CS3500: Object-Oriented Design
Spring 2014

Class 2
1.10.2014

Northeastern University
College *of* Computer and Information Science

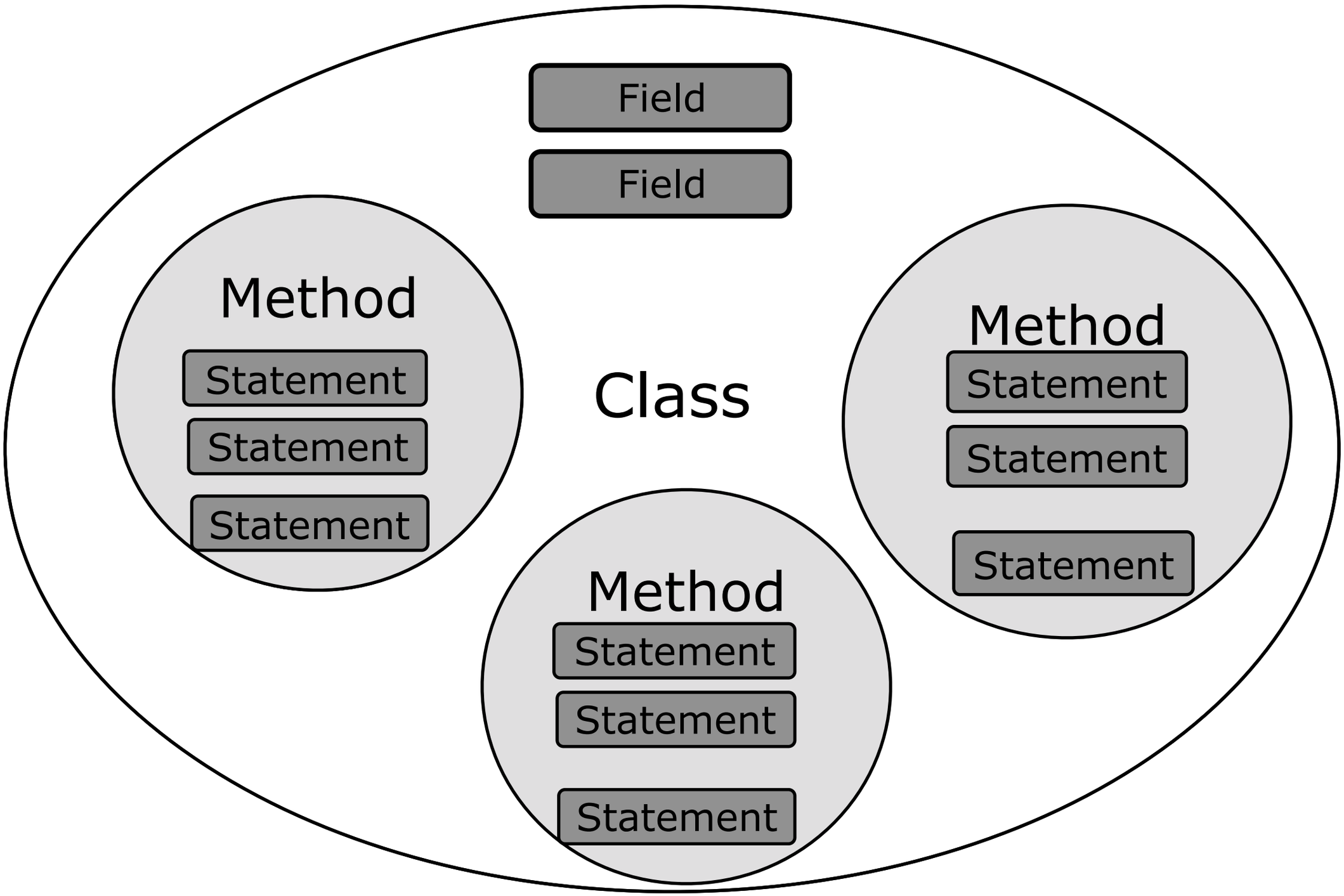
Java Syntax

- Sestoft is a great resource for Java syntax
- Java resources on the "Other resources" page of course site

