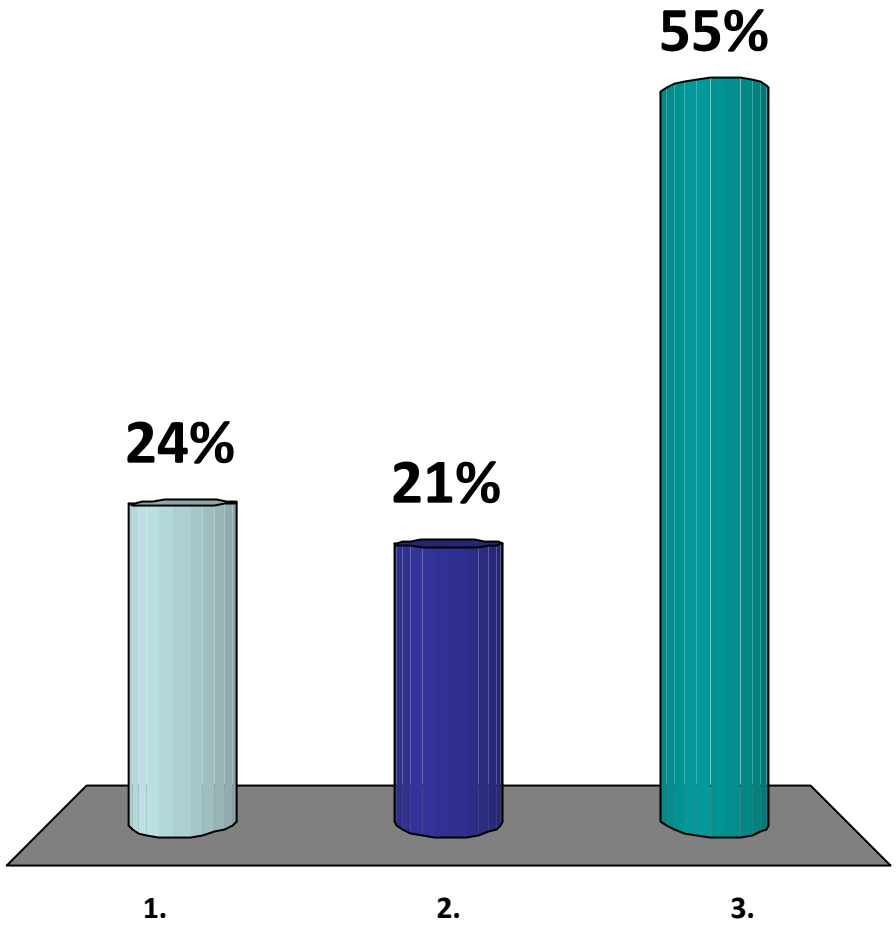
CS2020
Data Structures and Algorithms

Welcome!

Did you remember your clicker?

✓1. Yes
2. No
3. Dragons?



Discussion Groups / Problem Sessions

Scheduling

- CORS results announced
- Reorganization in progress…
  - Re-check your allocation.
  - It may have changed.

Originally:
- 1 tutorial had 1 student
- ~13 students unassigned

If you are still unallocated:

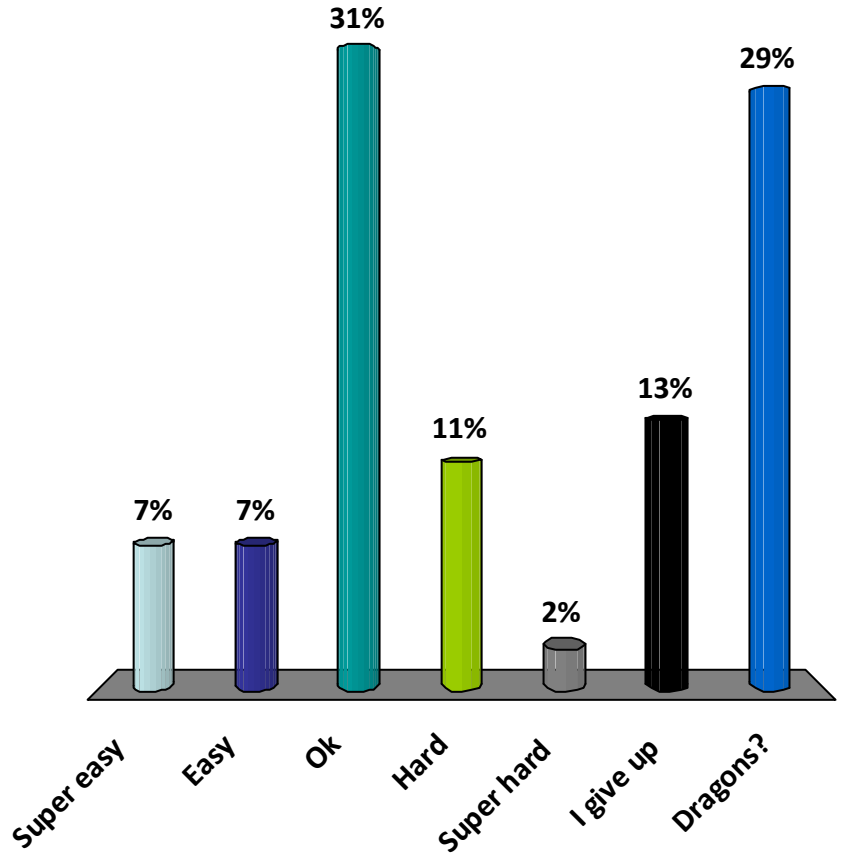
- E-mail me when you are free.

Problem Set 1

Due: Tuesday, 11:59pm

How was problem set 1?

a. Super easy
b. Easy
c. Ok
d. Hard
e. Super hard
f. I give up
g. Dragons?



Problem Set 2

Due: Tuesday, 11:59pm

Simpler self-contained problem directly implementing something from class.

One "exercise":

- Binary search (Friday's class)

Two problems:

- List management (today's class)
- Herbert the Robot (Friday's class)

