Maps,

Java: Collections, Exceptions

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 11, 14 February 2023

Abstract data types

- Separate public interface from private implementation

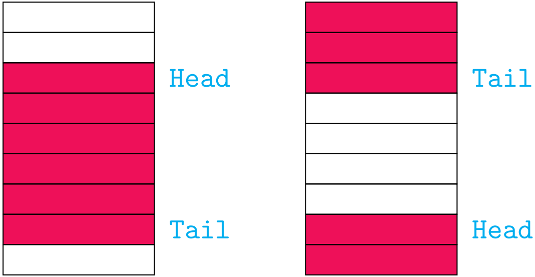
Abstract data types

- Separate public interface from private implementation
- For instance, a (generic) queue

```
public class Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

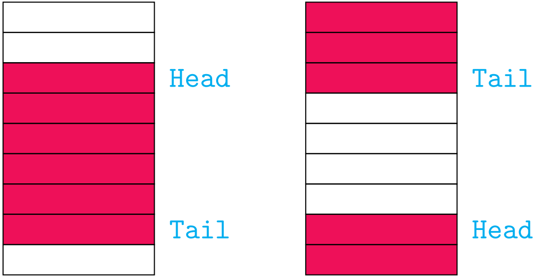
Abstract data types

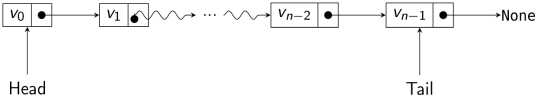
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array



Abstract data types

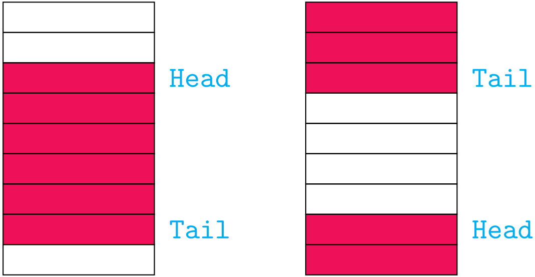
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list

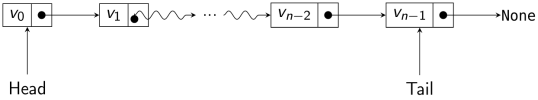




Abstract data types

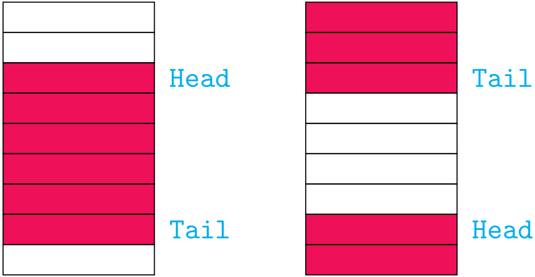
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class `Queue` can choose either one

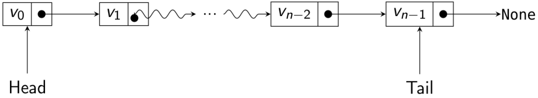




Abstract data types

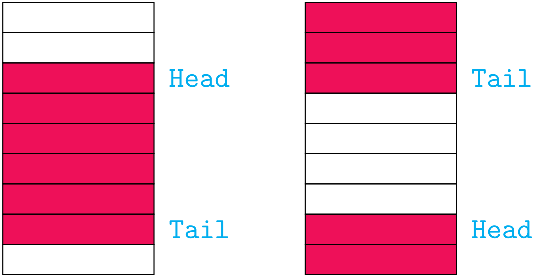
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class Queue can choose either one
- Public interface is unchanged

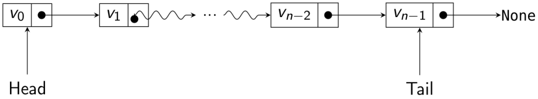




Abstract data types . . .

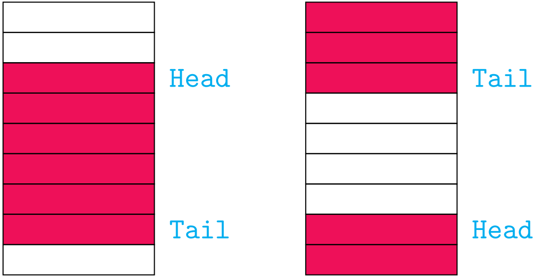
- Is the user indifferent to choice of implementation?

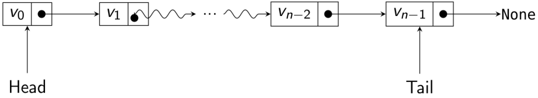




Abstract data types . . .

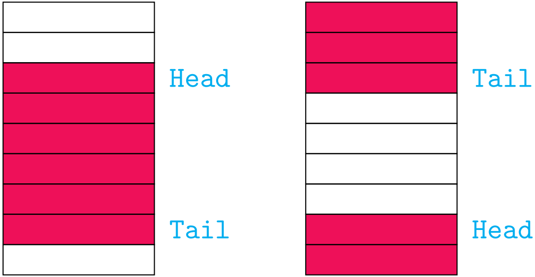
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects

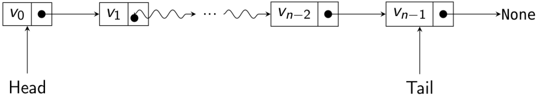




Abstract data types . . .

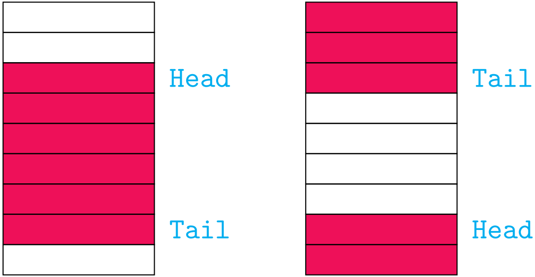
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation

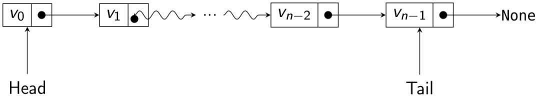




Abstract data types . . .

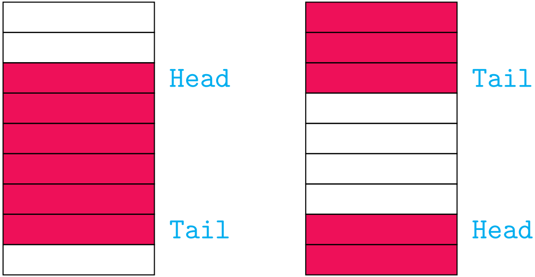
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
    - Circular array is better — one time storage allocation
- Flexibility
    - Linked list is better — circular array has bounded size

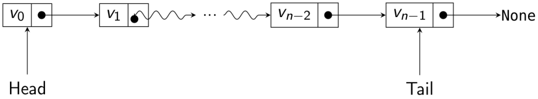




Abstract data types . . .

- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked list is better — circular array has bounded size
- Offer user a choice of implementation?





Multiple impementations

- Create two separate implementations

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}


public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations

- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`
- And also every function header, auxiliary variable, . . . associated with it

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}

public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

Only change this

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a Queue interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- Benefit of indirection — to use a different implementation for dateq, only need to update the instantiation

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

The power of indirection

- Use interfaces to flexibly choose between multiple concrete implementations
  - Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

"Fundamental theorem of software engineering"

All problems in computer science can be solved by another level of indirection.

Butler Lampson, Turing Award 1992

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, . . .

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, . . .
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, . . .
- Choose the one you need
- . . . but changing a choice requires multiple updates

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ...but changing a choice requires multiple updates
- Instead, organize these data structures by functionality

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ... but changing a choice requires multiple updates
- Instead, organize these data structures by functionality
- Create a hierarchy of abstract interfaces and concrete implementations
  - Provide a level of indirection

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- add() — add to the collection

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- add() — add to the collection
- iterator() — get an object that implements Iterator interface

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

The Collection interface

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- add() — add to the collection
- iterator() — get an object that implements Iterator interface
- Use iterator to loop through the elements

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}

public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
    - Implicitly creates an iterator and runs through it

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

```
if (element.equals(obj))
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added “for each” loop
    - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

  ```
  if (element.equals(obj))
  ```

- Later!

