CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 11 Template, Factory Method, Effective Java
Sep 29, 2019

Java External and Internal Iterators

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
 numbers.add(2);
 numbers.add(3);
 numbers.add(10);
```

External

```
Iterator<Integer> external = numbers.iterator();

while (external.hasNext()) {
    System.out.println("" + external.next());
}
```

Internal

```
numbers.forEach(value -> System.out.println("" + value));
```

Restriction on Internal arguments

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
 numbers.add(2);
 numbers.add(3);
 numbers.add(10);

 Iterator<Integer> external = numbers.iterator();

 int sum = 0;
 while (external.hasNext()) {
    sum = sum + external.next();
}
assert sum == 15;


sum = 0
numbers.forEach(value -> sum = sum + value);  // Compile Error
                                                    sum needs to be final
```

We can use a final object

```
public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(2);
    numbers.add(3);
    numbers.add(10);

    final Count newSum = new Count();
    numbers.forEach(value -> newSum.add(value));
    assert newSum.getCount() == 15;
}
```

```
class Count {
    int count = 0;

    public void add(int value) {
        count += value;
    }

    public int getCount() {
        return count;
    }
}
```

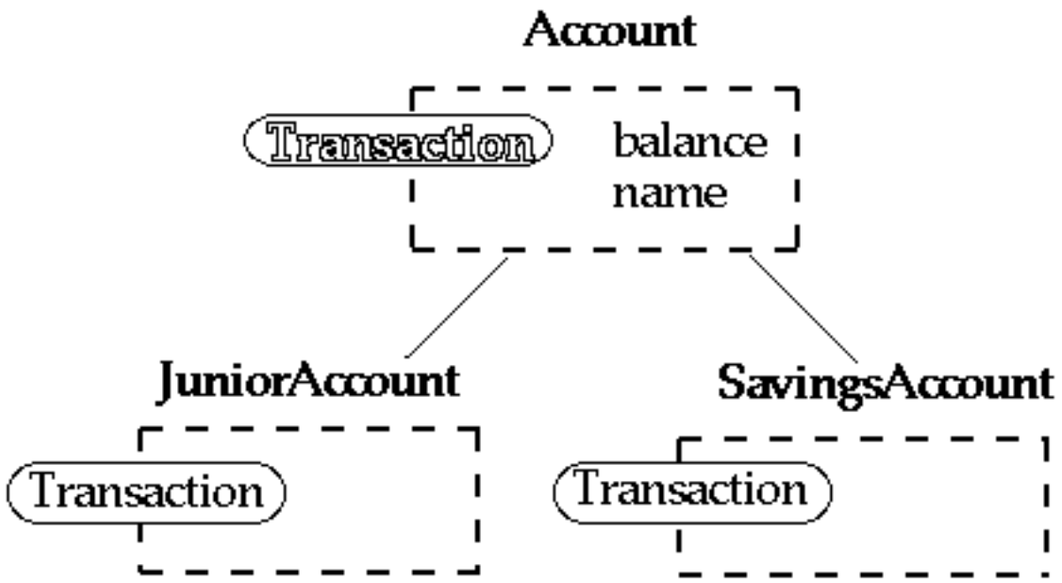
Template Method

Polymorphism

```
class Account {
    public:
        void virtual Transaction(float amount)
                { balance += amount;}
        Account(char* customerName, float InitialDeposit = 0);
    protected:
        char* name;
        float balance;
}

class JuniorAccount : public Account {
    public:   void Transaction(float amount) {//code here}
}

class SavingsAccount : public Account {
    public:   void Transaction(float amount) {//code here}
}

Account* createNewAccount(){
    // code to query customer and determine what type of
    // account to create
};
```

```
main() {
    Account*  customer;
    customer = createNewAccount();
    customer->Transaction(amount);
}
```