Today's Plan

Abstract data types

- Bags
- Lists
- Stacks
- Queues

Java

- Generics
- Inheritance
- Polymorphism

Object Oriented Paradigm

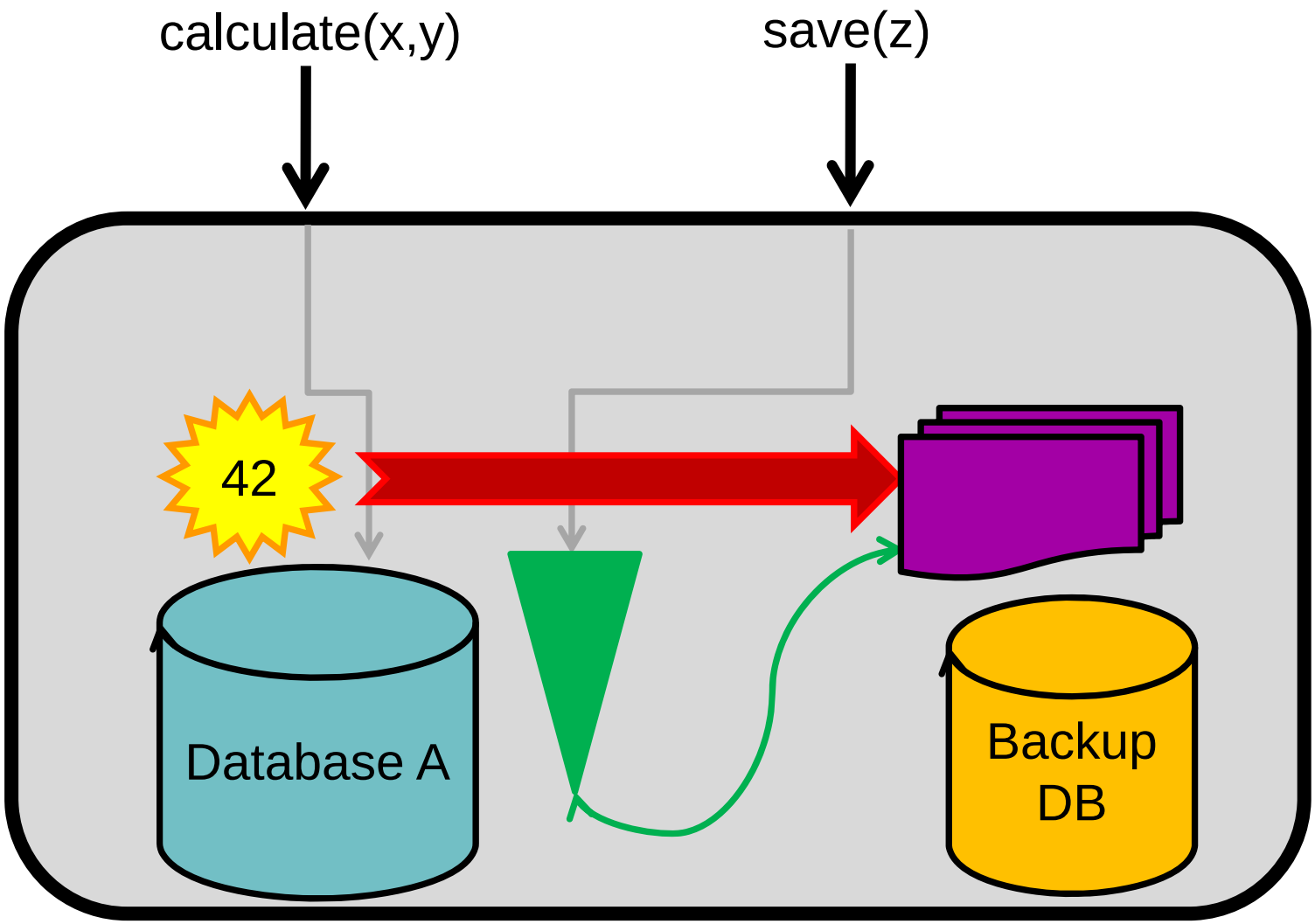
Encapsulation

Inheritance

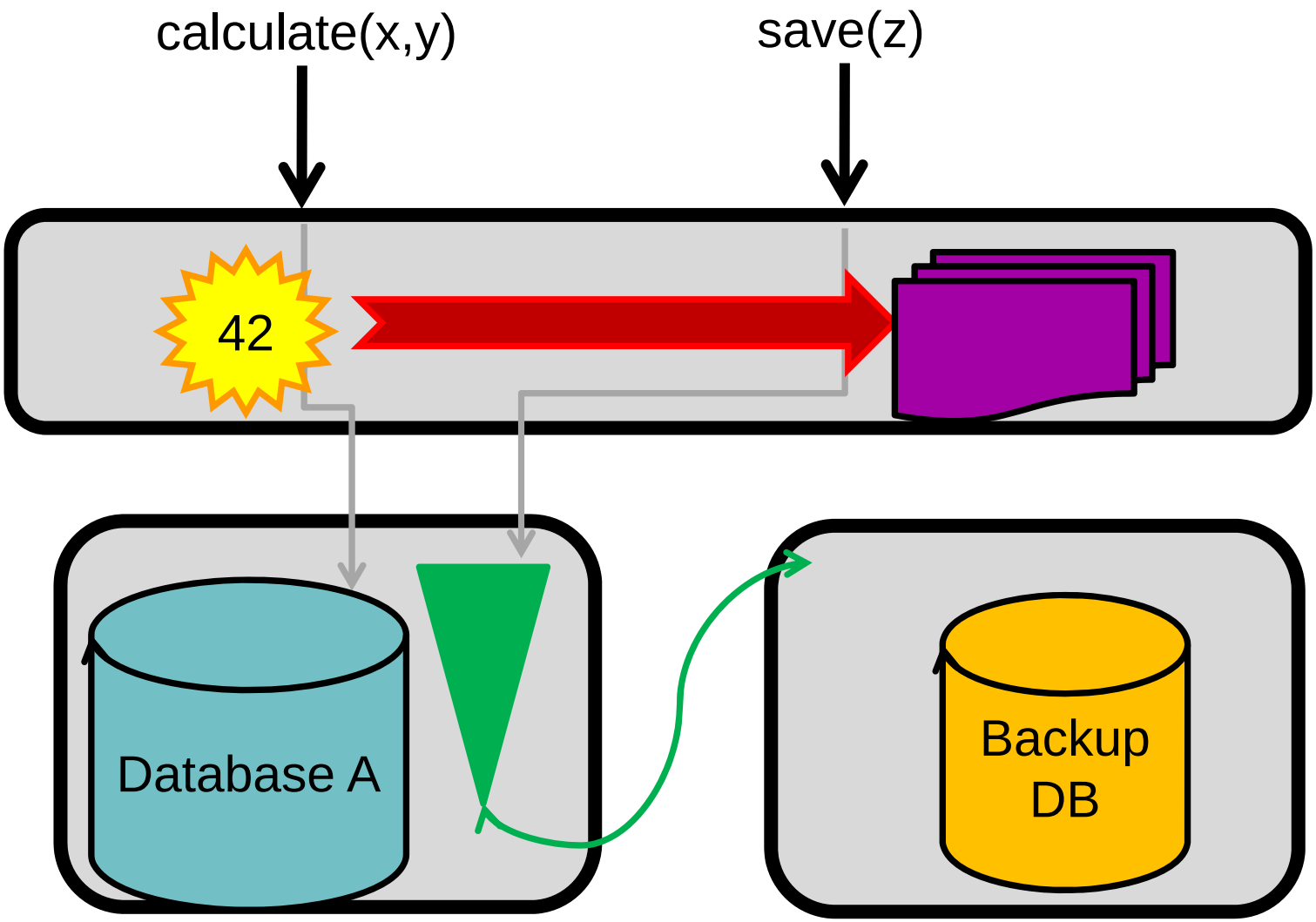
Polymorphism

Abstraction

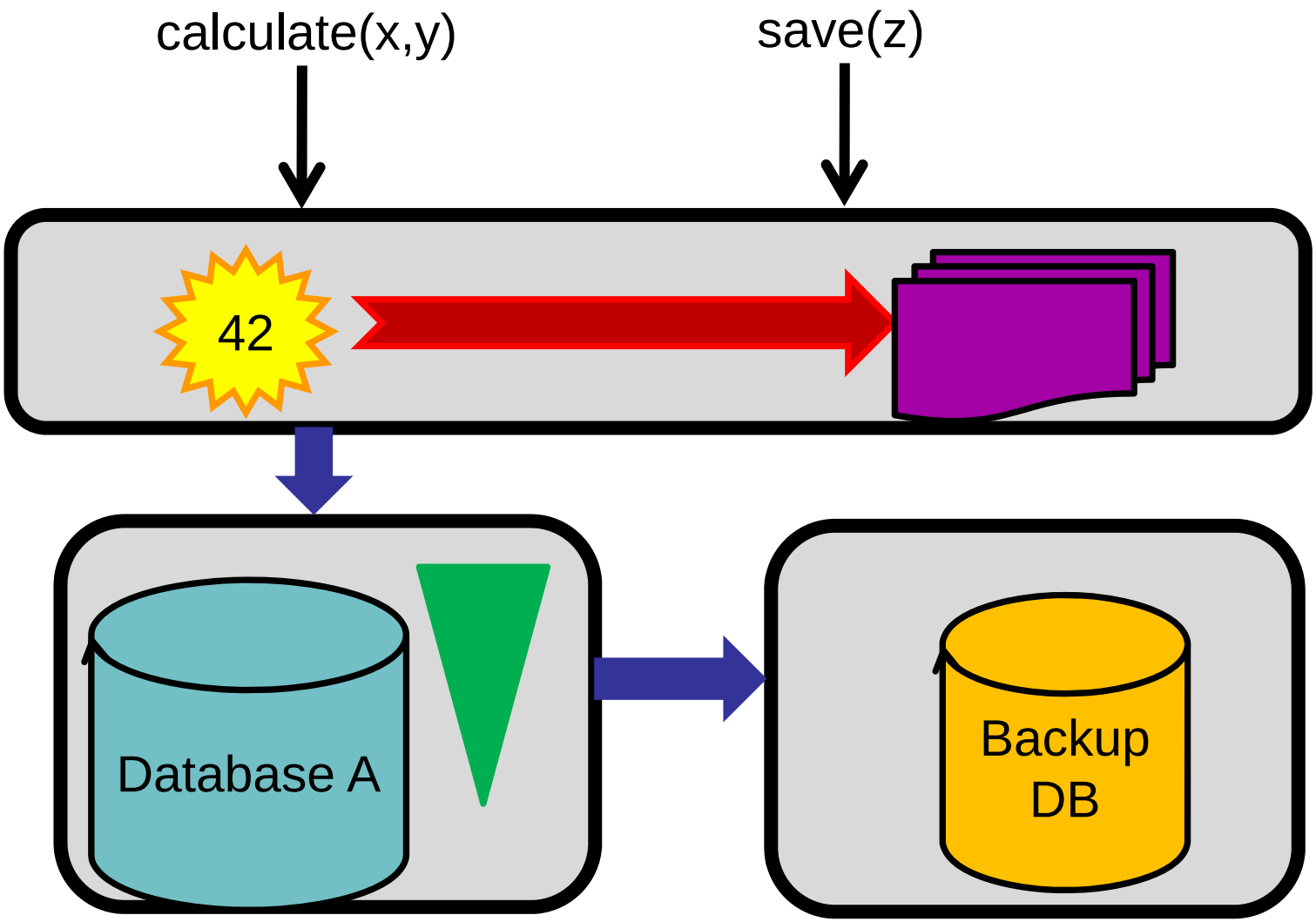
Abstraction



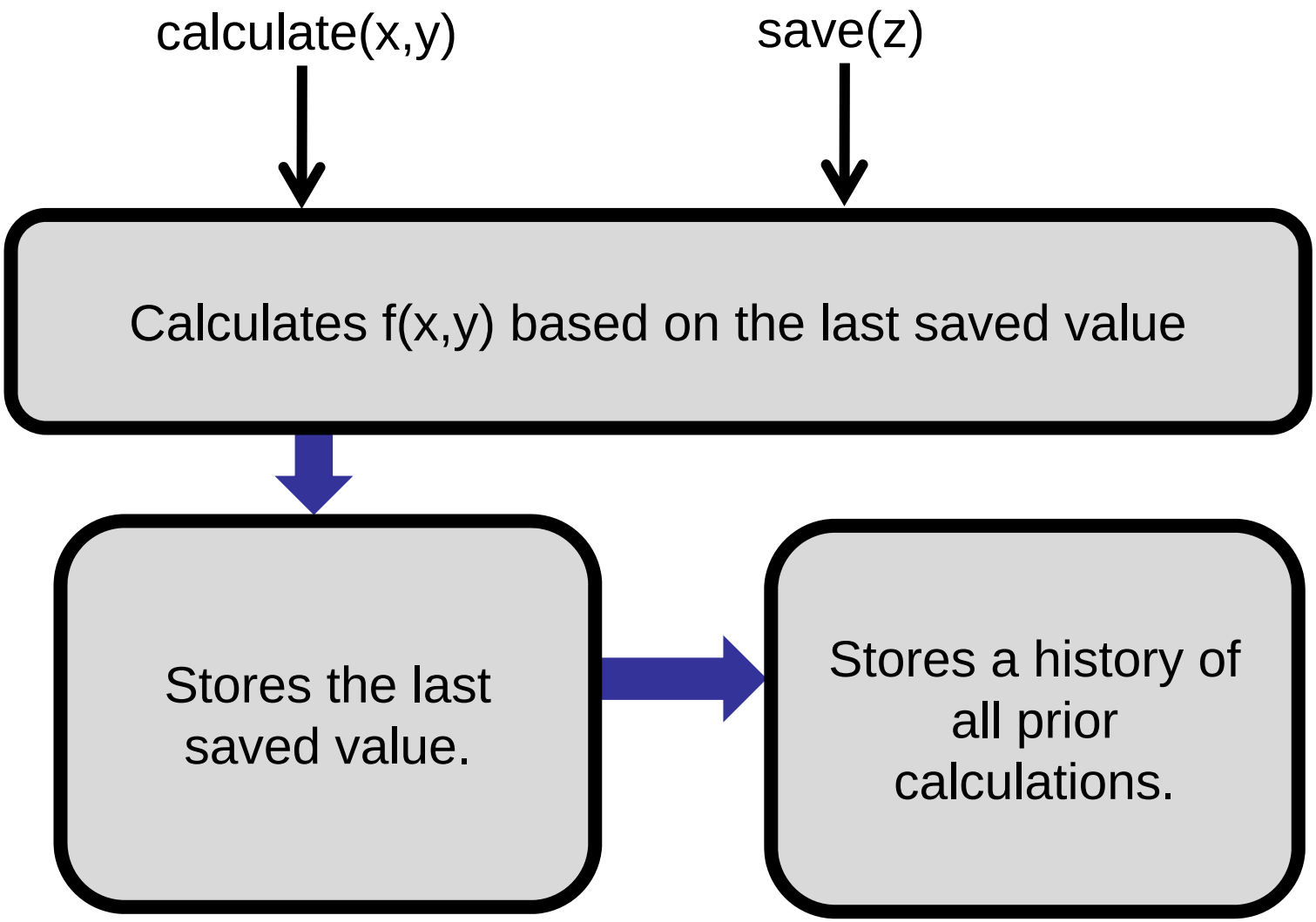
Abstraction



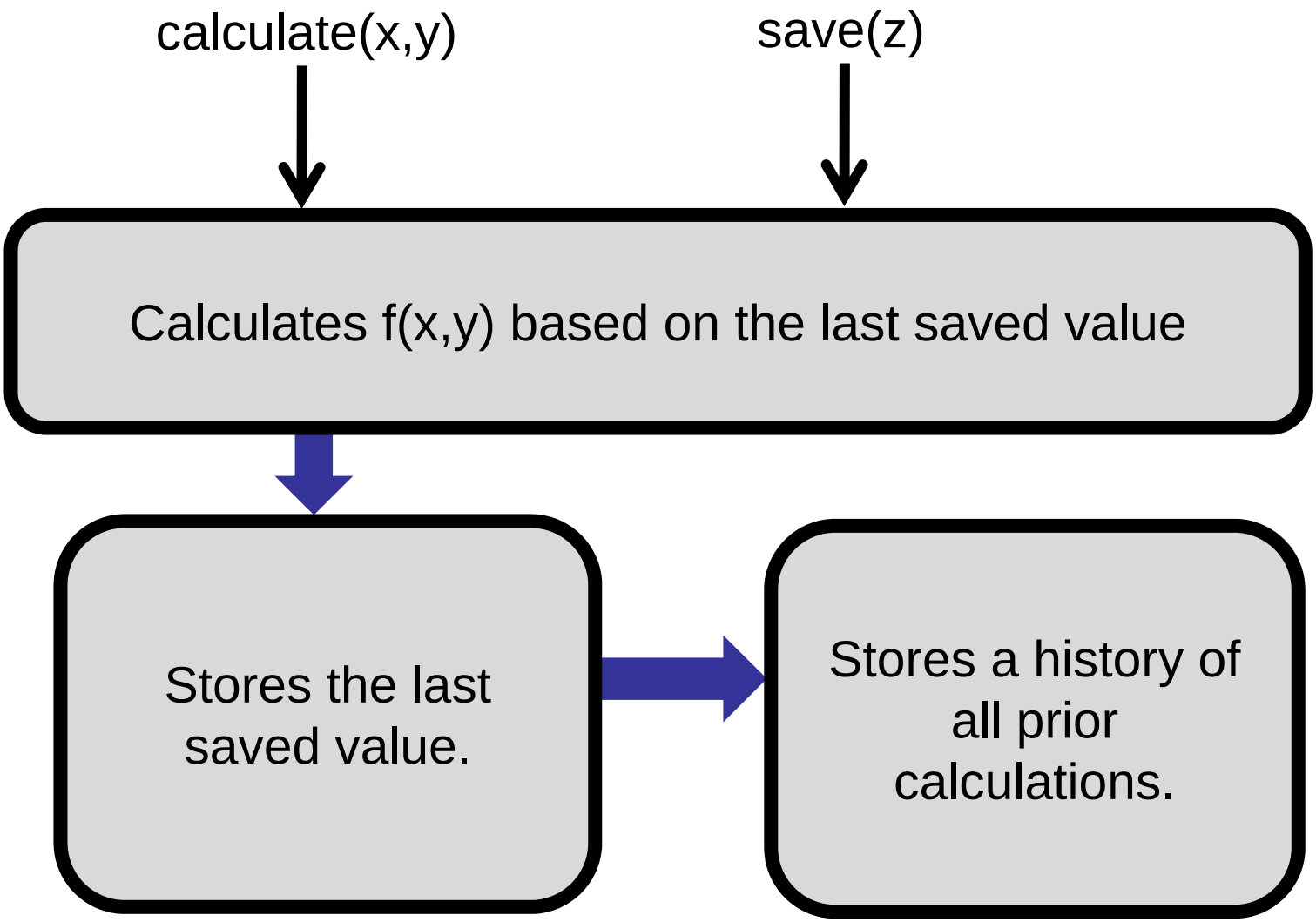
Abstraction



Abstraction



Top Down Design



Abstraction

Software engineering

- Divide problem into components.
- Define *interface* between components.
- Assign one team to build each component.
- (Recurse.)

- Top down design: get the big idea first, then figure out how to implement it.

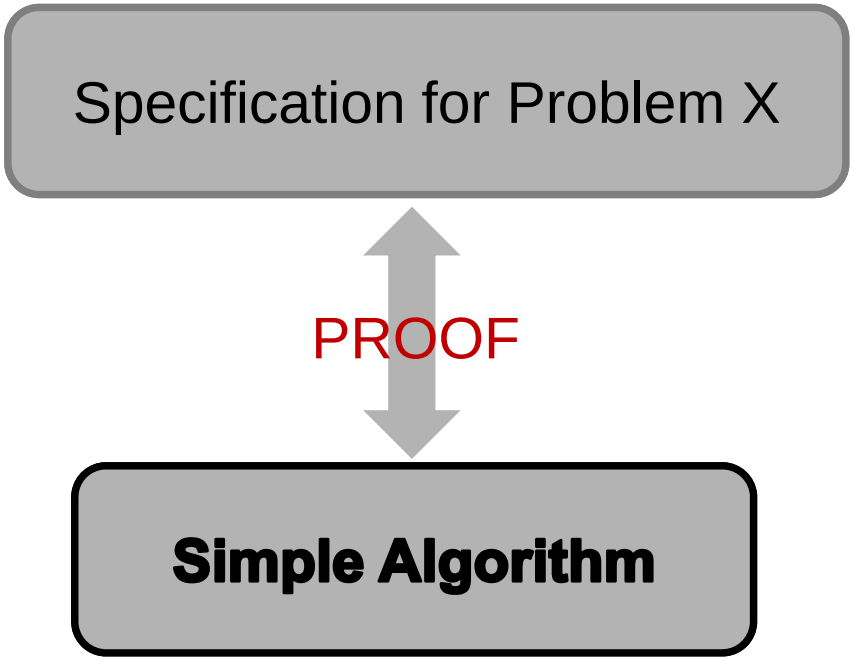
Abstraction

Algorithm design

- Divide problem into components.
- Define *interface* between components.
- Solve each problem separately.
- (Recurse.)
- Combine solutions.