Classes

- define collections of procedures
- define new data types

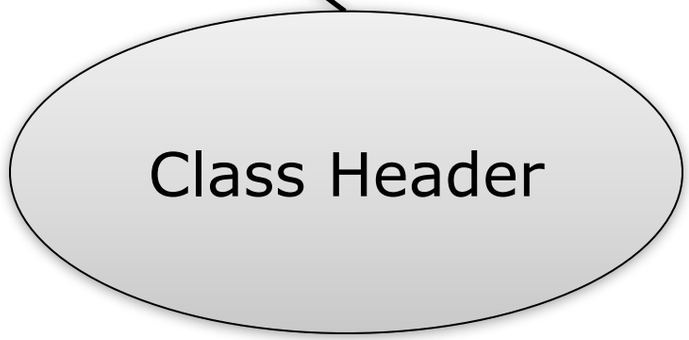
Java Classes



Modified from North Carolina State University Course Materials by S. Heckman & J. D. Young, 2009-2010

Class Syntax Template

```
public class <Name> {
    <field>
    <field>
    <method>
    …
    <method>
}
```



Modified from North Carolina State University Course Materials by S. Heckman & J. D. Young, 2009-2010

Class Declaration

[Sestoft, p.20]

class-modifiers `class C` extends-clause implements-clause {

  field-declarations

  constructor-declarations

  method-declarations

  class-declarations

  interface-declarations

  enum-type-declarations

  initializer-blocks

}

Class Header

```
<modifiers> class <Name>
      extends <...> implements <...>
```

```
public class Book
```

Interface Declaration

[Sestoft, p.60]

interface-modifiers `interface I` **extends-clause** {

field-descriptions

method-descriptions

class-declarations

interface-declarations

}

Method Declaration

[Sestoft, p.26]

```
method-modifiers return-type m(formal-list)
throws-clause method-body {

}
```

Method Header

```
<modifiers> <return-type> <method-name>(<parameters>)
```

```
public static void main(String [] args)
```

Field Declaration

[Sestoft, p.24]

field-modifiers type fieldname1, fieldname2, ...;

field-modifiers type field name = initializer, ... ;

Packages

What do we need in order to have an executable program?

Hello World!!

```
/**
 * This is an example class that
 * illustrates printing a message to the
 * screen
 * @author Jessica Young Schmidt jschmidt
 */
public class Hello {
   /**
    * main method which will print message
    * @param args command line arguments
    */
   public static void main(String [] args){
      System.out.println("Hello World!");
   }
}
```

- Define a class named `Hello`
- Define a method called `main`
  Main method required to start a Java program
- Defined statements of what the program should do
  In this case, print a string "Hello World!"

Modified from North Carolina State University Course Materials by S. Heckman & J. D. Young, 2009-2010

Write-Compile-Execute

- **Write**

  source code: instructions in a program – human readable (*.java*)