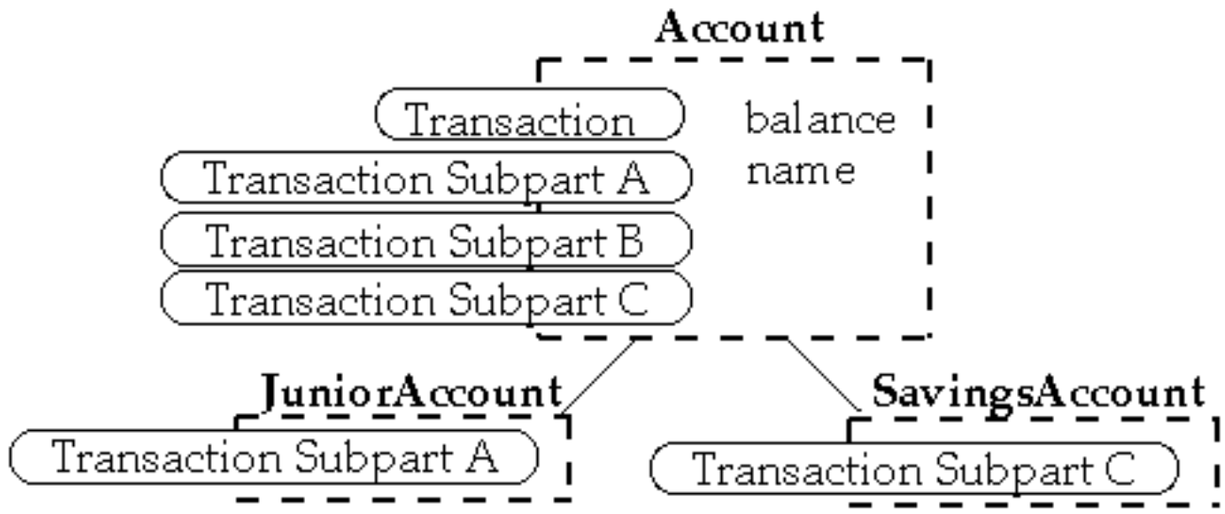
Deferred Methods




```
class Account {
   public:
      void virtual Transaction() = 0;
}

class JuniorAccount : public Account {
   public
      void Transaction() { put code here}
}
```

Template Method

```
class Account {
    public:
        void Transaction(float amount);
    protected:
        void virtual TransactionSubpartA();
        void virtual TransactionSubpartB();
        void virtual TransactionSubpartC();
}

void Account::Transaction(float amount)  {
    TransactionSubpartA();        TransactionSubpartB();
    TransactionSubpartC();        // EvenMoreCode;
}

class JuniorAccount : public Account {
    protected:            void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
    protected:            void virtual TransactionSubpartC(); }

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```



Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

Java Example

```
import  java.awt.*;
class  HelloApplication  extends  Frame
   {
   public  void  paint(  Graphics  display  )
      {
      int startX  =  30;
      int startY  =  40;
      display.drawString(  "Hello World", startX, startY  );
      }
}
```

Ruby LinkedList Example

```
class LinkedList
  include Enumerable

  def [](index)
    Code not shown
  end

  def size
    Code not shown
  end

  def each
     Code not shown
  end

  def push(object)
   Code note shown
  end

end
```

```
def testSelect
  list = LinkedList.new
  list.push(3)
  list.push(2)
  list.push(1)

  a = list.select {|x| x.even?}
  assert(a == [2])
end
```

Where does list.select come from?

Methods defined in Enumerable

all?	any?	collect	detect
each_cons	each_slice	each_with_index	entries
enum_cons	enum_slice	enum_with_index	find
find_all	grep	include?	inject
map	max	member?	min
partition	reject	select	sort
sort_by	to_a	to_set	zip

All use "each"

Implement "each" and the above will work

java.util.AbstractCollection

Subclass AbstractCollection

Implement
- iterator
- size
- add

Get
- addAll
- clear
- contains
- containsAll
- isEmpty
- remove
- removeAll
- retainAll
- size
- toArray
- toString

Consequences

This is the most commonly used of the 23 GoF patterns

Important in class libraries

Inverted control structure

- Parent class calls subclass methods

Consequences

Inverted control structure

Java's paint method is a primitive operation called by a parent method

Beginning Java programs don't understand how the following works:

```
import  java.awt.*;
class  HelloApplication  extends  Frame
  {
  public  void  paint(  Graphics  display  )
     {
     int startX  =  30;
     int startY  =  40;
     display.drawString(  "Hello World", startX, startY  );
     }
  }
```

Consequences

Template methods tend to call:

- Concrete operations
- Primitive (abstract) operations
- Factory methods
- Hook operations

Provide default behavior that subclasses can extend

It is important to denote which methods

- Must overridden
- Can be overridden
- Can not be overridden

Refactoring to Template Method

Simple implementation

- Implement all of the code in one method
- The large method you get will become the template method

Break into steps

- Use comments to break the method into logical steps
- One comment per step

Make step methods

- Implement separate methods for each of the steps

Call the step methods

- Rewrite the template method to call the step methods

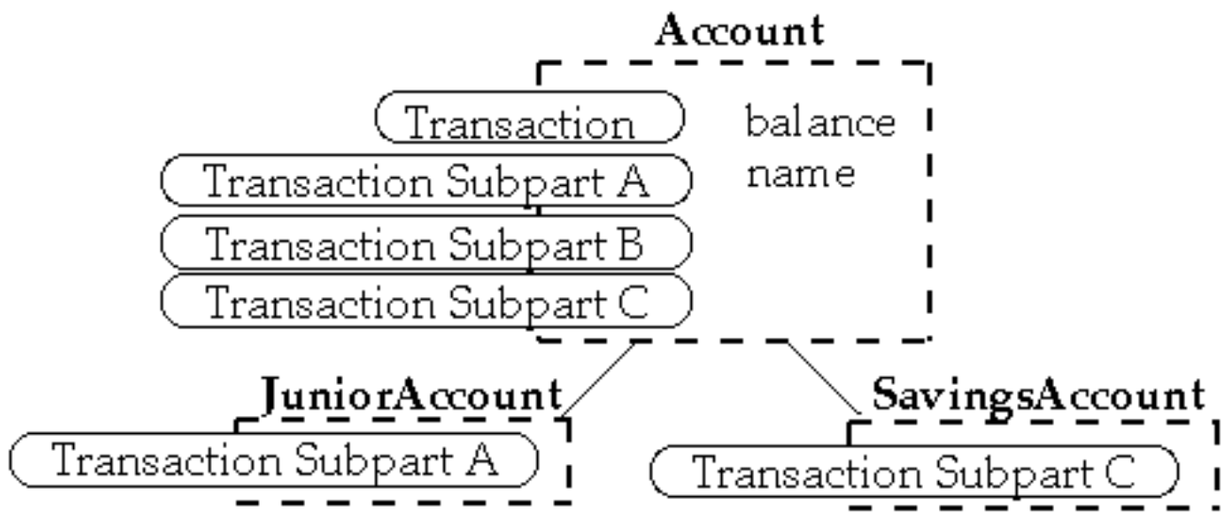
Repeat above steps

- Repeat the above steps on each of the step methods
- Continue until:
  - All steps in each method are at the same level of generality
  - All constants are factored into their own methods

Design Patterns Smalltalk Companion pp. 363-364.

Template Method & Functional Programming

```
class Account {
    public:
        void Transaction(float amount);
    protected:
        void virtual TransactionSubpartA();
        void virtual TransactionSubpartB();
        void virtual TransactionSubpartC();
}

void Account::Transaction(float amount)  {
    TransactionSubpartA();        TransactionSubpartB();
    TransactionSubpartC();        // EvenMoreCode;
}

class JuniorAccount : public Account {
    protected:            void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
    protected:            void virtual TransactionSubpartC(); }

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```



Template Method & Functional Programming

Pass in functions

```
def transaction(defaultPartA, defaultPartB, defaultPartC, amount, account) {
   defaultPartA();
   defaultPartB();
   defaultPartC();
   code code;
}
```

But this adds a lot of arguments

Requires knowing internal workings of transaction

Currying & Partial Evaluation

Pass in functions

```
def defaultTransaction = transaction(defaultPartA, defaultPartB, defaultPartC);
def juniorTransaction = transaction(juniorPartA, defaultPartB, defaultPartC);

defaultTransaction(amount, account);
juniorTransaction(amount, account);
```

But this requires knowing the account type

Multi-methods

```
defmulti transaction(amount, account) (getAccountType)

defmethod transaction(amount, account) (:default) {
   return defaultTransaction(amount, account);
}

defmethod transaction(amount, account) (:junior) {
   return juniorTransaction(amount, account);
}

transaction(amount, account);
```

Now have dynamic dispatch on the type like Java

Template Method vs Functional Solution

	Template Method	Functional
Method Variation	In multiple classes/files	In one place
Add new Variation	Add class/file + method	Add function
Type Logic	In class & parent class	

