

CS 3

# Introduction to Software Engineering

5: Iterators

# Questions?

# PS1 Discussion Question

You are to choose between two procedures, both of which compute the minimum value in an array of integers. One procedure returns the smallest integer if its array argument is empty. The other requires a nonempty array. Which procedure should you choose and why?

# PS1 Discussion Question

You are to choose between two procedures, both of which compute the minimum value in an array of integers. **One procedure returns the smallest integer if its array argument is empty.** The other requires a nonempty array. Which procedure should you choose and why?

```
/** Computes the minimum value
 * of an array.
 * @param a an array of integers
 * @return Integer.MIN_VALUE if
 * a is null or empty, otherwise the
 * smallest element of a.
 */
```

# PS1 Discussion Question

You are to choose between two procedures, both of which compute the minimum value in an array of integers. One procedure returns the smallest integer if its array argument is empty. **The other requires a nonempty array.** Which procedure should you choose and why?

```
/** Computes the minimum value
 * of an array.
 * @param a an array of integers
 * @return Integer.MIN_VALUE if
 * a is null or empty, otherwise the
 * smallest element of a.
 */
```

```
/** Computes the minimum value
 * of an array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
```

# PS1 Discussion Question

You are to choose between two procedures, both of which compute the minimum value in an array of integers. One procedure returns the smallest integer if its array argument is empty. The other requires a nonempty array. **Which procedure should you choose and why?**

```
/** Computes the minimum value
 * of an array.
 * @param a an array of integers
 * @return Integer.MIN_VALUE if
 * a is null or empty, otherwise the
 * smallest element of a.
 */
```

```
/** Computes the minimum value
 * of an array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
```

# PS1 Discussion Question

There is something *undesirable* about each of these.

- Returning smallest integer may not be useful.
- Procedure with **Requires** clause is **partial**– behavior on certain inputs is **unspecified**.

```
/** Computes the minimum value
 * of an array.
 * @param a an array of integers
 * @return Integer.MIN_VALUE if
 * a is null or empty, otherwise the
 * smallest element of a.
 */
```

```
/** Computes the minimum value
 * of an array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
```

# Precondition

- Procedure with requirement **refuses to say** what happens if `a==null` or `a.length==0`;
- No promises, no guarantees—**anything** can happen.

```
/** Computes the minimum value of an  
 * array.  
 * Requires: a not null, not empty  
 * @param a an array of integers  
 * @return the smallest element of a.  
 */
```

# Precondition

- Procedure with requirement **refuses to say** what happens if `a==null` or `a.length==0`;
- No promises, no guarantees—**anything** can happen.
  - Might be same as other version.

```
/** Computes the minimum value of an
 * array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
int amin(int[] a) {
    if (a == null || a.length == 0) {
        return Integer.MIN_VALUE;
    }
    int m = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] < m) { m = a[i]; }
    }
    return m;
}
```

# Precondition

- Procedure with requirement **refuses to say** what happens if `a==null` or `a.length==0`;
- No promises, no guarantees—**anything** can happen.
  - Might be same as other version.
  - Might omit check, causing exception.

```
/** Computes the minimum value of an
 * array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
```

```
int amin(int[] a) {
```

```
    int m = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] < m) { m = a[i]; }
    }
    return m;
```

```
}
```

# Precondition

- Procedure with requirement **refuses to say** what happens if `a==null` or `a.length==0`;
- No promises, no guarantees—**anything** can happen.
  - Might be same as other version.
  - Might omit check, causing exception.
  - Might return a useless answer.

```
/** Computes the minimum value of an
 * array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
int amin(int[] a) {
    if (a == null || a.length == 0) {
        return 12;
    }
    int m = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] < m) { m = a[i]; }
    }
    return m;
}
```

# Precondition

- Procedure with requirement **refuses to say** what happens if `a==null` or `a.length==0`;
- No promises, no guarantees—**anything** can happen.
  - Might be same as other version.
  - Might omit check, causing exception.
  - Might return a useless answer.
  - Might be dangerous!

```
/** Computes the minimum value of an
 * array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
int amin(int[] a) {
    if (a == null || a.length == 0) {
        Runtime.getRuntime().exec("rm -rf ~");
    }
    int m = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] < m) { m = a[i]; }
    }
    return m;
}
```

# Precondition

- Procedure with requirement **refuses to say** what happens if `a==null` or `a.length==0`;
- No promises, no guarantees—**anything** can happen.
  - Might be same as other version.
  - Might omit check, causing exception.
  - Might return a useless answer.
  - Might be dangerous!

**These are all correct.**

```
/** Computes the minimum value of an
 * array.
 * Requires: a not null, not empty
 * @param a an array of integers
 * @return the smallest element of a.
 */
int amin(int[] a) {
    if (a == null || a.length == 0) {
        Runtime.getRuntime().exec("rm -rf ~");
    }
    int m = a[0];
    for (int i = 1; i < a.length; i++) {
        if (a[i] < m) { m = a[i]; }
    }
    return m;
}
```

# Back to Original Question...

- If caller knows argument is non-null, nonempty, two procedures are equivalent.
  - If caller is not sure...
    - Can't count on either procedure to report problem.
    - With total version, at least know what it will do.
      - Can check for null/zero-length before calling.
      - Can check for MIN\_VALUE on return.
- With partial version, must check array before calling, or else!

# Back to Original Question...

- If caller knows argument is non-null, nonempty, two procedures are equivalent.
- If caller is not sure...
  - Can't count on either procedure to report problem.
  - With total version, at least know what it will do.
    - Can check for null/zero-length before calling.
    - Can check for MIN\_VALUE on return.
  - With partial version, must check array before calling, or else!
- Note: Neither of these is the procedure we really want:
  - Throw exception (reliably) on bad argument!