- **Compile**

  – translates program from one language to another

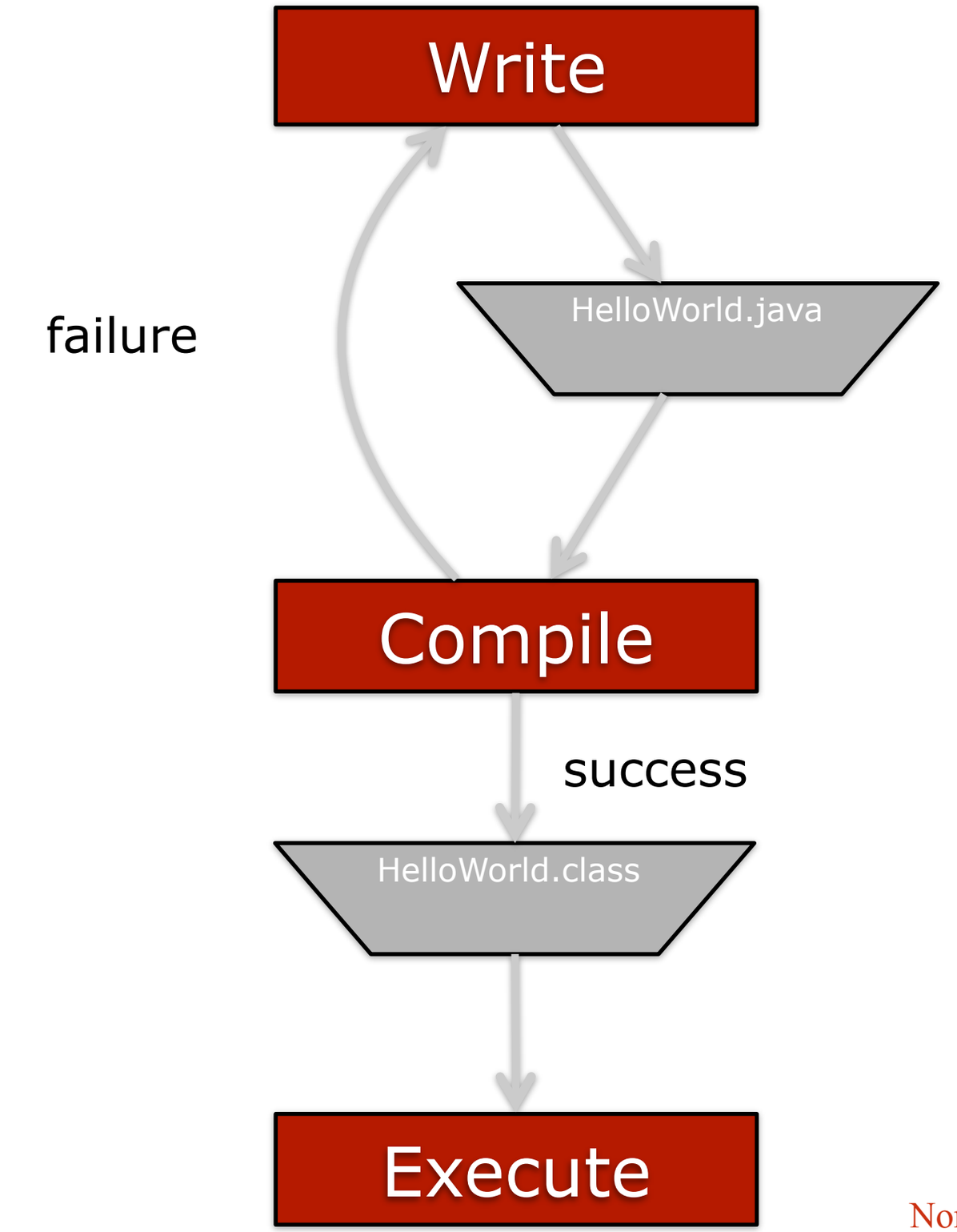
  – *byte code*: intermediate language that can be run by many computers (*.class*)

- **Execute**

  Uses the Java Virtual Machine (JVM) to run the program

Modified from North Carolina State University Course Materials by S. Heckman & J. D. Young, 2009-2010

Write-Compile-Execute



Modified from North Carolina State University Course Materials by S. Heckman & J. D. Young, 2009-2010

Static Methods

Mutation

JUnit

- "JUnit is an open source unit testing framework for automatically unit testing Java programs. JUnit provides functionality for writing and running unit test cases for a project under development, and is helpful when doing Test-Driven Development (TDD)."
(http://agile.csc.ncsu.edu/SEMaterials/tutorials/junit/junit_tutorial_jazz.html)

- http://web-cat.org/sigcse2012/cheatsheet1.html

equals **and** hashCode

Abstraction Barrier

- Every piece of software has, or should have, an abstraction barrier that divides the world into two parts: clients and implementors.
  - The clients are those who use the software. They do not need to know how the software works.
  - The implementors are those who build it. They need to know how the software works.

Abstract Data Type (ADT)

- What is an ADT?
  - set of data
  - set of operations
  - description of what operations do
- Within this course, when discuss ADTs, we will discuss them using:
  - a signature: names of operations and types
  - a specification: agreement between client and implementors

StackInt Signature

```
public static methods:
    empty    :                    -->  StackInt
    push     : StackInt x int     -->  StackInt
    isEmpty  : StackInt           -->  boolean
    top      : StackInt           -->  int
    pop      : StackInt           -->  StackInt
    size     : StackInt           -->  int
public dynamic methods:
    toString :                    -->  String
    equals   : Object             -->  boolean
    hashCode :                    -->  int
```

StackInt
Algebraic Specifications

```
StackInt.isEmpty(empty())  =  true

StackInt. isEmpty(push (s, n))  =  false


StackInt. top(push (s, n))  =  n


StackInt. pop(push (s, n))  =  s


StackInt. size(empty())  =  0

StackInt. size(push (s, n))  =  1 + StackInt.size(s)
```

Recipe Implementation of StackInt

StackInt Recipe Implementation

Determine which operations of the ADT are basic creators, and which are other operations.

```
empty:                      -> StackInt
push:    StackInt x int     -> StackInt
isEmpty: StackInt           -> boolean
top:     StackInt           -> int
pop:     StackInt           -> StackInt
size:    StackInt           -> int
```

Hints:

- Except for the basic creators, each operation is specified by one or more equations.