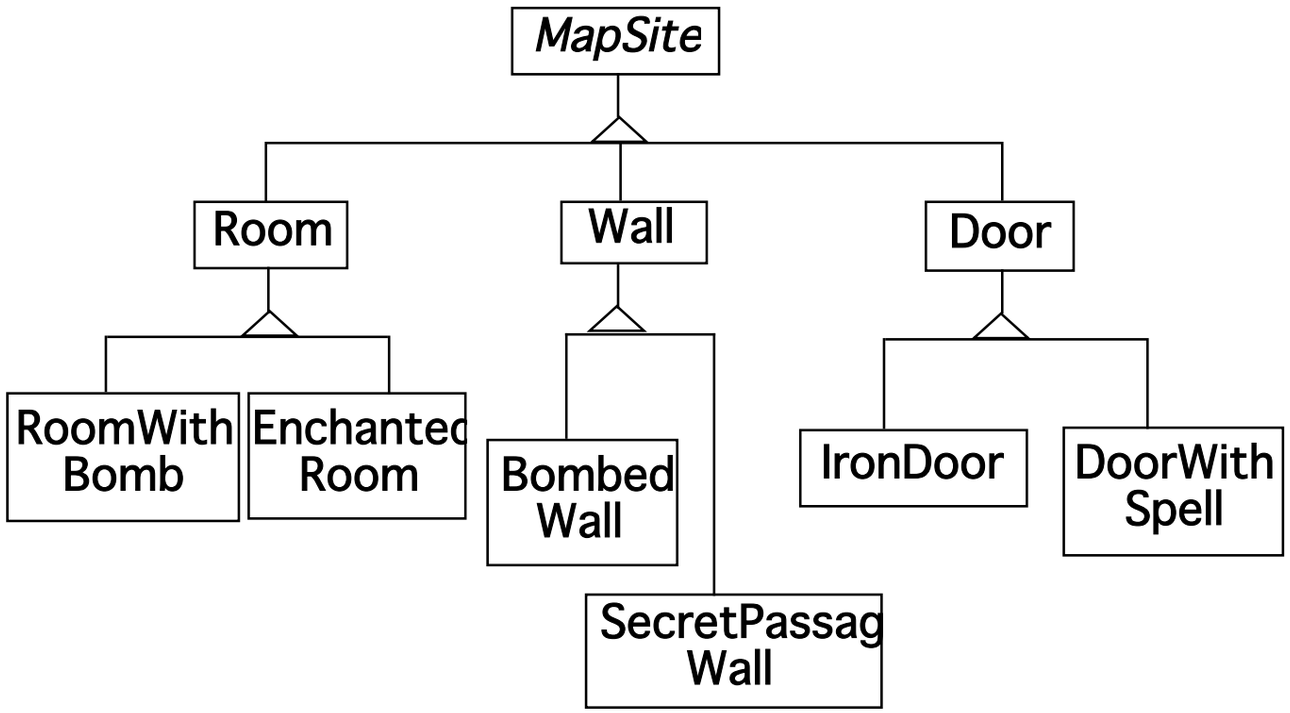
Factory Method

Factory Method

A template method for creating objects

```
public class Example {
   protected Bar bar() { return new Bar(); }

    public void foo() {
      blah
      Bar soap = bar();
      blah;
   }
```

Maze Game Example



Maze Game Example

```
class MazeGame{
    public Maze makeMaze() { return new Maze(); }
    public Room makeRoom(int n ) { return new Room( n ); }
    public Wall makeWall() { return new Wall(); }
    public Door makeDoor() { return new Door(); }

    public Maze CreateMaze(){
        Maze aMaze = makeMaze();
        Room r1 = makeRoom( 1 );
        Room r2 = makeRoom( 2 );
        Door theDoor = makeDoor( r1, r2);

        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        etc

        return aMaze;
        }
}
```

```
class BombedMazeGame extends MazeGame {

    public Room makeRoom(int n )  {
        return new RoomWithABomb( n );
    }

    public Wall makeWall() {
        return new BombedWall();
    }
}
```