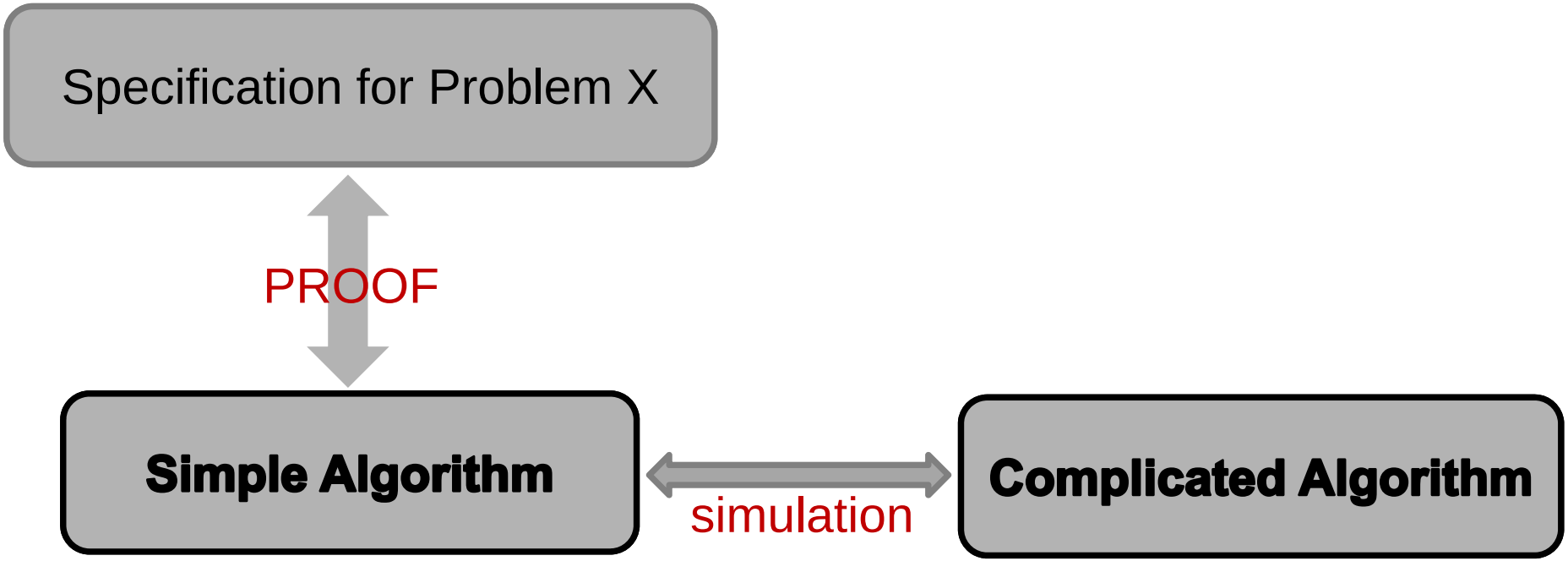
Abstraction

Proving an algorithm correct



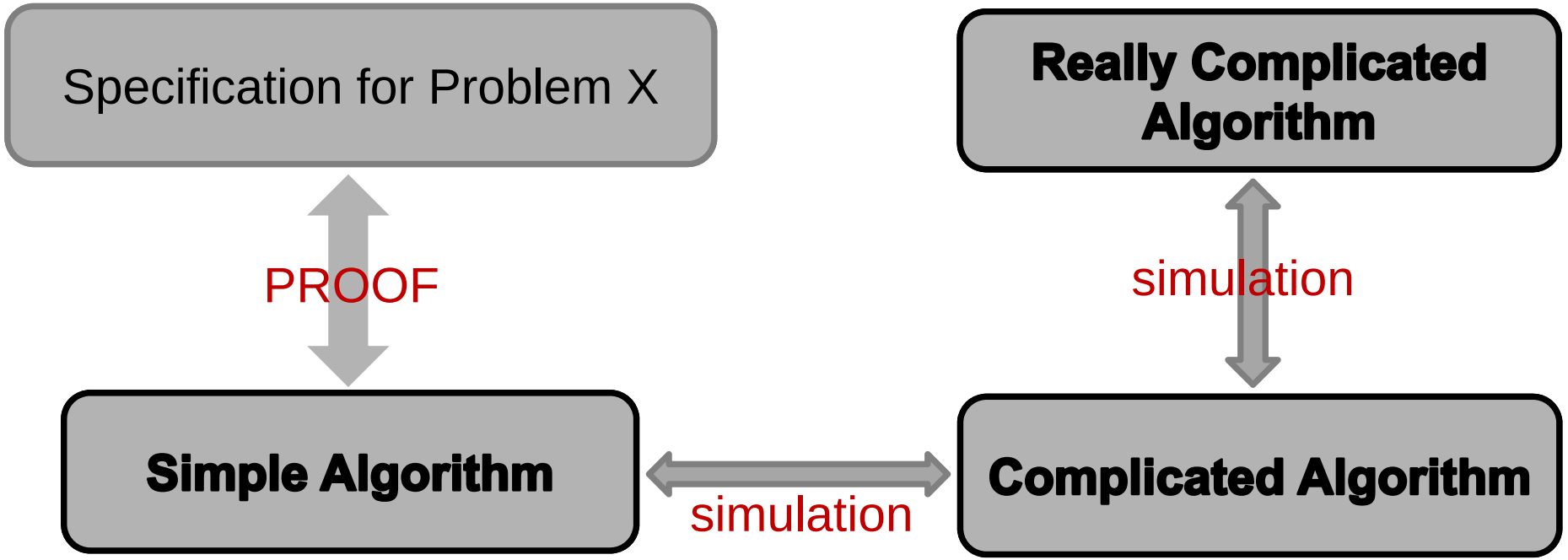
Abstraction

Proving an algorithm correct



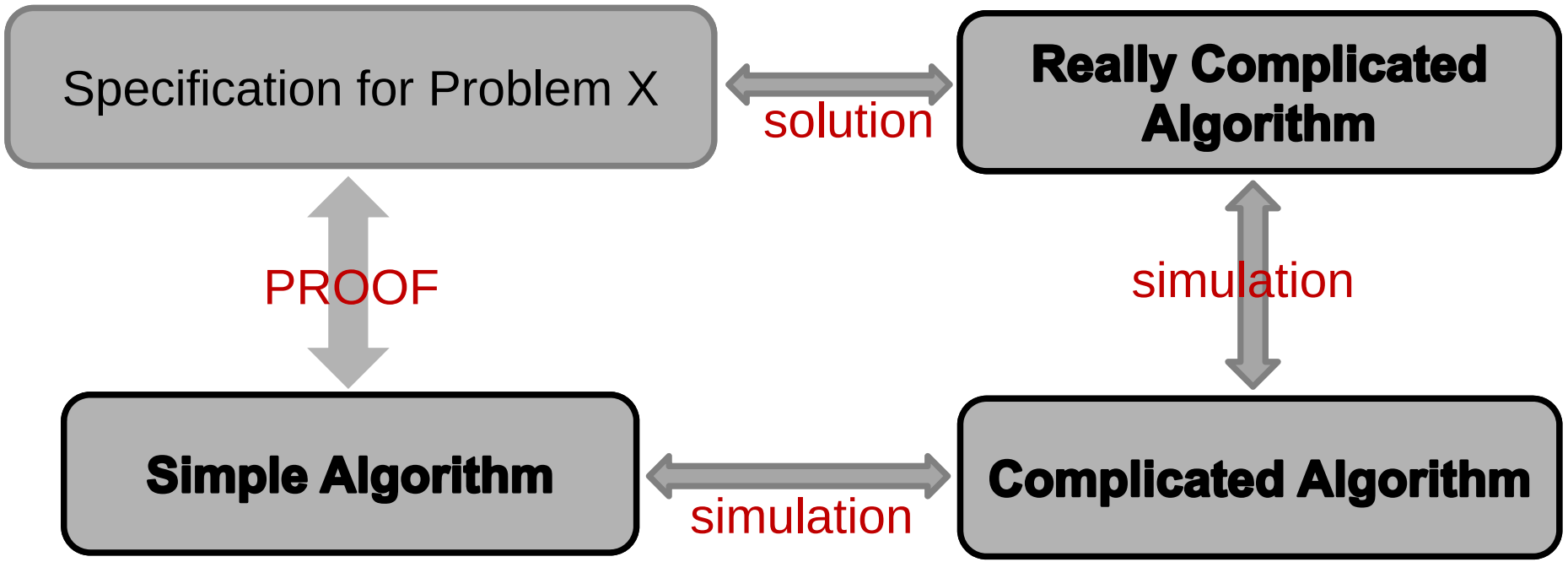
Abstraction

Proving an algorithm correct



Abstraction

Proving an algorithm correct



Abstraction

Key ideas

- Separate interface and implementation
- Hide implementation details
- Modularity: implement/analyze components separately

Abstract Data Types (Not Java specific.)

Specification:

- Interface
- Behavior

Implementation:

- Algorithm
- State

Abstract Data Types

Bag (of integers)

Interface:

```
void add(int x)
int remove()
boolean isEmpty()
```

Behavior:

- add(x) : adds an item to the bag
- remove() : removes an arbitrary item from the bag

Abstract Data Types

List

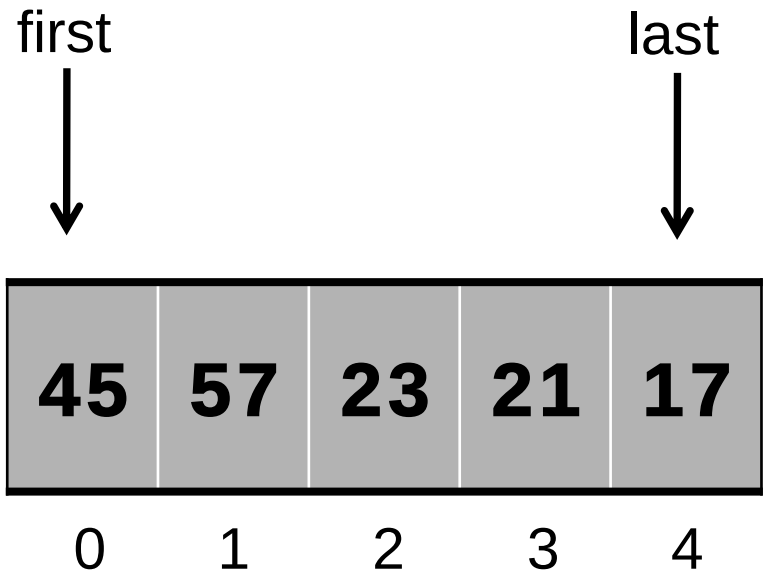
Interface:

Abstract Data Types

List

Interface:

```
void append(int x)
void prepend(int x)
void put(int x, int slot)
void remove(int x)
int getFirst()
int getLast()
int get(int slot)
boolean isEmpty()
```

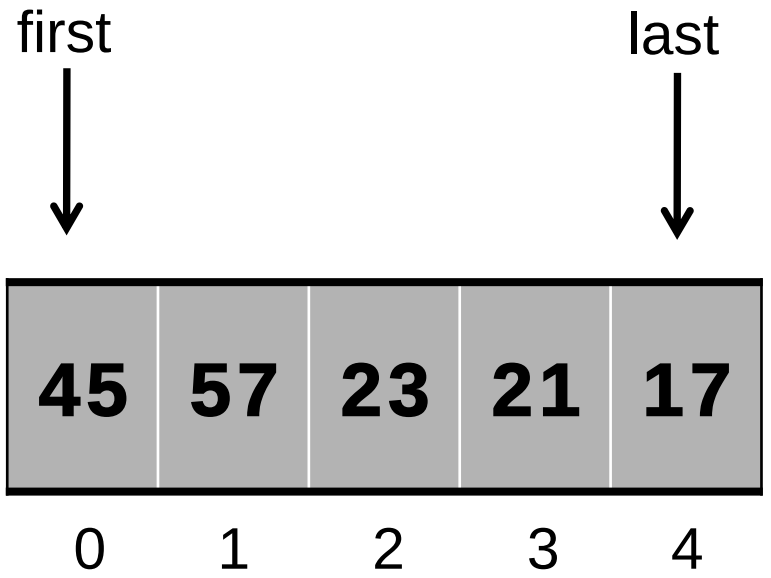


Abstract Data Types

interface java.util.List

Interface:

```
void add(int x)
void addAll(Collection c)
void clear()
void contains(int x)
void isEmpty()
int remove()
int set()
```

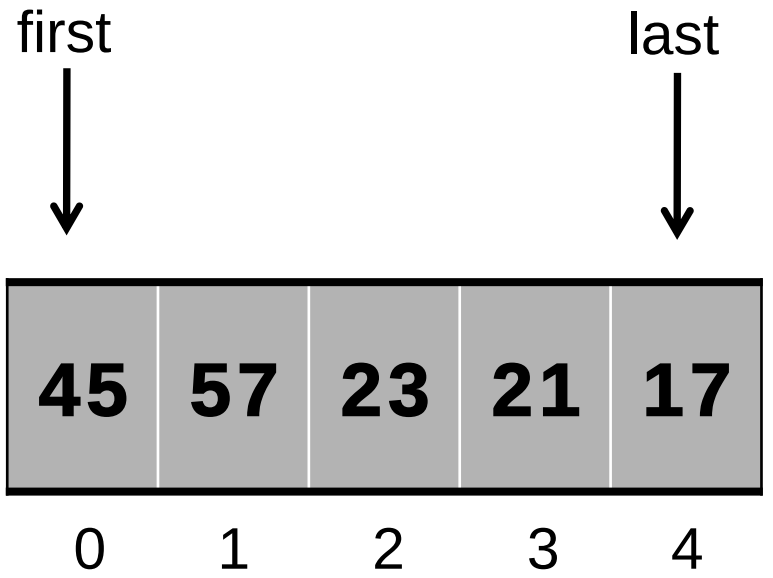


Abstract Data Types

interface java.util.List

Java implementations:

- `java.util.ArrayList`
- `java.util.Vector`
- `java.util.LinkedList`



Abstract Data Types

Stack

Interface:

- void push(element x)
- element pop()

Behavior: (LIFO: last-in, first-out)

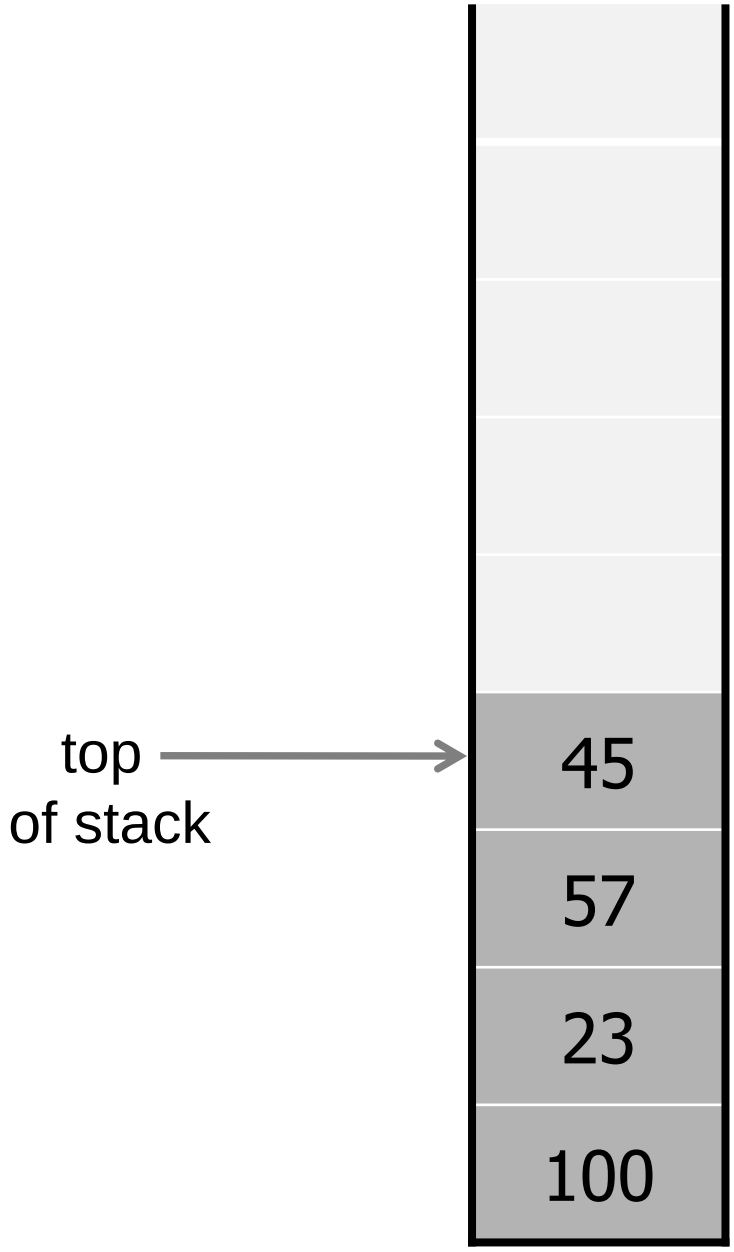
- push(x) : adds element x to the stack
- pop() : removes the mostly recently added element and returns it

Abstract Data Types

Stack

Interface:

- void push(element x)
- element pop()
- empty()

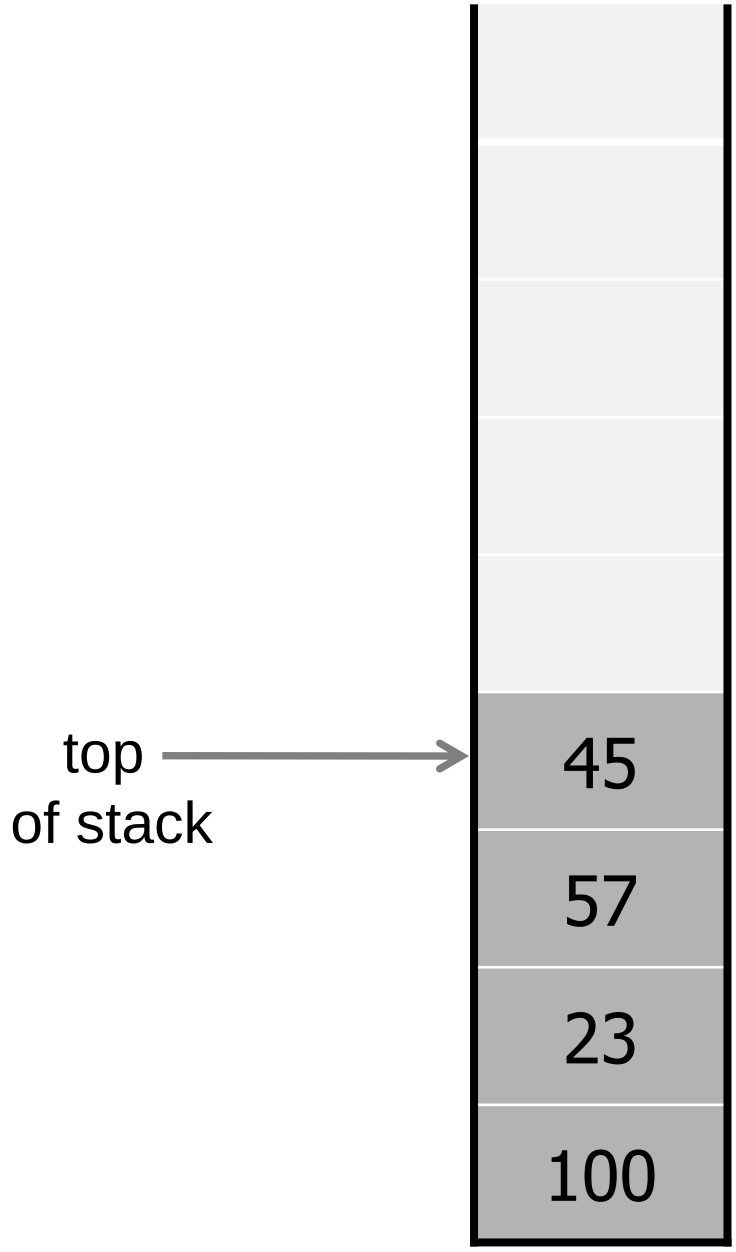


Abstract Data Types

Stack

Execution:

- push(77)

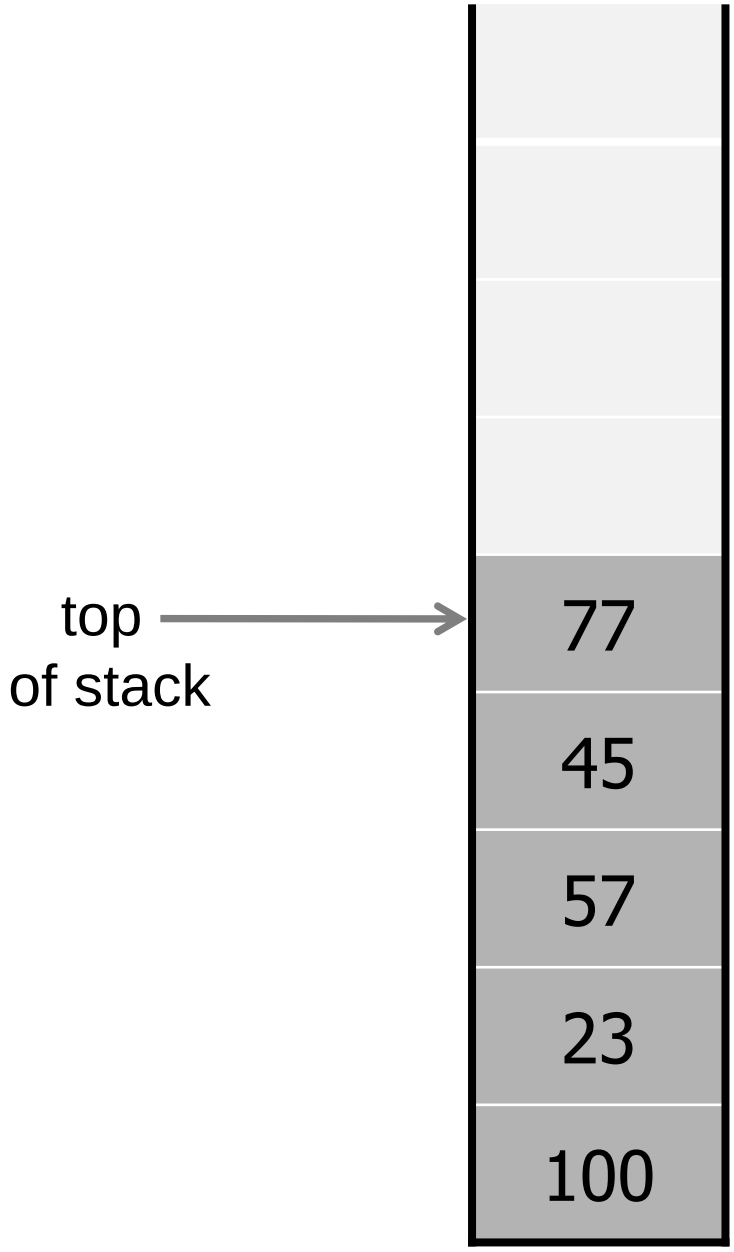


Abstract Data Types

Stack

Execution:

- push(77)

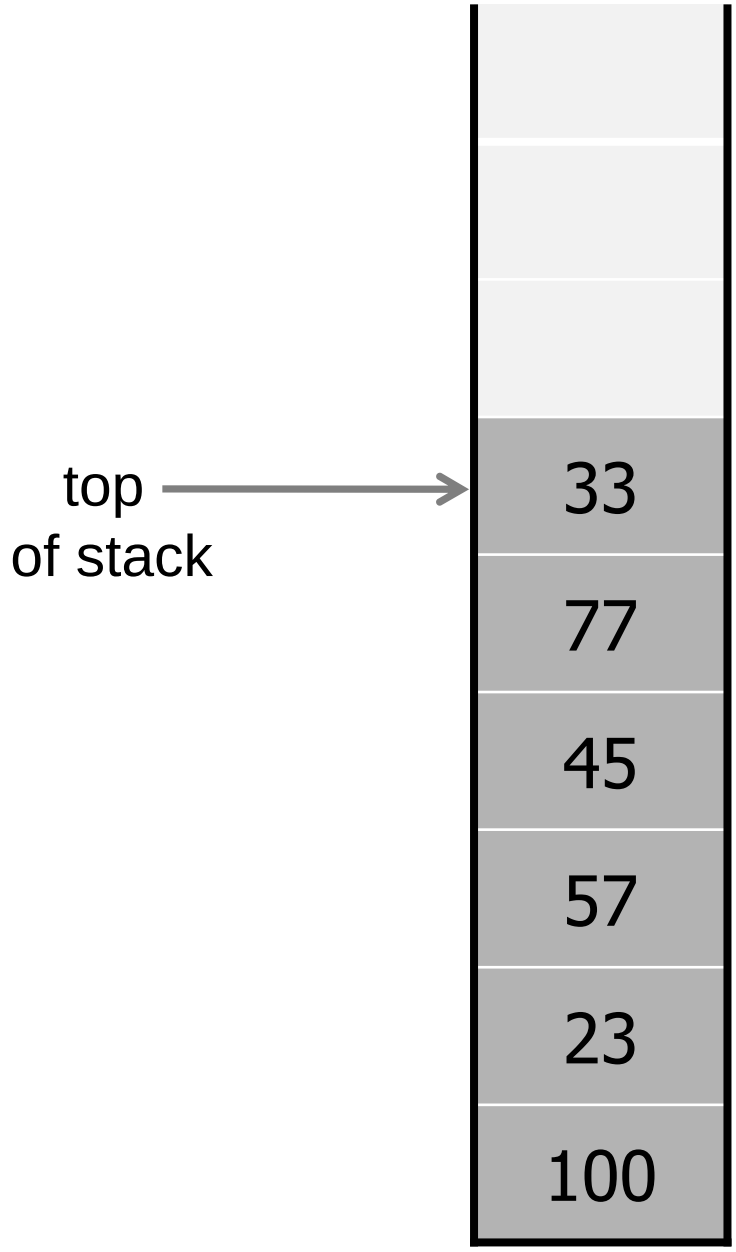


Abstract Data Types

Stack

Execution:

- push(77)
- push(33)

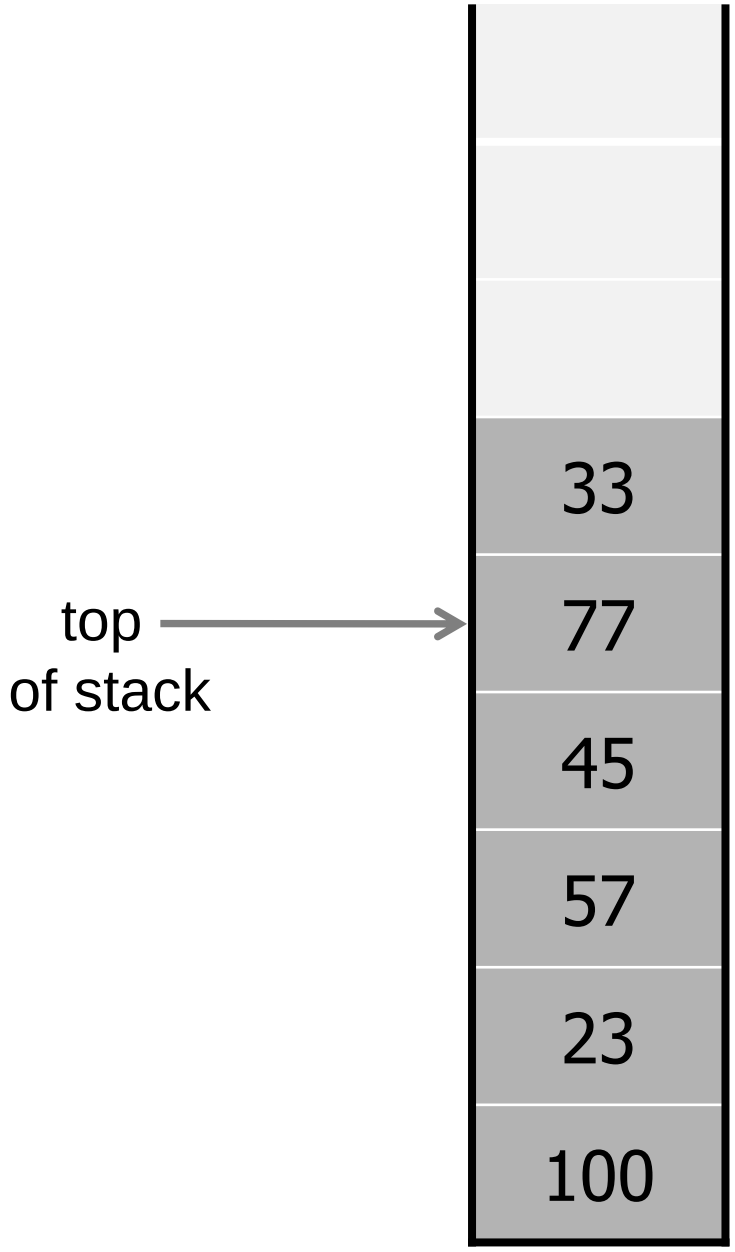


Abstract Data Types

Stack

Execution:

- push(77)
- push(33)
- pop() → ??

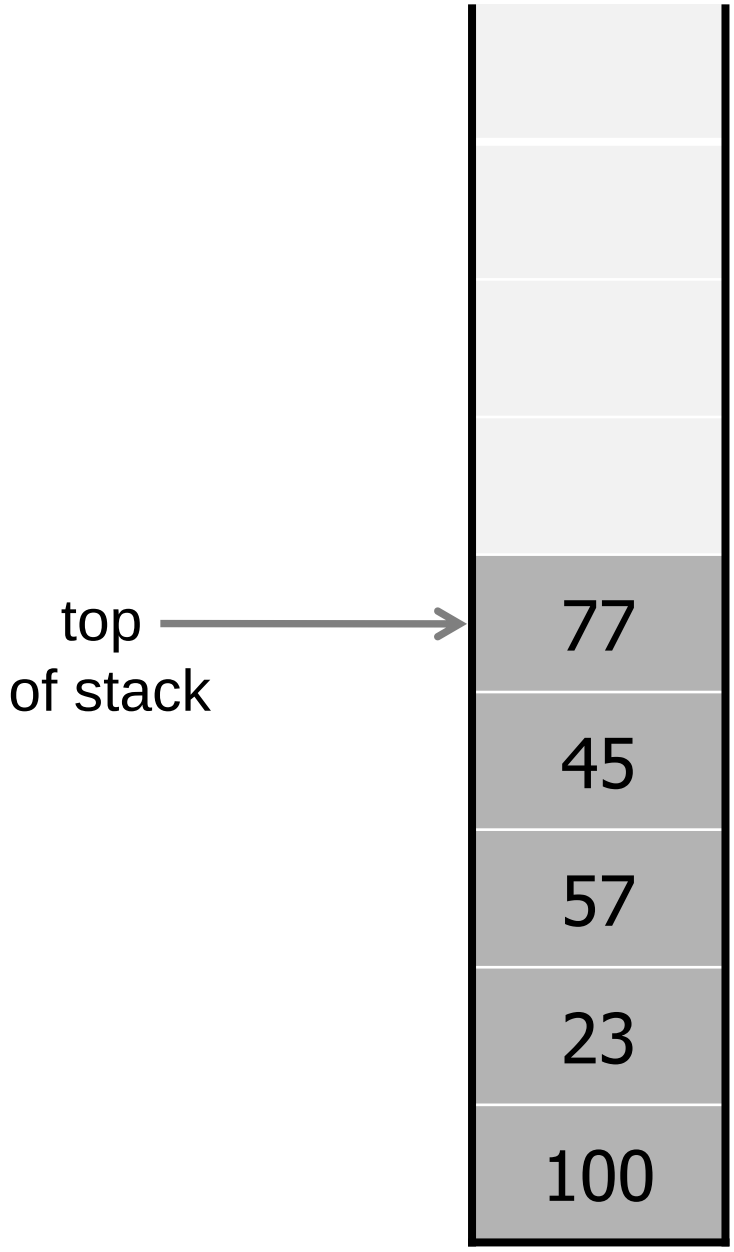


Abstract Data Types

Stack

Execution:

- push(77)
- push(33)
- pop() $\rightarrow$ 33

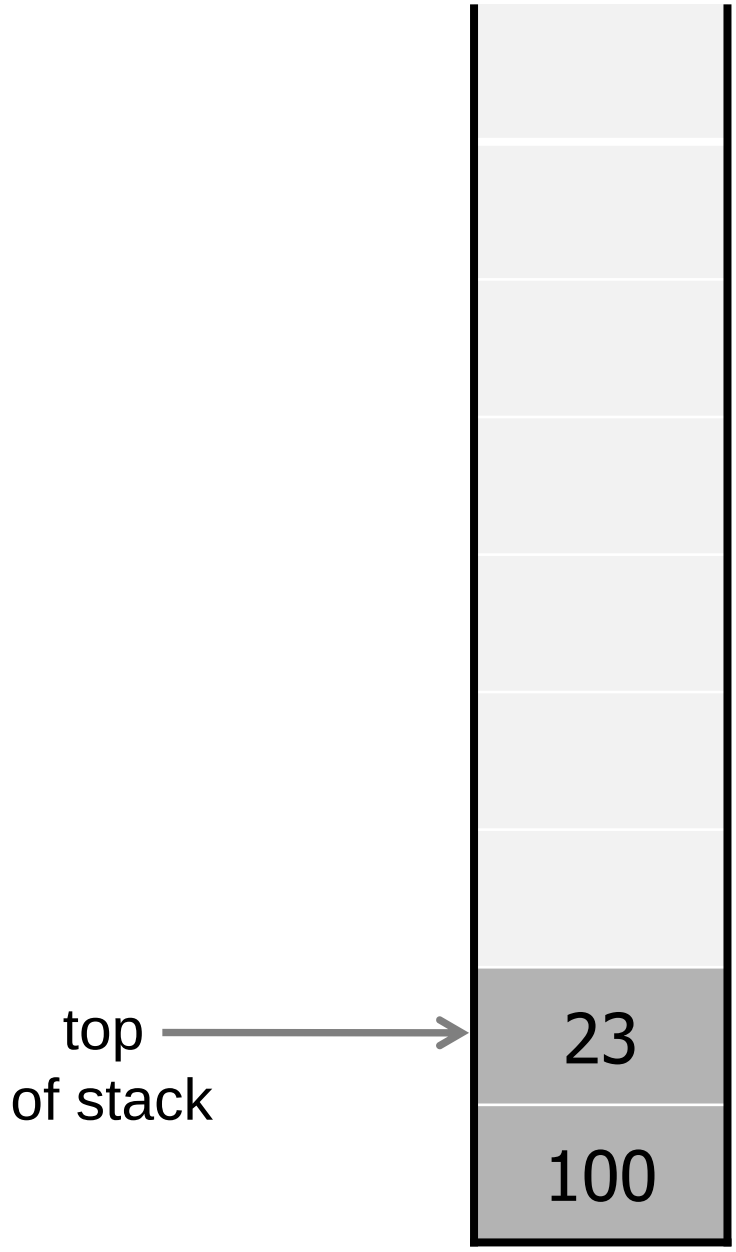


Abstract Data Types

Stack

Execution:

- push(77)
- push(33)
- pop() → 33
- pop() → 77
- pop() → 45
- pop() → 57

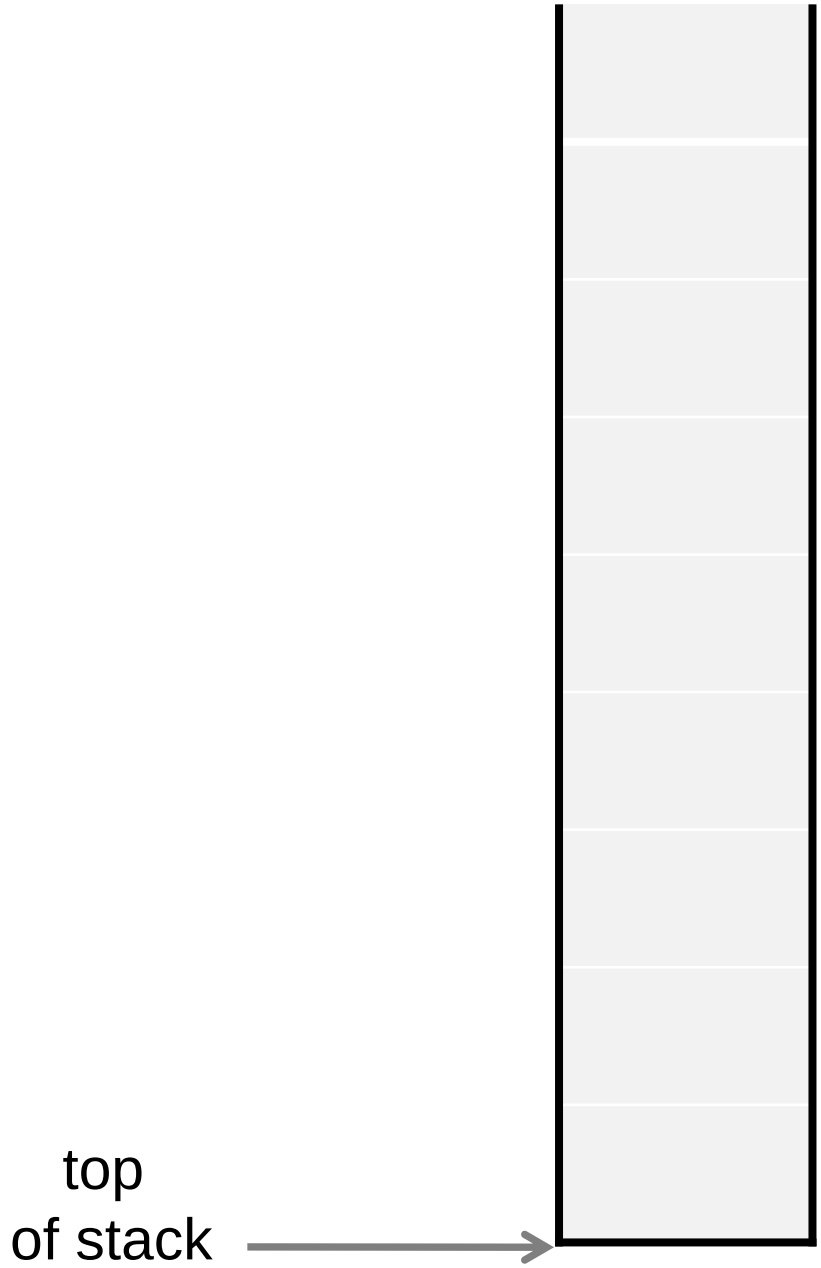


Abstract Data Types

Stack

Execution:

- pop() → 23
- pop() → 100

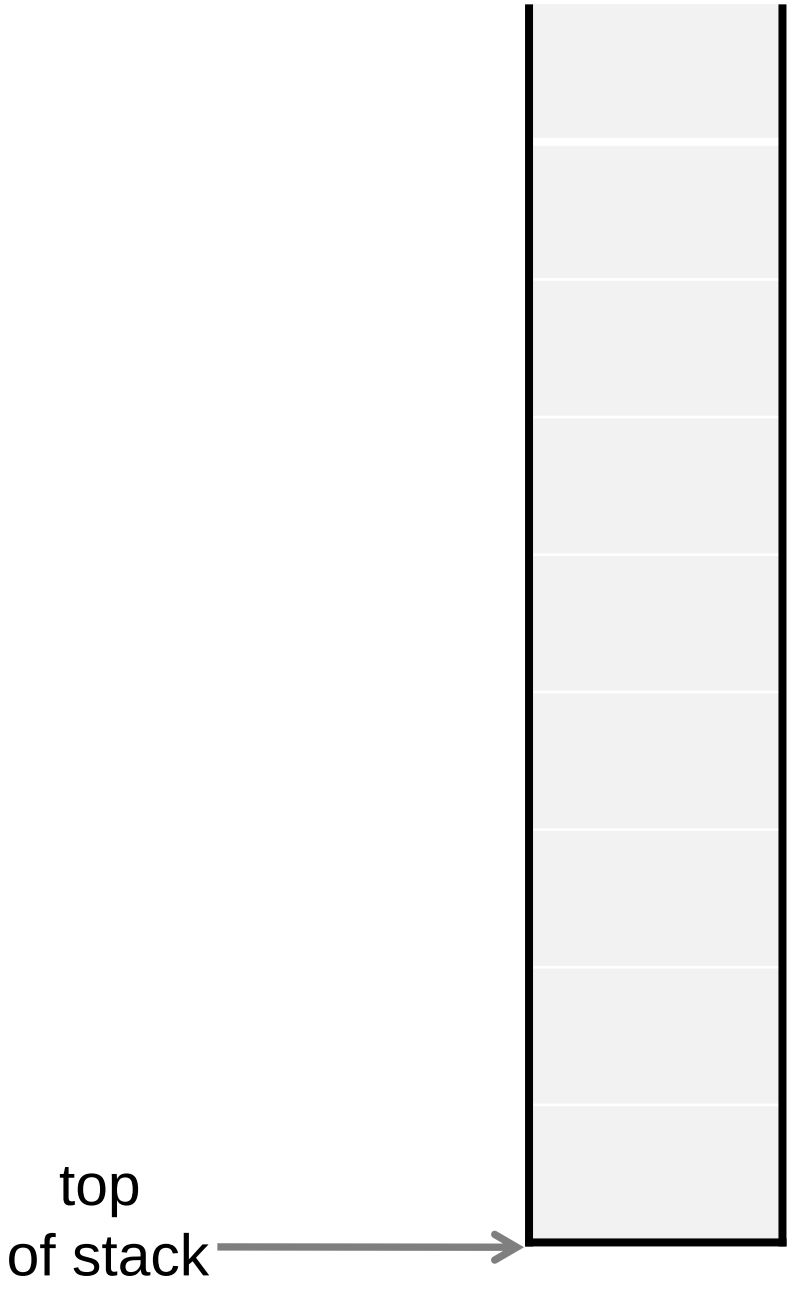


Abstract Data Types

Stack

Execution:

- pop() → 23
- pop() → 100
- pop() → ??



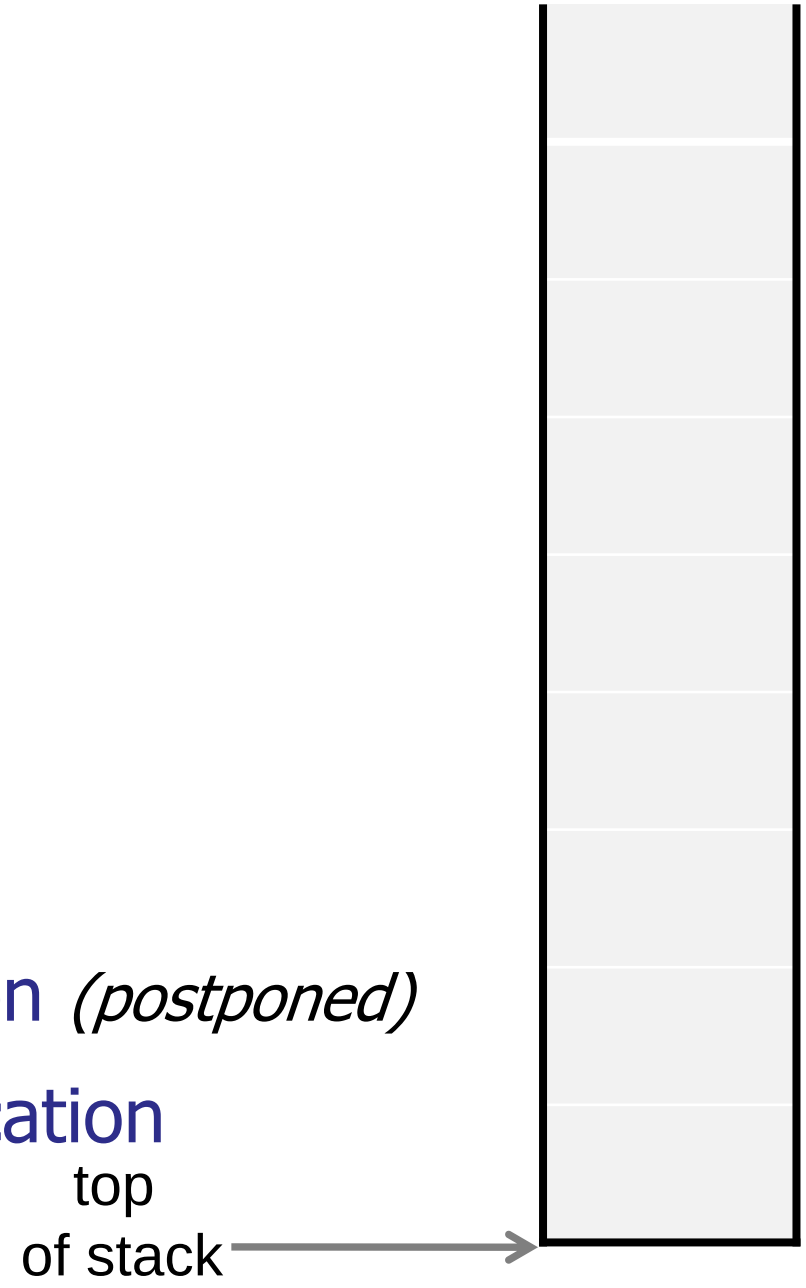
Abstract Data Types

Stack

Execution:

- pop() → 23
- pop() → 100
- pop() → ??

- Error!
  - Option 1: throw exception *(postponed)*
  - Option 2: modify specification

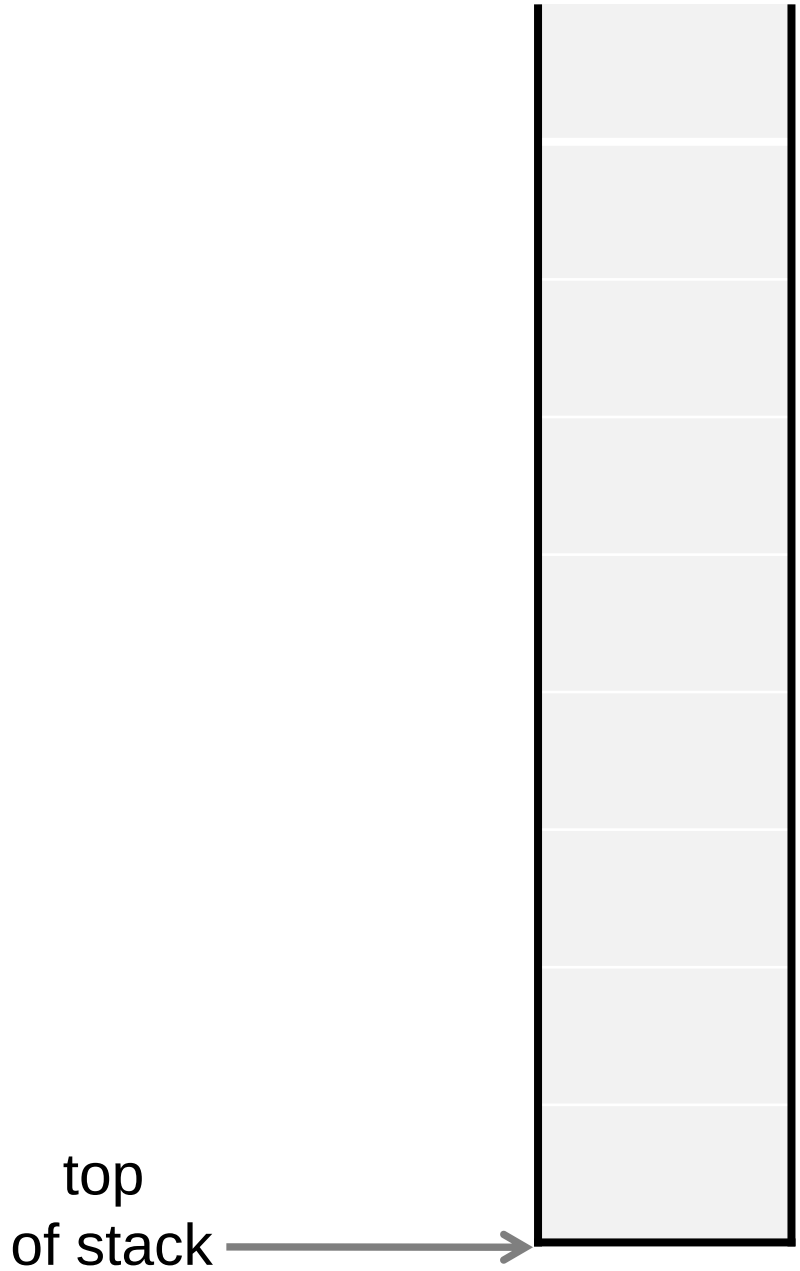


Abstract Data Types

Stack

Execution:

- pop() → 23
- pop() → 100
- empty() → true



Abstract Data Types

Queue

Interface:

- void enqueue(element x)
- element dequeue()

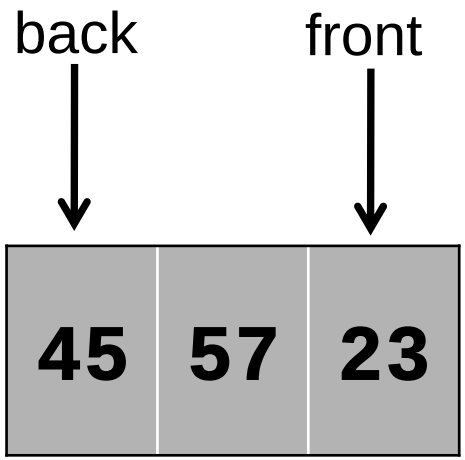
Behavior: (FIFO: first-in, first-out)

- enqueue(x) : adds element x to the front of the queue
- dequeue() : removes and returns element at the end of the queue

Abstract Data Types

Queue

Execution:

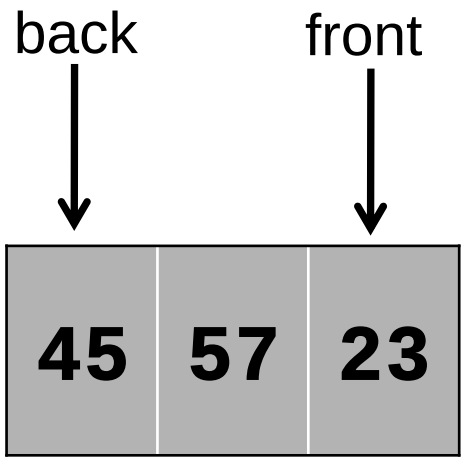


Abstract Data Types

Queue

Execution:

- enqueue(7)

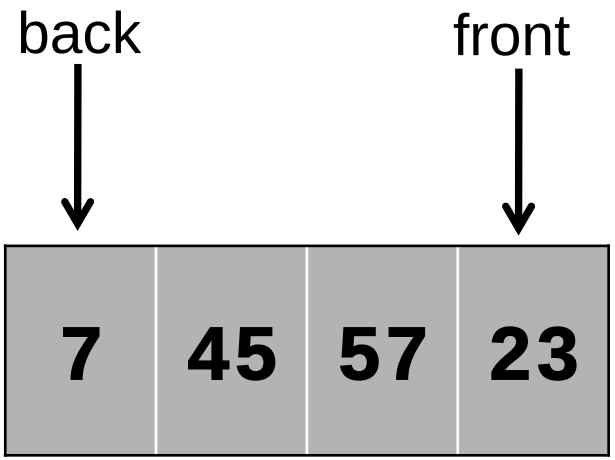


Abstract Data Types

Queue

Execution:

- enqueue(7)

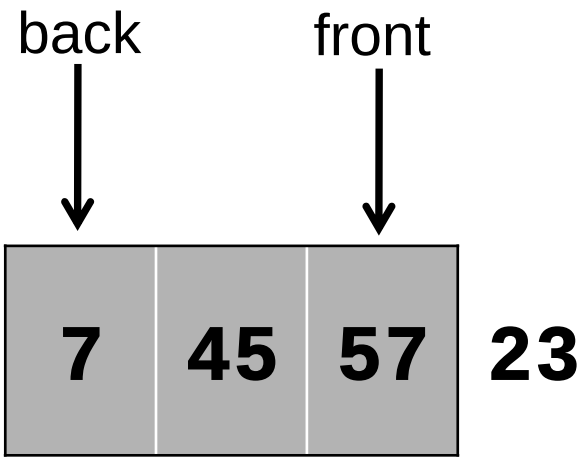


Abstract Data Types

Queue

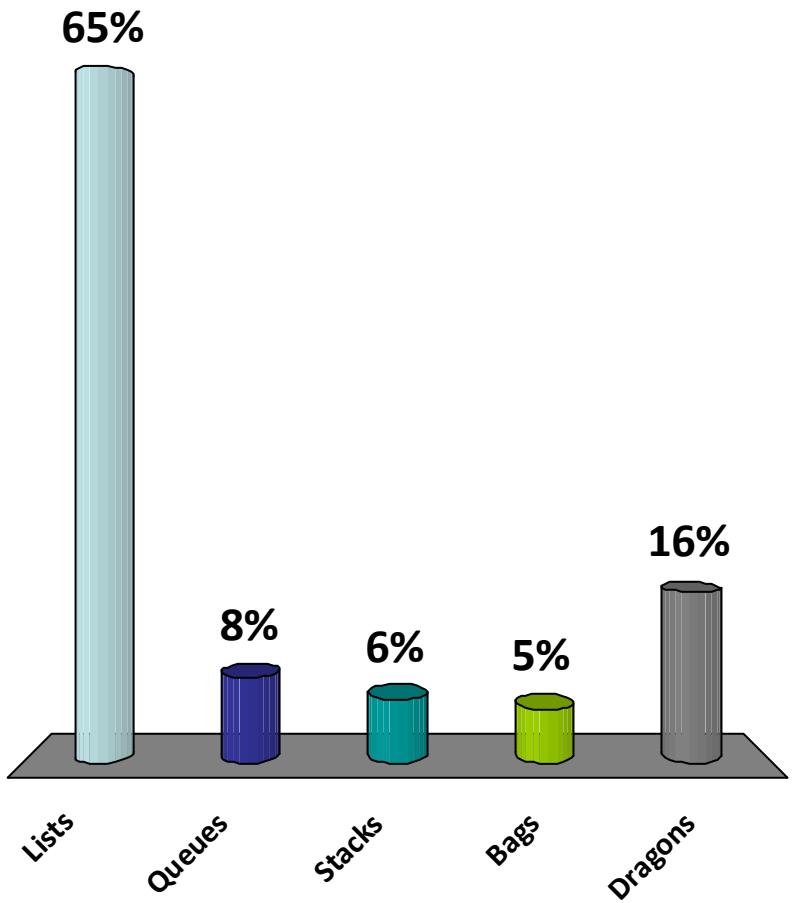
Execution:

- enqueue(7)
- dequeue() $\rightarrow$ 23



Which abstract data type appears most frequently in practice?

a. Lists
b. Queues
✓ c. Stacks
d. Bags
e. Dragons



Sorting with Stacks

Is it always possible to insert "pop" commands to make the output sorted?

Example:

6 5 4 3 2 1 ➔ 6 5 4 3 2 1 – – – – – –

Sorting with Stacks

Is it always possible to insert "pop" commands to make the output sorted?

Example:

6 5 4 3 2 1 ➔ 6 5 4 3 2 1 – – – – – –

1 2 3 4 5 6 ➔ 1 – 2 – 3 – 4 – 5 – 6

Sorting with Stacks

Is it always possible to insert "pop" commands to make the output sorted?