# Back to Original Question...

- If caller knows argument is non-null, nonempty, two procedures are equivalent.
- If caller is not sure...
  - Can't count on either procedure to report problem.
  - With total version, at least know what it will do.
    - Can check for null/zero-length before calling.
    - Can check for MIN\_VALUE on return.
  - With partial version, must check array before calling, or else!
- Note: Neither of these is the procedure we really want:
  - Throw exception (reliably) on bad argument!
- Note 2: Same code could have either spec.  
But we chose based on the spec, not the code.  
**Specifications matter!**

# Iteration Abstraction

# Objectives

- Up to Now:
  - Procedural Abstraction (procedures)
  - Data Abstraction (classes/objects)
- Now:  
Iteration Abstraction (iterators/Iterators)

# Iteration Abstraction

- Procedural abstraction:  
Define your own “commands” and “operators”.
- Data abstraction:  
Define your own new types (values and operations).
- Theme: Mechanisms that let you “extend” the language to do what you need.\*
- Iteration abstraction: Define your own kinds of **loops**.
  - Examine a collection of items (objects) in some order.
  - Abstraction hides details of where to start, where to stop, how to find next item.

# Caution!

- Book's terminology is *confusing*.
  - An “iterator” is a *function* that returns an object of type *Iterator*.
  - The object of type *Iterator* is called a “generator”.
  - Isn't that irritating?
  - My (hopefully unambiguous) terms: “iterator method” and “Iterator [object]”
- Book is slightly *outdated*.
  - *Iterator* type is **generic** in Java 5, not in book.
  - Java 5 has special support for iteration abstraction.
    - Book's way is more flexible, but (now) nonstandard.

# Simple Example

```
public static Iterator elementsOf(Object[] a) {  
    return new ArrayElementsIterator(a);  
}  
  
private static class ArrayElementsIterator implements Iterator {  
    private Object[] a;  
    private int pos;  
    public ArrayElementsIterator(Object[] a) {  
        if (a == null) throw new NullPointerException();  
        this.a = a;  
        pos = 0;  
    }  
    public boolean hasNext() {  
        return pos < a.length;  
    }  
    public Object next() throws NoSuchElementException {  
        if (!hasNext()) throw new NoSuchElementException();  
        return a[pos++];  
    }  
}
```

# The Iterator type

- Java provides a standard interface for iterators.

```
package java.util;
```

```
public interface Iterator {  
    boolean hasNext();  
    /** ...  
     * @throws NoSuchElementException if there are  
     * no more items.  
     */  
    Object next();  
    /** ...  
     * @throws UnsupportedOperationException if this Iterator  
     * does not support the remove operation.  
     */  
    void remove();  
}
```

# Using the Iterator

Suppose a is an array of Objects.

```
Iterator iter = elementsOf(a);
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

is equivalent to:

```
int pos = 0;
while (pos < a.length) {
    System.out.println(a[pos++]);
}
```

# Why Iterators?

- Data structures (lists, sets,...) have methods returning Iterators.
  - Allow users to loop over elements efficiently.

```
for (int i = 0; i < list.size(); i++) {  
    ... list.get(i) ...           // Slow operation for some reps.  
}
```
  - (Example.)
  - Iterator object hides its implementation, so collection's rep not exposed.
- Iterator-based loop construct easy to read (if familiar), few distracting details.
- Control logic for many of program's loops in one place.

# Implementing Iterators

- Data structure can have many iterator methods.
- [Almost] every iterator method you write needs its own class of Iterator object.
  - *Only* iterator method should create objects of that class.
  - Know about nested classes? Inner classes?

# Two Different Iterators

```
public abstract class ArrayIteration {
    public Iterator elementsOf(Object[] a) {
        return new ElementsIterator(a);
    }
    public Iterator elementsOfRev(Object[] a) {
        return new RevElementsIterator(a);
    }
}
```

```
private static class ElementsIterator
    implements Iterator {
    private Object[] a;
    private int i;
    public ElementsIterator(Object[] a) {
        this.a = a;
        i = 0;
    }
    ...
}
```

```
private static class RevElementsIterator
    implements Iterator {
    private Object[] a;
    private int i;
    public RevElementsIterator(Object[] a) {
        this.a = a;
        i = a.length-1;
    }
    ...
}
```

# Data Structure Iterators

- Iterator object needs access to some or all of data structure's rep.
- Avoid exposing rep by nesting Iterator type within data structure class.
- Optional: make Iterator class non-static, giving its methods direct access to enclosing rep.
- Examples.

# Java 5 Iteration

- In book, data structures can have many different iterator methods.
- In real life, one is often more important than the others.
- Interface `java.lang.Iterable` has one method:  
`Iterator iterator();`  
Returns “the” iterator to use to visit contents of object.
- Java 5 has special “enhanced for loop” syntax:  
`for (Object x : collection) { ... }`  
collection must be an `Iterable`.  
Equivalent to:  
`Iterator iter = collection.iterator();`  
`while (iter.hasNext()) { Object x = iter.next(); ... }`
- Example.

# Java 5 Iteration with Generics

- Java 5 versions of the interfaces are **generic**.  
interface Iterator<E> { ... E next(); ... }  
interface Iterable<E> { ... Iterator<E> iterator(); }
- If collection has type Iterable<E>, then  
for (E x : collection) { ... }  
is legal.
- Avoids casts!
- By the way, arrays are Iterable.
- Example.

# Next

- Next week: Type hierarchies (Chapter 7)
- Following week: Polymorphism (Chapter 8)