Don't repeat your self

```
public class LinkedList extends Collection {
   static public OrderedLinkedList() {
      this(defaultOrder());
   }

   public LinkedList(Order listOrder ) {
      this(listOrder, new OrderedCollection());
   }

   public LinkedList(Collection items) {
      this(defaultOrder(), items);
   }

   protected Order defaultOrder() {
      return new RandomOrder();
   }

   public LinkedList(Order listOrder, Collection items) {
      blah
```

Implementation Variation

```
class Hershey {

    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == MarsBars ) return new MarsBars();
        if ( id == M&Ms ) return new M&Ms();
        if ( id == SpecialRich ) return new SpecialRich();

        return new PureChocolate();
    }

class GenericBrand extends Hershey {
    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == M&Ms ) return new Flupps();
        if ( id == Milk ) return new MilkChocolate();
        return super.makeChocolateStuff(id);
    }
}
```

Smalltalk Variant

Return the class, caller creates an object

```
chocolateStuff
        ^SpecialRich
```

```
some code
candy := (self chocolateStuff) new
mode code
```

Use Factory Method When

A class can't anticipate the class of objects it must create

A class wants its subclasses to specify the objects it creates

You want to localize the knowledge of which help classes is used in a class

But when is this?

CS 580 Example - Testing a Server

```
public class SDTwitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = new ServerSocket( port );

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    void processRequest(InputStream in,OutputStream out) {
        do a bunch of stuff
    }

    etc.
```

Using Factory Method

```
public class SDTwitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = this.serverSocket( port );

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    ServerSocket serverSocket( int port) {
        return new ServerSocket(port);
    }

    etc.
```

TestServer

```
public class TestServer extends SDTwitterServer {
   MockServerSocket testSocket;

   ServerSocket serverSocket( int port) {
      return testSocket;
   }
}
```

Other than using a different type of socket it performs the operations as the parent class

```
public class Tests extends Testcase {
   public void testLogin() {
      TestServer server = new TestServer();
      server.testSocket = new MockServerSocket("client command to login");
      server.run();
      assertTrue(server.testSocket.serverResponse() = "the correct response here");
   }
}
```

MockServerSocket

Returns a fake (Mock) client connection

Fakes client connection

- Does not use network
- Contains fixed requests
- Records server responses

Dependency Injection

```
public class SDTwitterServer {
    ServerSocket input;
    public SDTwitterServer(ServerSocket input) {
        this.input = input;
    }

    public void run(int port) throws IOException {
        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
}
```

Dependency Injection

"One object (or static method) supplies the dependencies of another object"
Wikipedia

Constructor injection
Setter injection
Interface injection

Effective Java

Effective Java

Book by Joshua Bloch

First Edition 2001
Second Edition 2008

Item 1. Consider Static Factory methods

Consider using static Factory methods instead of constructors

Java String class

```
public static String valueOf(boolean b) {
        return b ? "true" : "false";
    }


public static String valueOf(char c) {
    char data[] = {c};
    return new String(data, true);
}
```