Example:

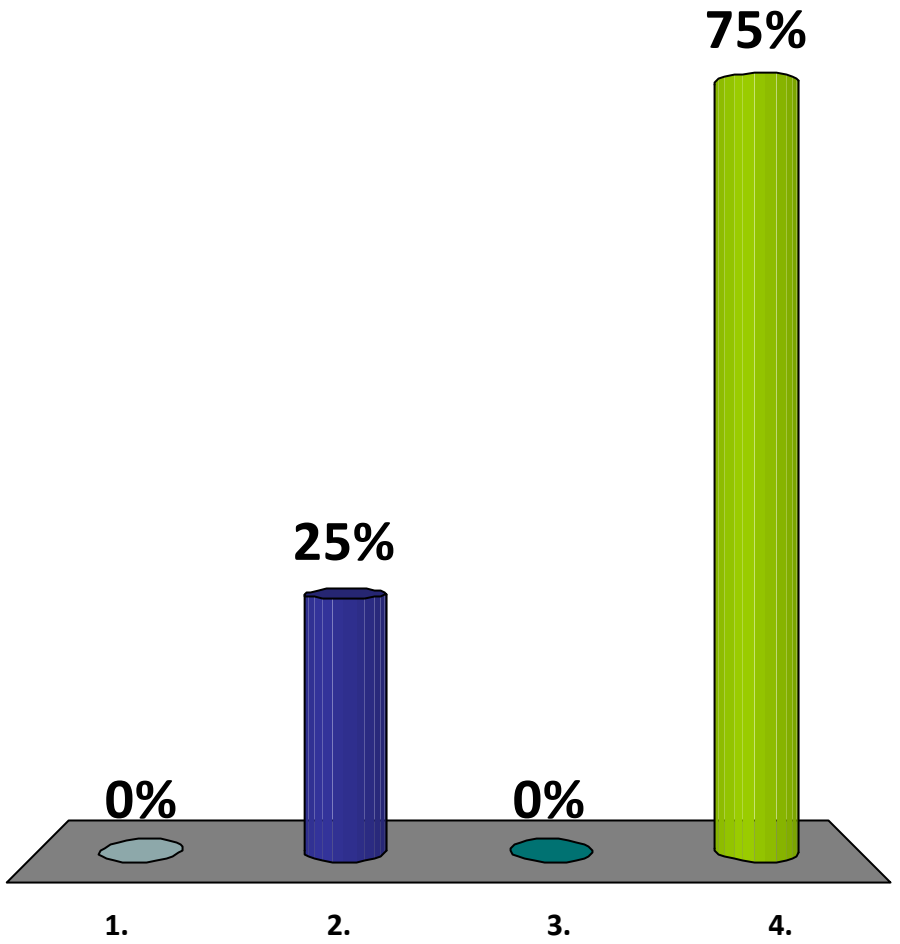
6 5 4 3 2 1 ➔ 6 5 4 3 2 1 – – – – – –

1 2 3 4 5 6 ➔ 1 – 2 – 3 – 4 – 5 – 6 –

4 1 3 2 6 5 ➔ 4 1 –3 2 – – –6 5 – –

Is it always possible to insert pop() commands to make the output sorted?

1. Yes
2. ✓ No
3. I have no idea
4. How does a stack work?



Sorting with Stacks

(Easy) Challenge:

Devise an algorithm that can determine how to sort a sequence with a stack, if it is possible (and fails if it is impossible).

Decide how many stacks you need to sort the sequence. (May be hard??)

NB assignment: be the first to write the solution here.

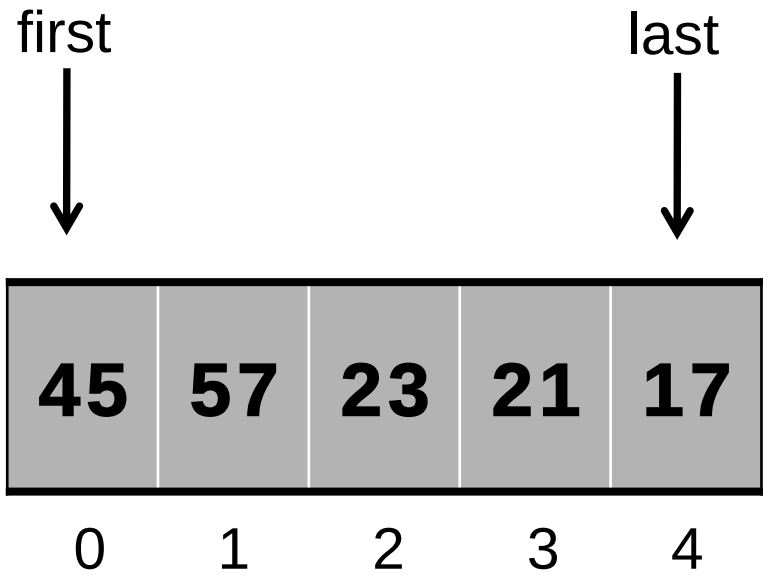
Implementing ADTs

Abstract Data Types

List

Interface:

```
void append(int x)
void prepend(int x)
void put(int x, int slot)
void remove(int x)
int getFirst()
int getLast()
int get(int slot)
boolean isEmpty()
```



```
public class FixedLengthList{

        final int MAXSIZE = 100;
        int[] m_list = new int[100];
        int lastElement= -1;

        // Add new key to the list
        void append(int key){
            lastElement++;
            m_list[lastElement] = key;
        }

        // Search the list
        boolean contains(int key){
            // Linear search
            for (int i=0; i<=lastElement; i++){
                if (m_list[i] == key) return true;
            }
            return false;
        }
```

```
public class FixedLengthList{

    final int MAXSIZE = 100;
    int[] m_list = new int[100];
    int lastElement= -1;

    // Add new key to the list
    void append(int key){
        lastElement++;
        m_list[lastElement] = key;
    }

    // Search the list
    boolean contains(int key){
        // Linear search
        for (int i=0; i<=lastElement; i++){
            if (m_list[i] == key) return true;
        }
        return false;
    }
```

```
public class FixedLengthList{

    final int MAXSIZE = 100;
    int[] m_list = new int[100];
    int lastElement= -1;

    // Add new key to the list
    void append(int key){
        lastElement++;
        m_list[lastElement] = key;
    }

    // Search the list
    boolean contains(int key){
        // Linear search
        for (int i=0; i<=lastElement; i++){
            if (m_list[i] == key) return true;
        }
        return false;
    }
```

```
public class FixedLengthList{
        final int MAXSIZE = 100;
        int[] m_list = new int[100];
        int lastElement= -1;

        // Add new key to list
        void append(int key) {
            if (lastElement<MAXSIZE-1) {
                lastElement++;
                m_list[lastElement] = key;
            }
            else {
                System.out.println("Error: overfull list.");
            }
        }
        // Search list
        boolean contains(int key){
            // Linear search
            for (int i=0; i<=lastElement; i++){
                if (m_list[i] == key) return true;
            }
            return false;
        }
```

```
public class FixedLengthList{
        final int MAXSIZE = 100;
        int[] m_list = new int[100];
        int lastElement= -1;

        // Add new key to list
        void append(int key) {
            if (lastElement<MAXSIZE-1) {
                lastElement++;
                m_list[lastElement] = key;
            }
            else{
                System.out.println("Error: overfull list.");
            }
        }
        // Search list
        boolean contains(int key){
            // Linear search
            for (int i=0; i<=lastElement; i++){
                if (m_list[i] == key) return true;
            }
            return false;
        }
```

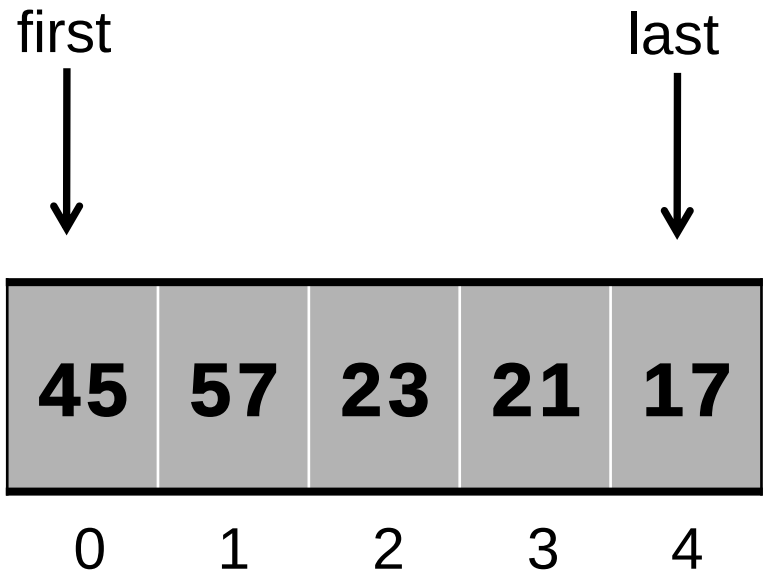
```
// Remove key in specified slot
public void remove(int elementNumber){

    // Do error checking
    ...

    // Move every item over by one
    for (int i=elementNumber; i<lastElement; i++){
        m_list[i] = m_list[i+1];
    }

    // Decrement lastElement and return
    lastElement--;
    return;
}
```

Abstract Data Types

List

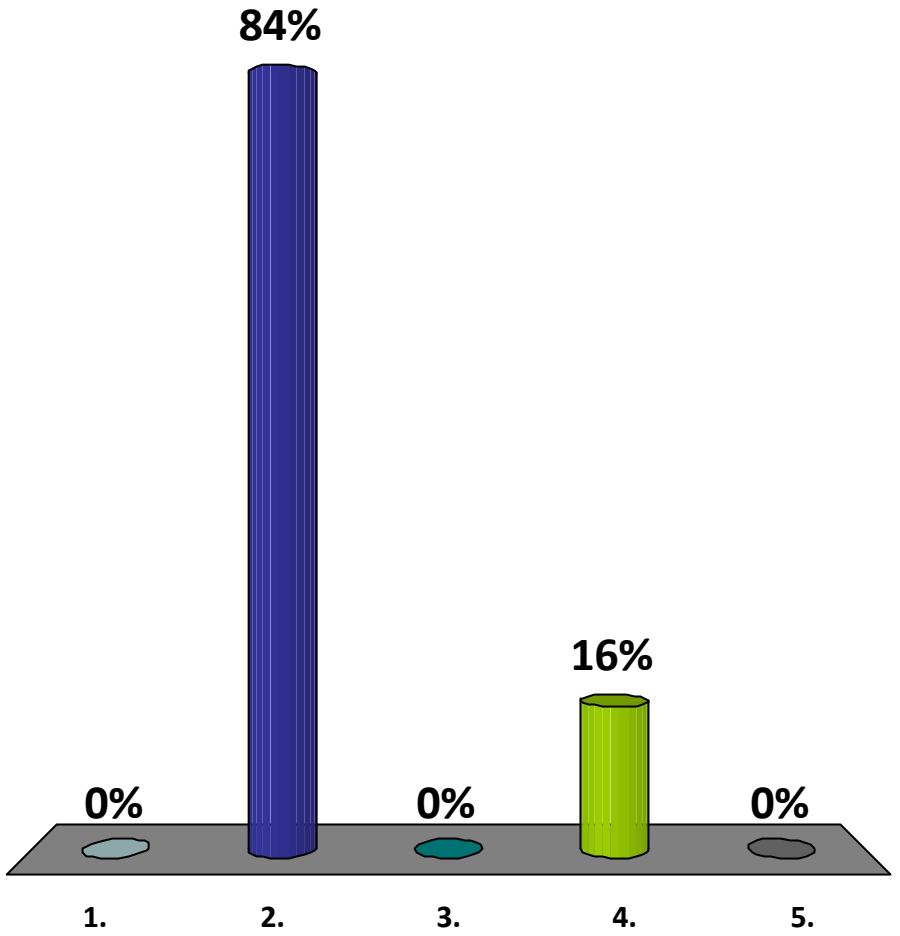
Interface:

```
void append(int x)
void prepend(int x)
void put(int x, int slot)
void remove(int x)
int getFirst()
int getLast()
int get(int slot)
boolean isEmpty()
```



What is the cost of adding an item to the beginning of the list in this implementation?

1. O(log n)
2. ✓ O(n)
3. O(n log n)
4. O(n^2)
5. O(2^n)



```
public class FixedLengthList{

        final int MAXSIZE = 100;
        int[] m_list = new int[100];
        int lastElement= -1;

        // Add new key to the list
        void append(int key){
            lastElement++;
            m_list[lastElement] = key;
        }

        // Search the list
        boolean contains(int key){
            // Linear search
            for (int i=0; i<=lastElement; i++){
                if (m_list[i] == key) return true;
            }
            return false;
        }
```

Implementing a Stack

Stack (of integers) :

```
class Stack{
    int[1000] stackArray;
    int top = 0;
```

```
boolean empty()
    return (top==0);
```

```
void push(int x)
    top++;
    stackArray[top] = x;
```

```
int pop()
    int i = stackArray[top];
    top--;
    return i;
```

Implementing a Stack

Stack (of integers) :

```
class Stack{
    int[1000] stackArray;
    int top = 0;
```

```
boolean empty()
   return (top==0);
```

```
void push(int x)
   top++;
   stackArray[top] = x;
```

What if stack is empty?

```
int pop()
   int i = stackArray[top];
   top--;
   return i;
```

Implementing a Stack

Stack (of integers) :

What if stack has 1001 elements?

```
class Stack{
   int[1000] stackArray;
   int top =0;
```

```
boolean empty()
   return (top==0);
```

```
void push(int x)
   top++;
   stackArray[top] = x;
```

```
int pop()
   int i= stackArray[top];
   top--;
   return i;
```

Implementing a Queue…

Queue

Interface:

- void enqueue(element x)
- element dequeue()

Exercise…

Implementing a Queue…

Queue

class QueueOfIntegers

class QueueOfFloat

class QueueOfString

…

class QueueOfStackOfIntegers

Problem:

- Rewriting code is tedious
- Maintaining many copies leads to errors

Parameterized Data Types

Generics

Parameterize queue with desired type

```
Queue<String> stringQueue = new Queue<String>();
Queue<Job> jobQueue = new Queue<Job>();

String name = “Joe”;
Job work = new Job(“CleanLaundry”);

stringQueue.enqueue(name);
jobQueue.enqueue(work);

Job nextJob = jobQueue.dequeue();
```

Generic Stack

```
public class SpecialStack<ItemType> {

      ItemType[] m_array = new ItemType[MAXITEMS];

      public void push(ItemType key){
         ...
      }

      public ItemType pop(){
         ...
      }
}
```

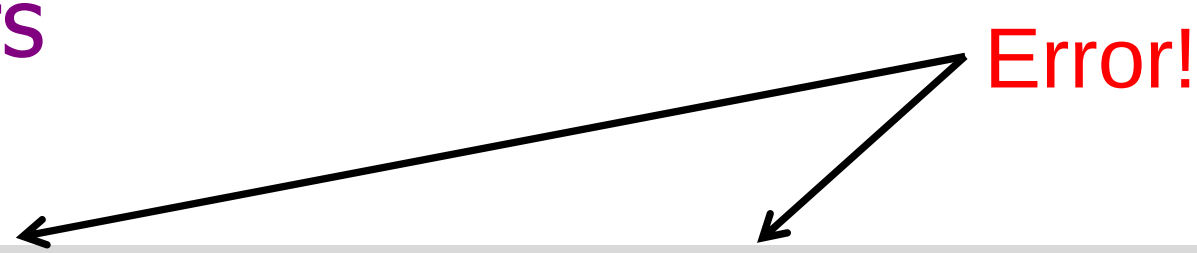
Parameterized Data Types

Error!

```
Queue<int> intQueue = new Queue<int>();


intQueue.enqueue(23);
int i = intQueue.dequeue();
```

Parameterized Data Types

Wrappers




```
Queue<int> intQueue = new Queue<int>();


intQueue.enqueue(23);
int i = intQueue.dequeue();
```

Parameterized Data Types

Wrappers

```
Queue<Integer> intQueue = new Queue<Integer>();


intQueue.enqueue(23);
int i = intQueue.dequeue();
```

- Integer wraps int
- Float wraps float
- Character wraps char
- Boolean wraps boolean

Parameterized Data Types

Wrappers

AutoBoxing

```
Queue<Integer> intQueue = new Queue<Integer>();


intQueue.enqueue(23);
int i = intQueue.dequeue();
```

AutoUnboxing

Inheritance

Building a better stack

What if I want to build a better stack?

- Add functionality
- Improve efficiency

Building a better stack

What if I want to build a better stack?

- Option 1: implement stack again

```
class myBetterStack  implements IStack{
    // implement push, pop, and empty
    ...
}
```

- Useful when:

  Entirely new implementation (e.g., don't use an array, use fractional cascading on a buffered tree).

Building a better stack

What if I want to build a better stack?