- The basic creators are used as arguments in the left side of the equations.

```
isEmpty (empty())   =  true
isEmpty (push (s, n))   =  false
top (push (s, n))   =  n
pop (push (s, n))   =  s
size (empty())   =  0
size (push (s, n))   =  1 + size (s)
```

StackInt Recipe Implementation

Determine which operations of the ADT are basic creators, and which are other operations.

```
empty:                          -> StackInt
push:     StackInt x int        -> StackInt
isEmpty:  StackInt              -> boolean
top:      StackInt              -> int
pop:      StackInt              -> StackInt
size:     StackInt              -> int
```

Hints:

- Except for the basic creators, each operation is specified by one or more equations.

- The basic creators are used as arguments in the left side of the equations

```
isEmpty (empty())    =  true
isEmpty (push (s, n))  =  false
top (push (s, n))    =  n
pop (push (s, n))    =  s
size (empty())    =  0
size (push (s, n))   =  1 + size (s)
```

The basic creators are empty and push.

StackInt Recipe Implementation

Define an abstract class
named T.

StackInt Recipe Implementation

Define an abstract class named T.

```
/**
 * StackInt represents a stack
 * of ints
 */
public abstract class StackInt {
}
```

StackInt Recipe Implementation

For each basic creator c of the ADT, define a concrete subclass of T.

StackInt Recipe Implementation

For each basic creator c of the ADT, define a concrete subclass of T.

```
/**
 * Empty represents a stack that
 * contains no ints
 */
class Empty extends StackInt {}

/**
 * Push represents a non-empty
 * stack of ints
 */
class Push extends StackInt {}
```

StackInt Recipe Implementation

For each concrete subclass, declare instance variables with the same types and names as the arguments that are passed to the corresponding basic creator c.

```
empty:                          -> StackInt

push:  StackInt x int -> StackInt
```

StackInt Recipe Implementation

For each concrete subclass, declare instance variables with the same types and names as the arguments that are passed to the corresponding basic creator c.

```
empty:                      -> StackInt
push:  StackInt x int -> StackInt


/**
 * Empty represents a stack that
 * contains no ints
 */
class Empty extends StackInt {}
/**
 * Push represents a non-empty
 * stack of ints
 */
class Push extends StackInt {
  /** other elements of this StackInt */
  StackInt s;
  /** topmost element of this StackInt */
  int n;
}
```

StackInt Recipe Implementation

For each concrete subclass, define a Java constructor that takes the same arguments as the basic creator c and stores those arguments into the instance variables.

```
empty:                          -> StackInt

push:  StackInt x int -> StackInt
```

StackInt Recipe Implementation

For each concrete subclass, define a Java constructor that takes the same arguments as the basic creator c and stores those arguments into the instance variables.

```
empty:                      -> StackInt
push:  StackInt x int -> StackInt
```

```
/**
 * Constructor for Empty
 */
Empty () { }
```

```
/**
 * Constructor for Push
 * @param s the stack without topmost
 *          element
 * @param n is the topmost element of the
 *          stack
 */
Push (StackInt s, int n) {
    this.s = s;
    this.n = n;
}
```

StackInt Recipe Implementation

For each operation f of the ADT that is not a basic creator, declare an abstract method f in the abstract class T.

The explicit arguments of the abstract method f should be the same as the arguments of the operation f, except the abstract method f will take one less argument than f. That missing argument will be the receiver for calls to the abstract method f (so that argument will still be available to the abstract method f, as the value of Java's special variable this). That missing argument should be of type T. If the operation f has more than one argument of type T, so you have some choice as to which argument to omit, then you should omit the argument that discriminates between the various equations for f in the algebraic specification.

```
empty:                    -> StackInt
push:    StackInt x int   -> StackInt
isEmpty: StackInt         -> boolean
top:     StackInt         -> int
pop:     StackInt         -> StackInt
size:    StackInt         -> int
```

```
isEmpty (empty())   =  true
isEmpty (push (s, n))   =  false
top (push (s, n))   =  n
pop (push (s, n))   =  s
size (empty())   =  0
size (push (s, n))   =  1 + size (s)
```

StackInt Recipe Implementation

For each operation f of the ADT that is not a basic creator, declare an abstract method f in the abstract class T.

