleton {
  private static class Holder {
    private static final Singleton INSTANCE = new
    ↪  Singleton();
  }
  private Singleton() { }

  public static Singleton getInstance() {
    return Holder.INSTANCE;
  }
}
```

17. Builder Pattern



IntentSeparate the construction of a complex object from its representation so that the same construction process can create different representations.

Participants

- **Builder**: Object which is used to build other objects.

Listening 6.64 : *-NoValue-*

```
public class Pizza {
  private final int size;
  private final boolean cheese, pepperoni, bacon;

  public static class Builder {
    private final int size;  // required
    private boolean cheese = false;  // optional
    private boolean pepperoni = false;
    private boolean bacon = false;
    private boolean pinapple = false;

    Builder(int size) { this.size = size; }

    Builder cheese(boolean c) { cheese = c; return this; }
    Builder pepperoni(boolean p) { pepperoni=p; return this; }
    Builder bacon(boolean b) { bacon = b; return this; }
    // return finished build pizza (create method)
    Pizza build() { return new Pizza(this); }
  }

  public static class HawaiBuilder {
    private final int size;  // required
    private boolean cheese = HawaiChees;  // optional
    private boolean pinapple = true;

    Builder(int size) { this.size = size; }
    Pizza build() { return new Pizza(this); }
  }

  private genericPizza(Builder builder) {
      size = builder.size;
      cheese = builder.cheese;
      pepperoni = builder.pepperoni;
      bacon = builder.bacon;
  }
}
```

Listening 6.65 : *Usage*

```
Pizza pizza = new Pizza.Builder(12).
                        .cheese(true)
                        .pepperoni(true)
                        .bacon(true)
                        .build();
Pizza pizza = new Pizza.HawaiBuilder(12);
```

1. General Example

Listening 6.66 : *Product*

```
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough) {
        this.dough = dough;
    }

    public void setSauce(String sauce) {
        this.sauce = sauce;
    }

    public void setTopping(String topping) {
        this.topping = topping;
    }
}
```

Listening 6.67 : *Abstract Builder*

```
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }

    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Listening 6.68 : *Concrete Builder*

```
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("cross");
    }

    public void buildSauce() {
        pizza.setSauce("mild");
    }

    public void buildTopping() {
        pizza.setTopping("ham+pineapple");
    }
}
```

Listening 6.69 : *Concrete Builder*

```
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("pan baked");
    }

    public void buildSauce() {
        pizza.setSauce("hot");
    }

    public void buildTopping() {
        pizza.setTopping("pepperoni+salami");
    }
}
```

Listening 6.70 : *Director*

```
class Waiter {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }

    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }

    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

Listening 6.71 : *Customer*

```
public class PizzaBuilderDemo {
    public static void main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzabuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        waiter.setPizzaBuilder( hawaiianPizzabuilder );
        waiter.constructPizza();

        Pizza pizza = waiter.getPizza();
    }
}
```

Structural Patters

Creational Patters