- Add functionality
- Improve efficiency

Solutions:

- Implement from scratch
- Modify original class
- Copy-paste old code to new class

Building a better stack

Inheritance

- MySpecialStack is a subclass (child) of Stack
- Stack is the superclass(parent) of MySpecialStack

```
class MySpecialStack extends Stack{

      void newFunction(){
      ...
      }

}
```

Building a better stack

Inheritance

- Subclass has all the functionality of the parent!

```
class MySpecialStack extends Stack{

     void newFunction(){
     ...
     }
}
```

```
MySpecialStack stack = new MySpecialStack();

stack.push(7)
stack.newFunction();
```

Building a better stack

Inheritance

- Subclass has all the functionality of the parent!

```
class MySpecialStack extends Stack{

     void newFunction(){
     ...
     }
}
```

MySpecialStack is a Stack

```
MySpecialStack stack = new MySpecialStack();

stack.push(7)
stack.newFunction();
```

Inheritance

Subclass substitutivity

- If `TypeBase` is a parent of `TypeOne`

  `TypeOne` extends `TypeBase`

```
TypeOne  first;

TypeBase  base =
  first;
```

Inheritance

Subclass substitutivity

- If `TypeBase` is a parent of `TypeOne`

  `TypeOne` extends `TypeBase`

- If `TypeOne` implements `TypeBase`

  `TypeBase` is an interface

```
TypeOne  first;

TypeBase  base =
  first;
```

Building a better stack

Inheritance

- Subclass has all the functionality of the parent!

```
class MySpecialStack extends Stack{

     void newFunction(){
     ...
     }
}
```

```
MySpecialStack stack = new MySpecialStack();

stack.push(7)
stack.newFunction();
```

Building a better stack

Inheritance

- Subclass can override parent class

```
class MySpecialStack extends Stack{

      @override
      void push(k){
            count++;
            specialPush(k);
      }
}
```

Building a better stack

Inheritance

- Subclass can override parent class

```
class MySpecialStack extends Stack{

      @override
      void push(k){
            count++;
            super.push(k);
      }
}
```

Building a better stack

Inheritance

- Subclass can override parent class

```
class animal {

      void eat(){ ... }

      void sleep(){ ... }

      void talk(){
            System.out.println("Hello");
      }

}
```

Building a better stack

Inheritance

- Subclass can override parent class

```
class dog extends animal{

      @override
      void talk(){
            System.out.println(“Woof”);
      }

}
```

Building a better stack

Inheritance

- Subclass can override parent class

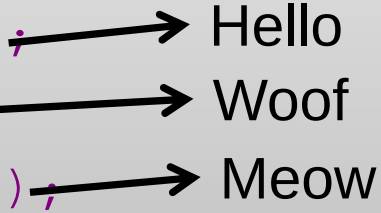
```
class cat extends animal{

      @override
      void talk(){
            System.out.println(“Meow”);
      }

}
```

Building a better stack

Inheritance

- Subclass can override parent class

```
animal Alice = new animal();
animal Doug = new Dog();
animal Collin = new Cat();

Alice.talk();
Doug.talk();
Collin.talk();
```

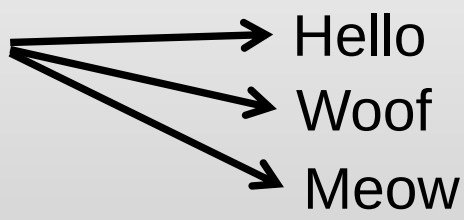


Building a better stack

Inheritance

- Subclass can override parent class

```
void pet(animal George){
     George.talk()
}
```



Building a better stack

Inheritance

- Subclass can override parent class

```
void pet(animal George){
     George.talk()
}



animal Doug = new Dog();
pet(Doug);
```

Building a better stack

Inheritance

- Subclass can override parent class

```
void pet(animal George){
     George.talk()
}
```


```
animal Doug = new Cat();
pet(Doug);
```

Building a better stack

Using a stack:

```
void fillStack(Stack store)
{
    for (int i=0; i<1000; i++)
    {
        store.push(i);
    }
}
```

```
{
    Stack A = new SlowStack()
    fillStack(A);
}
```

```
{
    Stack B = new FastStack()
    fillStack(B);
}
```

Building a better stack

Inheritance

```
class animal{
  private numEyes;
  protected numEars;
}
```

```
class dog extends animal{

   void updateEyes(){
     numEyes= 7;
   }

   void updateEars(){
     numEars= 10;
   }
}
```

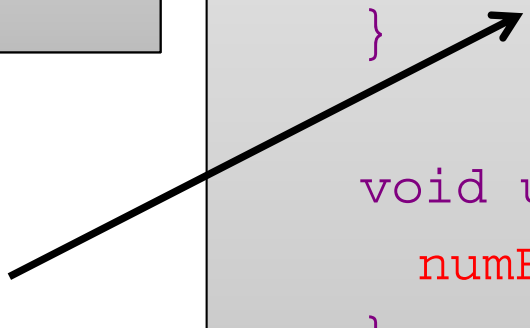
Building a better stack

Inheritance

- Access: public, private, protected

```
class animal{
  private numEyes;
  protected numEars;
}
```

```
class dog extends animal{

   void updateEyes(){
     numEyes= 7;
   }

   void updateEars(){
     numEars= 10;
   }
}
```



Building a better stack

Inheritance

- Constructors are not inherited

```
class animal {
   public animal(int  j){
     // Build your animal here
   }
}
```

```
class dog extends animal {
   public dog(int j){
      super(j);
   }
}
```

Building a better stack

Inheritance

- Default: child classes call empty parent constructor

```
class animal {
   public animal(int  j){
     // Build your animal here
   }
}
```

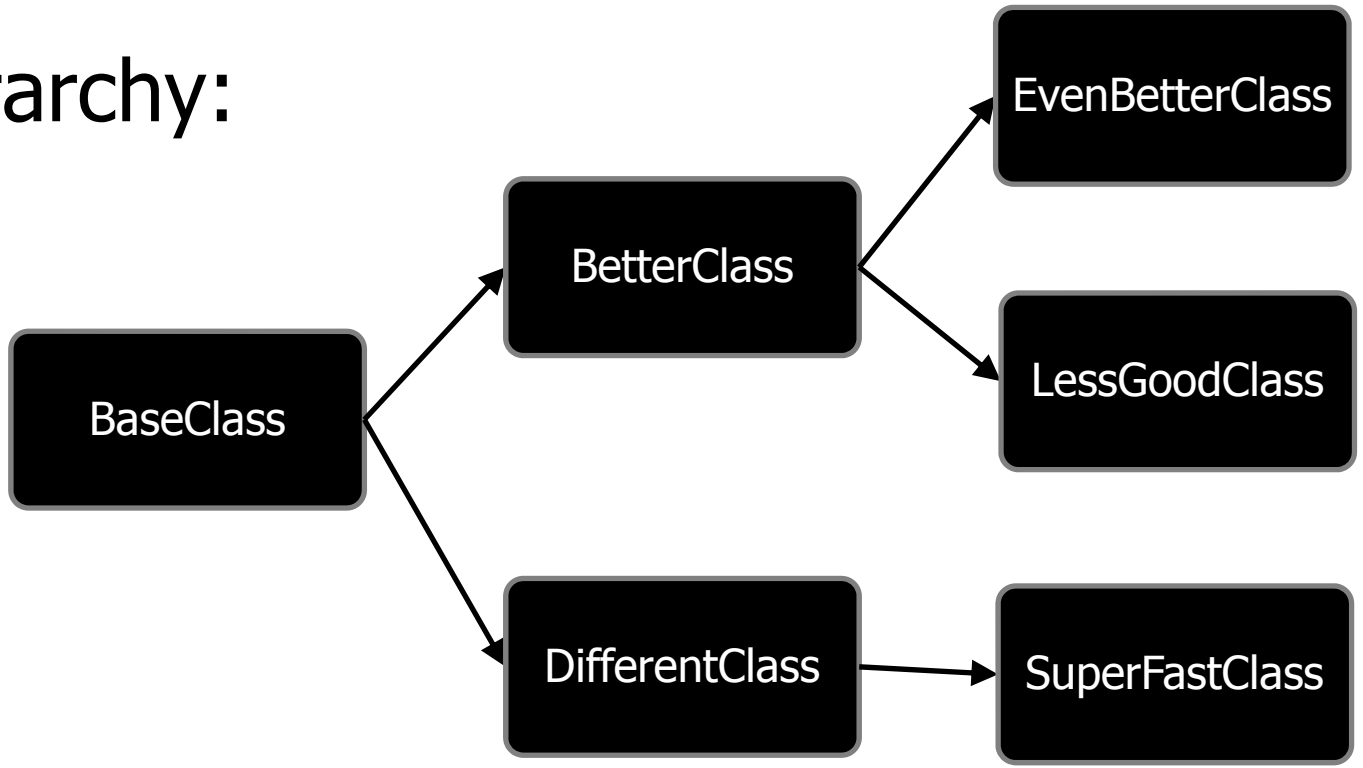
```
class dog extends animal {
   public dog(int j){
      super();
   }
}
```

Inheritance

Rules of inheritance:

- You can implement many interfaces.
- You can only extend one class.

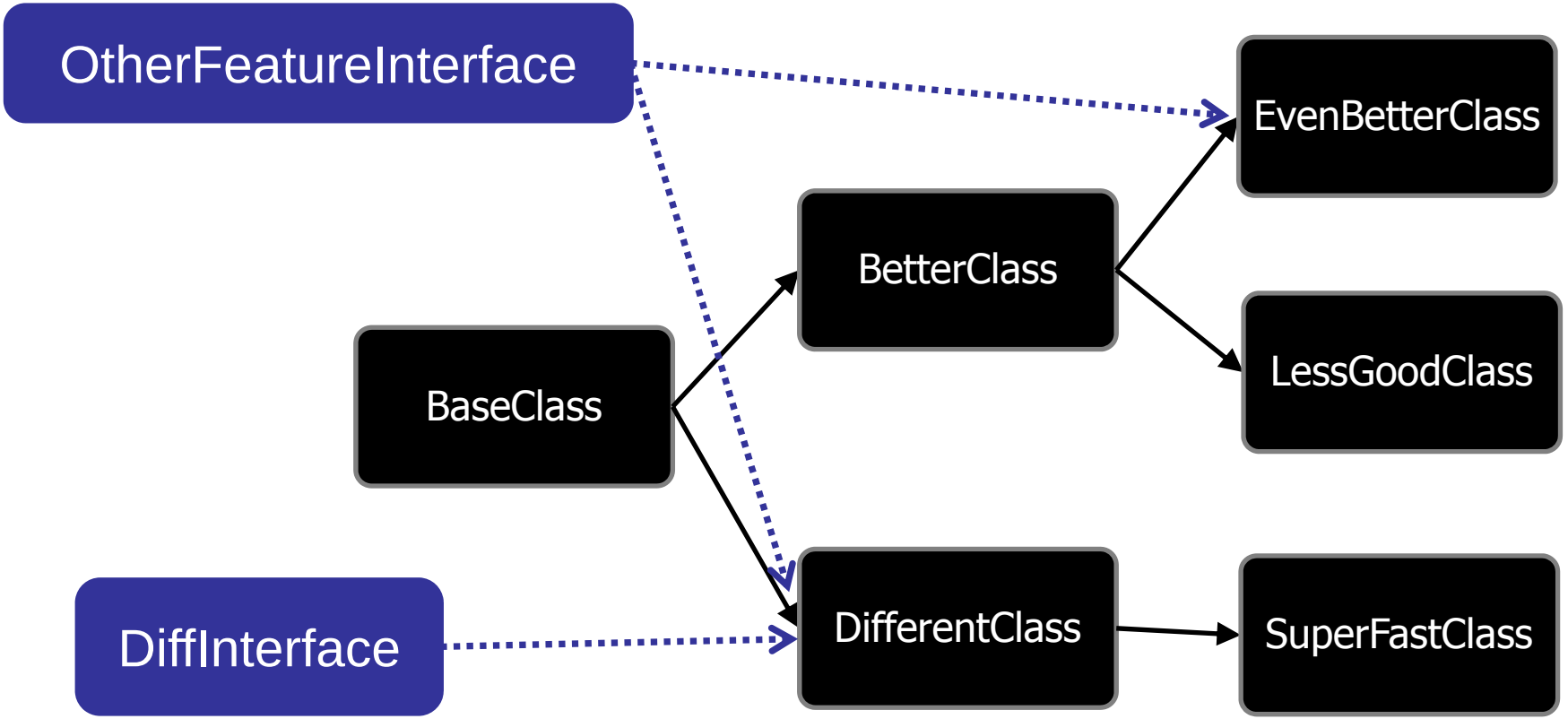
Class hierarchy:



Inheritance

Rules of inheritance:

- You can implement many interfaces.
- You can only extend one class.



Inheritance

VectorTextFile class:

- v1: slow
- v2: improved string management
- v3: improved sorting
- v4: no sorting

Problem:

- How to figure out what changed from v2 to v3?

Inheritance

VectorTextFile class:

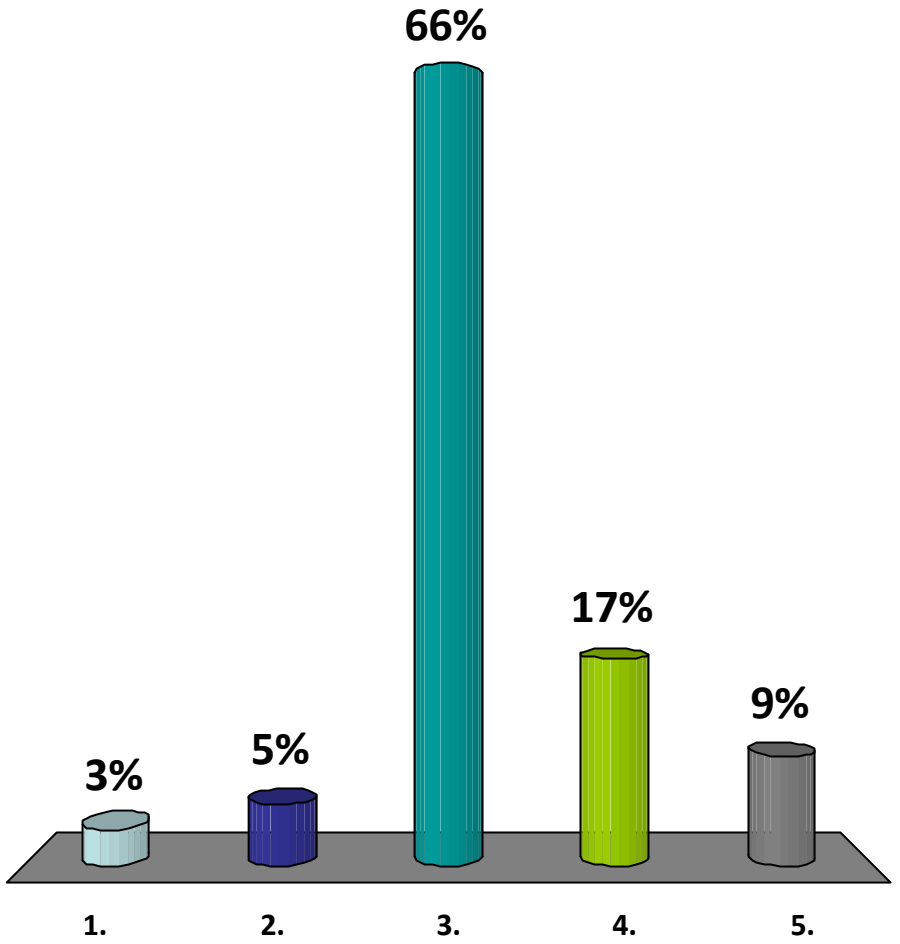
- v1: slow
- v2: improved string management
- v3: improved sorting
- v4: no sorting

Good practice:

- Use inheritance!
- Each version contains only what is new.