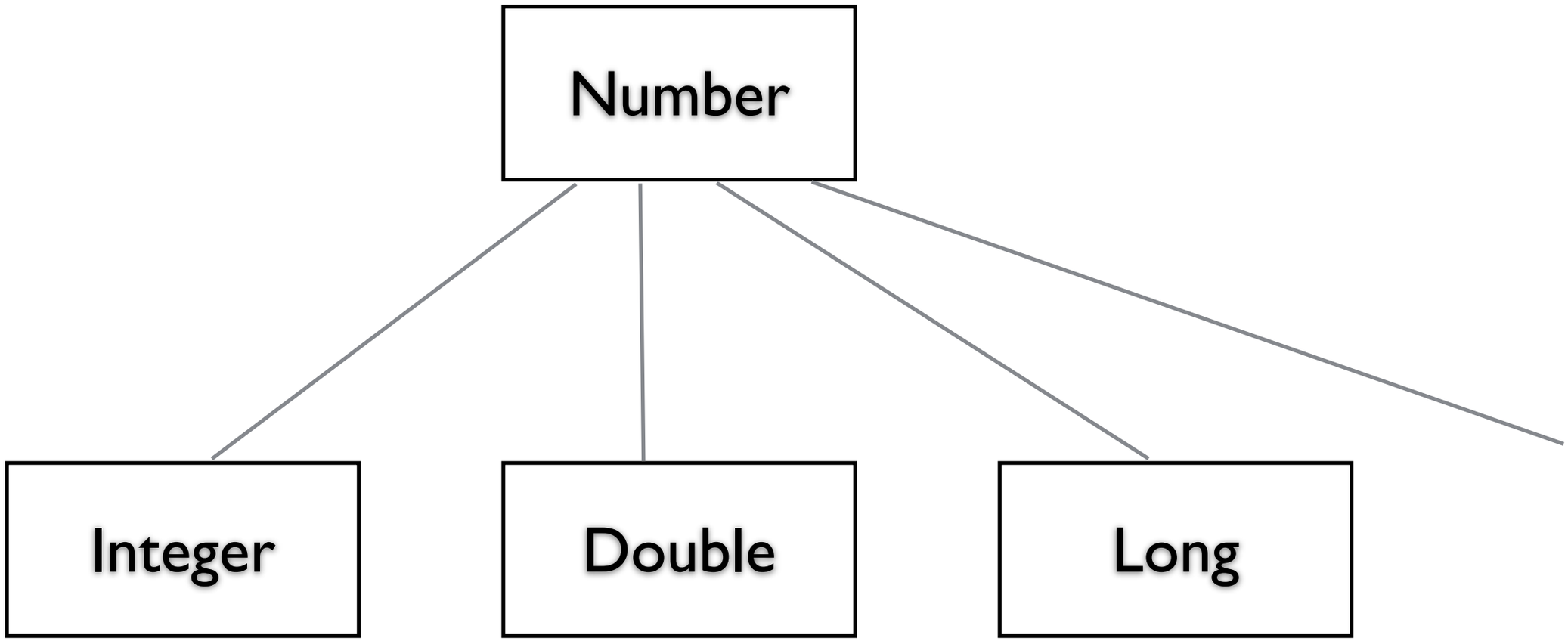
Advantages of Static Factory methods

They have names

Don’t need to create a new object each time

They can return an Object of any type

Java Boxing of Primitives




```
Integer x = new Integer( 5);
Boolean y = new Boolean( true);
```

Objective C Boxing of Primitives

Uses static factory methods in Number

```
Number x = Number.value(5);
Number y = Number.value(true);
```

Programmers only need to know Number class

Class Cluster

Smalltalk

No constructors

Just static factory methods

Item 12 Minimize accessibility

Rule of thumb

Make each class or member as inaccessible as possible

Item 13 Favor Immutability

Immutable objects are simple

Immutable objects are thread-safe

Immutable objects can be shared freely

Immutable objects are good building blocks for other objects

Item 13 Favor Immutability

Don’t provide any methods that modify the object (setters)

Ensure that no methods may be overridden

Make all fields final

Make all fields private

Ensure exclusive use to any mutable components

Make defensive copies of data provided/given to client

Item 24 Make Defensive Copies when Needed

```
public final class Period {
   private final Date start;
   private final Date end;

   public Period(Date start, Date End) {
      if (start.compareTo(end) > 0 )
         throw new IllegalArgumentException(start + " is after " + end);
      this.start = start;
      this.end = end;
   }

   public Date start() {
      return start;
   }
```

Item 24 Make Defensive Copies when Needed

```
public final class Period {
   private final Date start;
   private final Date end;

   public Period(Date start, Date End) {
      this.start = new Date(start.getTime());
      this.end = new Date(end.getTime());
      if (this.start.compareTo(this.end) > 0 )
         throw new IllegalArgumentException(start + " is after " + end);
   }

   public Date start() {
      return start.clone();
   }
```

Item 14 Favor Composition over Inheritance

Inheritance breaks encapsulation

Safe to use inheritance when

Superclass and subclass in same package

When superclass is designed for inheritance

Item 16 Prefer Interfaces to Abstract Classes

Existing classes can be modified to implement a new interface

Interfaces are ideal for defining mixins

Interfaces allow construction of nonhierarchical frameworks

Provide skeletal implementation class to go with nontrivial interface

Item 30 Know and use the Libraries

Item 32 Avoid strings if other types are better

String compoundKey = name + “#” + i.next();

What happens if “#” is in name?

Create CompoundKey class

Item 34 Refer to objects by their Interfaces

Your code will be more flexible

List subscribers = new Vector();

Vector subscribers = new Vector();

If no interface exists then ok to refer to object via class