The explicit arguments of the abstract method f should be the same as the arguments of the operation f, except the abstract method f will take one less argument than f. That missing argument will be the receiver for calls to the abstract method f (so that argument will still be available to the abstract method f, as the value of Java's special variable this). That missing argument should be of type T. If the operation f has more than one argument of type T, so you have some choice as to which argument to omit, then you should omit the argument that discriminates between the various equations for f in the algebraic specification.

```
empty:                    -> StackInt
push:     StackInt x int  -> StackInt
isEmpty:  StackInt        -> boolean
top:      StackInt        -> int
pop:      StackInt        -> StackInt
size:     StackInt        -> int
```

```
isEmpty (empty())   =  true
isEmpty (push (s, n))   =  false
top (push (s, n))   =  n
pop (push (s, n))   =  s
size (empty())   =  0
size (push (s, n))   =  1 + size (s)
```

StackInt Recipe Implementation

For each operation f of the ADT that is not a basic creator, declare an abstract method f in the abstract class T.

The explicit arguments of the abstract method f should be the same as the arguments of the operation f, except the abstract method f will take one less argument than f. That missing argument will be the receiver for calls to the abstract method f (so that argument will still be available to the abstract method f, as the value of Java's special variable this). That missing argument should be of type T. If the operation f has more than one argument of type T, so you have some choice as to which argument to omit, then you should omit the argument that discriminates between the various equations for f in the algebraic specification.

```
empty:                    -> StackInt
push:    StackInt x int -> StackInt
isEmpty: StackInt          -> boolean
top:     StackInt          -> int
pop:     StackInt          -> StackInt
size:    StackInt          -> int
```

```
abstract boolean isEmptyMethod ();
abstract int topMethod ();
abstract StackInt popMethod ();
abstract int sizeMethod ();
```

StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.

- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named "this".) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be: return x1.f (x2, x3, x4);

```
empty:                    -> StackInt
push:    StackInt x int   -> StackInt
isEmpty: StackInt         -> boolean
top:     StackInt         -> int
pop:     StackInt         -> StackInt
size:    StackInt         -> int
```

StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- **If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.**
- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named "this".) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be: return x1.f (x2, x3, x4);

```
empty:                      -> StackInt
push:    StackInt x int     -> StackInt
isEmpty: StackInt           -> boolean
top:     StackInt           -> int
pop:     StackInt           -> StackInt
size:    StackInt           -> int
```

StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- **If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.**

- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named "this".) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be: return x1.f (x2, x3, x4);

```
empty:                      -> StackInt
push:    StackInt x int     -> StackInt
isEmpty: StackInt           -> boolean
top:     StackInt           -> int
pop:     StackInt           -> StackInt
size:    StackInt           -> int
```

```
public static StackInt empty () {
  return new Empty ();
}

public static StackInt
        push (StackInt s, int n) {
  return new Push (s, n);
}
```

StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.
- **If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named "this".) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be: return x1.f (x2, x3, x4);**

```
empty:                    -> StackInt
push:    StackInt x int   -> StackInt
isEmpty: StackInt         -> boolean
top:     StackInt         -> int
pop:     StackInt         -> StackInt
size:    StackInt         -> int
```

StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.
- **If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named "this".) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be: return x1.f (x2, x3, x4);**

```
empty:                    -> StackInt
push:    StackInt x int   -> StackInt
isEmpty: StackInt         -> boolean
top:     StackInt         -> int
pop:     StackInt         -> StackInt
size:    StackInt         -> int
```

```
public static boolean isEmpty (StackInt s) {
  return s.isEmptyMethod();
}

public static int top (StackInt s) {
  return s.topMethod();
}

public static StackInt pop (StackInt s) {
  return s.popMethod();
}

public static int size (StackInt s) {
   return s.sizeMethod();
}
```

StackInt Recipe Implementation

For each concrete subclass C of T, define all of the abstract methods that were declared within the abstract base class for T. (These definitions define dynamic (as opposed to static) methods.)

For each abstract method f that is defined within a subclass C, the body of f should return the value specified by the algebraic specification for the case in which Java's special variable this will be an instance of the class C.

-If the algebraic specification contains only one equation that describes the result of an operation f applied to an instance of C, and that equation has no side conditions, then the body of the dynamic method f should return the value expressed by the right hand side of that equation.

-If the algebraic specification does not contain any equations that describe the result of an operation f applied to an instance of C, then the body of the dynamic method f should throw a RuntimeException such as an IllegalArgumentException.