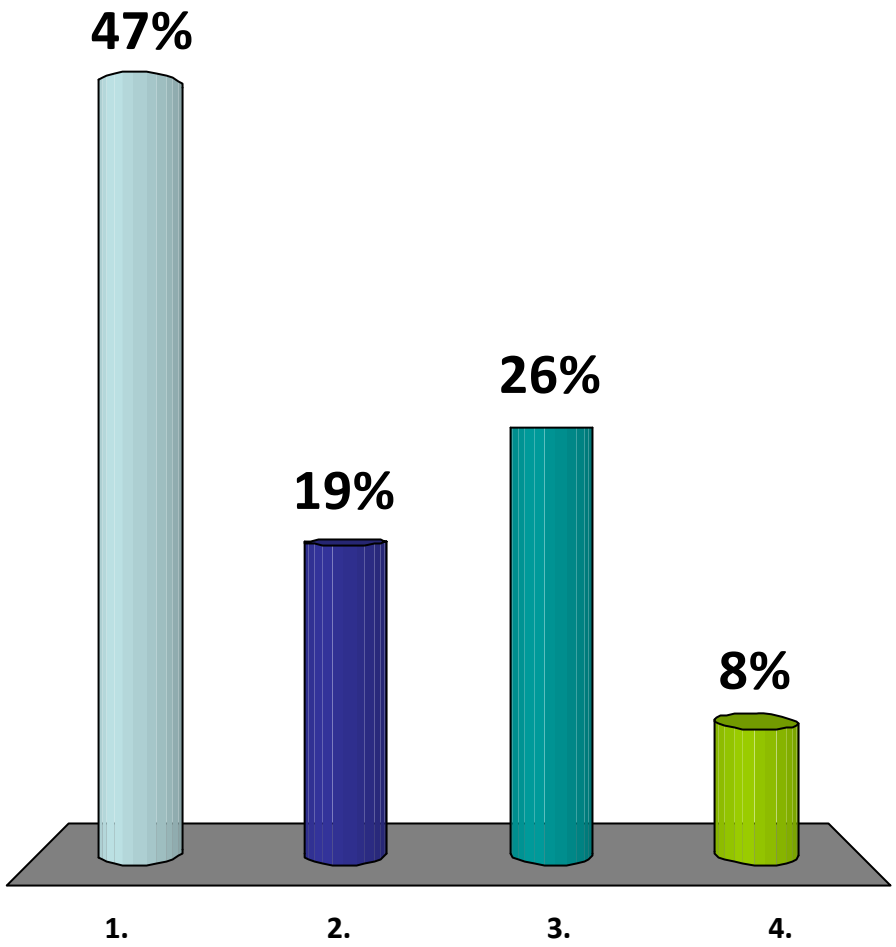
Which of the following is the LEAST common reason for using inheritance?

1. Modular design
2. Adding new functionality
3. Minimizing deep nesting.
4. Abstraction
5. Specializing a class for a particular use.



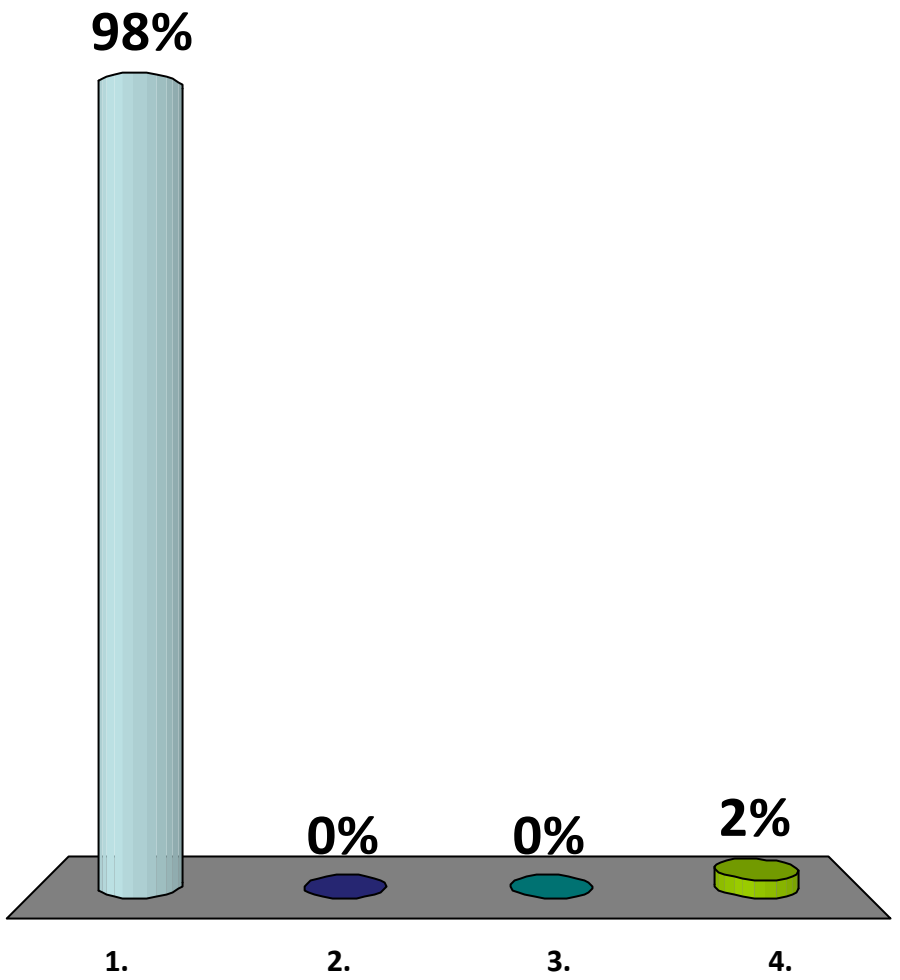
What is the best way to describe an abstract data type in Java?

✓1. Interface
2. Class
3. Either class or interface
4. Neither



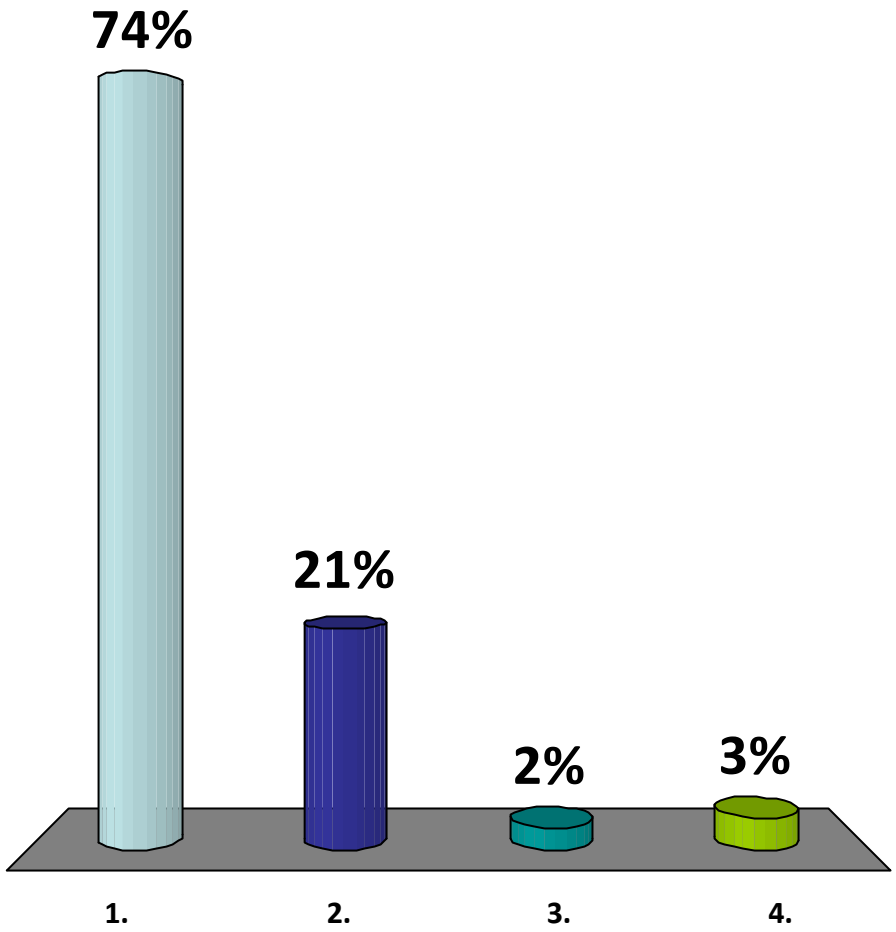
What is a: Queue<Stack<IntegerBag>> dbase

1. Queue of stacks of integer bags
2. Bag of stack of queues of integers
3. Stack of queues of integer bags
4. None of the above.



Assume Child extends Parent. Which constructor is executed first on new Child()?

✓1. Parent()
2. Child()
3. Undefined by Java specification
4. Depends on the parameter lists.



Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            super();
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            super();
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            super();
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            super();
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            // No call to super
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            // No call to super
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            // No call to super
            System.out.println(`Dog constructor.');
      }
}
```

Constructors and Inheritance

```
class animal {

      animal(){
            System.out.println(`Animal constructor.');
      }
}
```

```
class dog extends animal{

      dog(){
            // No call to super
            System.out.println(`Dog constructor.');
      }
}
```

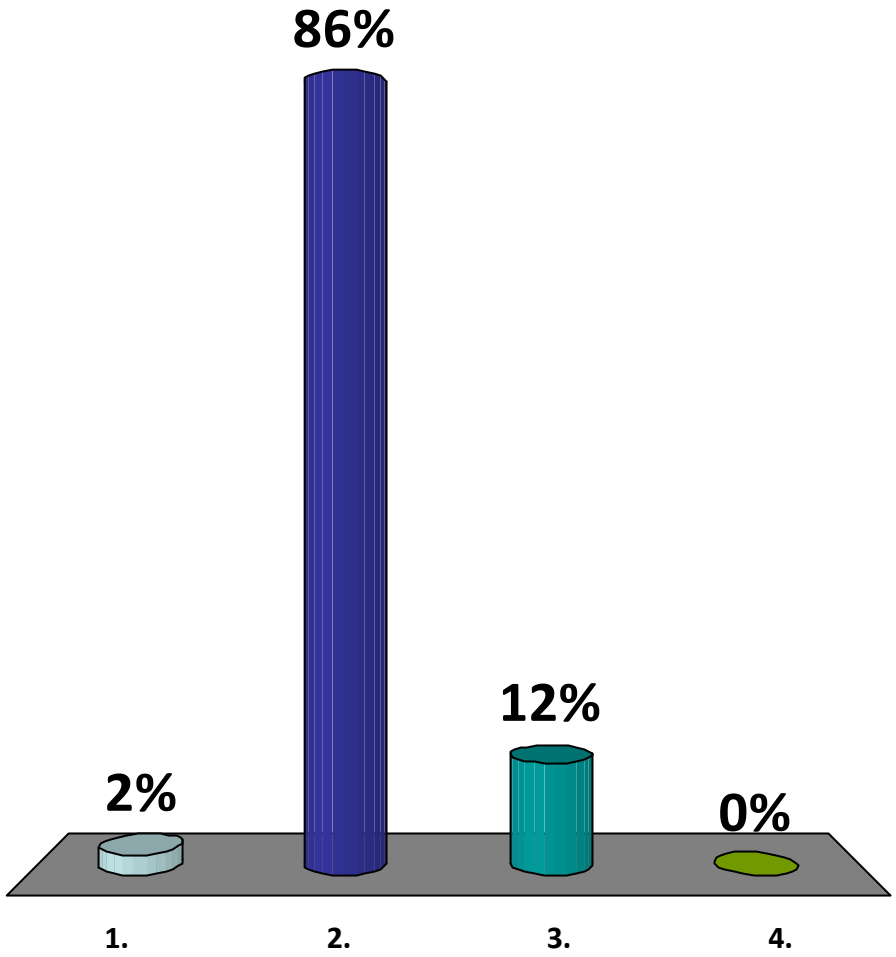
Constructors and Inheritance

Moral of the story:

- The parent class must always be constructed before the child class.
- If the child class begins with super(…), then the child class constructor is executed, which then executes the specified parent constructor.
- If the child class does not begin with super(…), then the default empty parent constructor is called.

Assume Child extends Parent, and assume Parent implements Grandparent. Which constructor executes first on new Child():

1. Child
2. ✓ Parent
3. Grandparent
4. Undefined in the Java specification



Today's Plan

Abstract data types

- Bags
- Lists
- Stacks
- Queues

Java

- Generics
- Inheritance
- Polymorphism

Handling Errors

What should we do when something goes wrong?

Handling Errors

What should we do when something goes wrong?

- Bugs…
- User error…
- Bad input…
- Corrupted files…
- Unexpected results…

Handling Errors

What should we do when something goes wrong?

Option 1: Terminate

```
// Require: n >= 0
public static int factorial(int n) {
   if (n < 0) System.exit(0);
   assert(n >= 0);
   int ans = 1;
   for (int i=2; i<= n; i++) ans*=i;
   return ans;
}
```

Handling Errors

What should we do when something goes wrong?

Option 1: Terminate.

- Pros:
  - Halts immediately.
  - Clearly indicates a problem.
- Cons:
  - No attempt at recovery.
  - All cases treated identically.
  - Little debugging information provided.

Handling Errors

What should we do when something goes wrong?

Option 2: Print out an error.

```
// Require: n >= 0
public static int factorial(int n) {
   if (n < 0) {
         System.out.println("Error! Bad input.");
         return -1;
   }
   assert(n >= 0);
   int ans = 1;
   for (int i=2; i<= n; i++) ans*=i;
   return ans;
}
```

Handling Errors

What should we do when something goes wrong?

Option 2: Print out an error.

- Pros:
  - Provides some debugging information.

- Cons:
  - Program keeps running.
  - What value should be returned?
  - No indication of error to the program.
  - No mechanism for recovery.
  - What if your "user" is another program?

Handling Errors

What should we do when something goes wrong?

Option 3: Integrate into control flow.

```
// Require: k != null
public int insert(key k) {
   boolean success = true;

   ...

   if (success) return 1;
   else return -1;
}
```

Handling Errors

What should we do when something goes wrong?

Option 3: Integrate into control flow.

- Pros:
  - Returns information on errors.
  - Can provide specific error information.
  - Enables recovery.
- Cons:
  - Complicates main program. For example, does every method have to return an error status??

Exceptions

Goals:

- Indicates when an error has occurred.
- Stops execution on error.
- Simplifies recovery from errors by providing information that the calling **program** can use to recover.
- Minimal overhead when there are no errors.
- Simplifies debugging of errors.

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Exceptions

Construct an exception object:

- Exceptions are just objects:
  - Exception (base class for all exceptions)
  - ArithmeticException
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException
  - UnsupportedOperationException
  - FileNotFoundException

Exceptions

Two types of exceptions:

- Checked exceptions:
  - IOException
  - MySpecialException
- Runtime exceptions
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

Exceptions

Construct an exception object:

- Exceptions are just objects
- Build a new exception object:

```
Exception e = new IllegalArgumentException(“Bad
   input: key should not be null.”);
```

Exceptions

Construct an exception object:

- Exceptions are just objects
- Build a new exception object.
- All exceptions extend class **Exception**.
- You can/should create your own exception types:

```
class LinkedListException extends Exception {
}
```

Exceptions

All exceptions support:

public class	**ExceptionClass**	
	ExceptionClass(String msg)	*Constructor*
String	getMessage()	*Returns message*
void	printStackTrace()	*Prints the call stack*

Exceptions

Construct an exception object:

- Exceptions are just objects
- Build a new exception object.
- All exceptions extend **Exception**.
- You can/should create your own exception types.

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Handling Errors

```
// Require: n >= 0
public int fact(int n) throws FactorialException
{
   if (n < 0) {
         throw new FactorialException("n < 0");
   }
   assert(n >= 0);
   int ans = 1;
   for (int i=2; i<= n; i++) ans*=i;
   return ans;
}
```

Exceptions

Throwing exceptions:

- On error, throw the exception!
- Method signature must indicate which (checked) exceptions it may throw.

Handling Errors

```
// Require: n >= 0
public static int factorial(int n)
   throws IllegalArgumentException,
          ArithmeticException,
          FactorialException
{
...
}
```

Exceptions

Throwing exceptions:

- On error, throw the exception!
- Method signature must indicate which (checked) exceptions it may throw.
- May throw many types of exceptions.

Handling Errors

```
interface MyExcellentInterface {

   int factorial(int n) throws FactorialException;

   void doWork(Foo f) throws IOException;

}
```

Exceptions

Throwing exceptions:

- On error, throw the exception!
- Method signature must indicate which (checked) exceptions it may throw.
- May throw many types of exceptions.
- Exceptions must be declared in the interface.

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
```

Exceptions

Handling exceptions:

- Wrap risky code in a **try block**.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
   // Oops, there was a problem.
   // Do something!
}
```

Exceptions

Handling exceptions:

- Wrap risky code in a **try block**.
- **Catch** your (checked) exceptions.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
   // Oops, there was a problem.
   // Do something!
}
finally {
   // Cleanup code
   // This code is executed in all cases.
}
```

Exceptions

Handling exceptions:

- Wrap risky code in a **try block**.
- **Catch** your (checked) exceptions.
- Put any clean-up code in a **finally** block.
  - Example: closing files
  - Example: completing initialization
  - Example: removing inconsistent states

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
   // Oops, there was a problem.
   // Do something!
   // Terminate?
   // Print out an error?
   // Return an indicator?
   // Recover and continue?
}
```

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
 throw new Exception("Problem with factorial:" + e);
}
```

Handling Errors

```
void doWork(){   // Does not throw any exceptions

   // We're doing something risky:
   try {
         int j = factorial(n);
         System.out.println("Factorial was a success");
   }
   catch (FactorialException e){
         // Handle the exception here.
         // Do not pass it on!
   }

}
```

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.