-(Note that other conditions are listed in the recipe.}

```
empty:                    -> StackInt
push:    StackInt x int   -> StackInt
isEmpty: StackInt         -> boolean
top:     StackInt         -> int
pop:     StackInt         -> StackInt
size:    StackInt         -> int
```

```
isEmpty (empty())   =  true
isEmpty (push (s, n))   =  false
top (push (s, n))   =  n
pop (push (s, n))   =  s
size (empty())   =  0
size (push (s, n))   =  1 + size (s)
```

StackInt Recipe Implementation

For each concrete subclass C of T, define all of the abstract methods that were declared within the abstract base class for T. (These definitions define dynamic (as opposed to static) methods.)

For each abstract method f that is defined within a subclass C, the body of f should return the value specified by the algebraic specification for the case in which Java's special variable this will be an instance of the class C.

- If the algebraic specification contains only one equation that describes the result of an operation f applied to an instance of C, and that equation has no side conditions, then the body of the dynamic method f should return the value expressed by the right hand side of that equation.
- If the algebraic specification does not contain any equations that describe the result of an operation f applied to an instance of C, then the body of the dynamic method f should throw a RuntimeException such as an IllegalArgumentException.
- (Note that other conditions are listed in the recipe.}

```
isEmpty (empty())  =  true
isEmpty (push (s, n))  =  false
top (push (s, n))  =  n
pop (push (s, n))  =  s
size (empty())  =  0
size (push (s, n))  =  1 + size (s)

class Empty extends StackInt {

    Empty () { }

    boolean isEmptyMethod () {
        return true;
    }

    int topMethod () {
        String msg1
            = "attempted to compute the top of an empty "
              + "StackInt";
        throw new RuntimeException (msg1);
    }

    StackInt popMethod () {
        String msg1
            = "attempted to pop from an empty StackInt";
        throw new RuntimeException (msg1);
    }

    int sizeMethod () {
        return 0;
    }
}
```

StackInt Recipe Implementation

For each concrete subclass C of T, define all of the abstract methods that were declared within the abstract base class for T. (These definitions define dynamic (as opposed to static) methods.)

For each abstract method f that is defined within a subclass C, the body of f should return the value specified by the algebraic specification for the case in which Java's special variable this will be an instance of the class C.

- If the algebraic specification contains only one equation that describes the result of an operation f applied to an instance of C, and that equation has no side conditions, then the body of the dynamic method f should return the value expressed by the right hand side of that equation.
- If the algebraic specification does not contain any equations that describe the result of an operation f applied to an instance of C, then the body of the dynamic method f should throw a RuntimeException such as an IllegalArgumentException.
- (Note that other conditions are listed in the recipe.}

```
isEmpty (empty())  =  true
isEmpty (push (s, n))  =  false
top (push (s, n))  =  n
pop (push (s, n))  =  s
size (empty())  =  0
size (push (s, n))  =  1 + size (s)
```

```
class Push extends StackInt{
    StackInt s;
    int n;

    Push(StackInt s, int n){
        this.s = s;
        this.n = n;
    }

boolean isEmptyMethod(){
    return false;
}

int topMethod(){
    return n;
}

StackInt popMethod(){
    return s;
}

int sizeMethod(){
    return 1 + StackInt.size(s);
}
}
```

Abstract Class vs. Concrete Class

- Concrete class: full implementation of the type.
- Abstract classes: at most a partial implementation of the type

Assignment 1

- Due: Tuesday, January 14, 2014 at 11:59 pm
- Recipe implementation in Java of the MySet ADT
- MySet is an immutable abstract data type whose values represent finite sets with elements of type Long.

Recipe Assumption

The operations of the ADT are to be implemented as static methods of a class named T.

`equals` method

- If s1 is a value of the MySet ADT, then s1.equals(null) returns false.
- If s1 is a value of the MySet ADT, but x is not, then s1.equals(x) returns false.
- If s1 and s2 are values of the MySet ADT, then s1.equals(s2) if and only if for every Long k

  MySet.contains(s1, k) if and only if MySet.contains(s2, k)

Theorem

The following conditions are equivalent:

1. for every Long k, MySet.contains (s1, k) if and only if MySet.contains (s2, k)

2. MySet.isSubset (s1, s2) and MySet.isSubset (s2, s1)

`hashCode` **method**

If s1 and s2 are values of the MySet ADT, and s1.equals(s2), then s1.hashCode() == s2.hashCode().