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Removing elements

- Iterator also has a `remove()` method
    - Which element does it remove?

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // Delete element if it has some property
  if (property(element)) {
     iter.remove();
  }
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.remove(); // Error
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.next();
iter.remove();
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`
- To remove the first element, need to access it first

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();

// Remove first element in cstr
iter.next();
iter.remove();
```

The Collection interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

The Collection interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

- Actually, `Collection` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The `Collection` interface — the full story

- How does this line work?

  `if (element.equals(obj))`

- Actually, `Collection` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`
- To implement the `Collection` interface, need to implement all these methods!

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- “Correct” solution — provide default implementations in the interface

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- “Correct” solution — provide default implementations in the interface
- Added to Java interfaces later!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, AbstractCollection abstract class implements Collection

```
public abstract class AbstractCollection<E>
                               implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, AbstractCollection abstract class implements Collection
- Concrete classes now extend AbstractCollection
  - Need to define iterator() based on internal representation
  - Can choose to override contains(), ...

```
public abstract class AbstractCollection<E>
                        implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

Concrete collections

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries

Concrete collections

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Collections can be further organized based on additional properties
  - Are the elements ordered?
  - Are duplicates allowed?
  - Are there constraints on how elements are added, removed?

Concrete collections

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Collections can be further organized based on additional properties
  - Are the elements ordered?
  - Are duplicates allowed?
  - Are there constraints on how elements are added, removed?
- In the spirit of indirection, these are captured by interfaces that extend `Collection`
  - Interface `List` for ordered collections
  - Interface `Set` for collections without duplicates
  - Interface `Queue` for ordered collections with constraints on addition and deletion

The List interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access
- Additional functions for random access

```
public interface List<E>
                 extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);
}
```

l[i]

l[i] = v

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access
- Additional functions for random access
- `ListIterator` extends `Iterator`
  - `void add(E element)` to insert an element before the current index
  - `void previous()` to go to previous element
  - `boolean hasPrevious()` checks that it is legal to go backwards

```
public interface List<E>
                extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

The `List` interface and random access

- Random access is not equally efficient for all ordered collections
  - In an array, can compute location of element at index `i`
  - In a linked list, must start at the beginning and traverse `i` links

```
public interface List<E>
                extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

The List interface and random access

- Random access is not equally efficient for all ordered collections
  - In an array, can compute location of element at index i
  - In a linked list, must start at the beginning and traverse i links
- Tagging interface RandomAccess
  - Tells us whether a List supports random access or not
  - Can choose algorithmic strategy based on this

```
public interface List<E>
             extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

```
if (c instanceof RandomAccess) {
  // use random access algorithm
} else {
  // use sequential access algorithm
}
```

The Set interface

- A set is a collection without duplicates

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element

The Set interface

- A set is a collection without duplicates
- Set interface is identical to Collection, but behaviour is more constrained
  - add() should have no effect, and return false, if the element already exists
  - equals() should return true if contents match after disregarding order
- Two interfaces, same signature?
- Use Set to constrain values to satisfy additional constraints

- Set implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints

- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function
- Or arrange values in a two dimensional structure
  - Balanced search tree

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order

Concrete sets

- HashSet implements a hash table
  - Underlying storage is an array
  - Map value v to a position h(v)
  - If h(v) is unoccupied, store v at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if v is at position h(v)
- Unordered, but supports iterator()
- Scan elements in unspecified order
- Visit each element exactly once

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection
- Iterator will visit elements in sorted order

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection
- Iterator will visit elements in sorted order
- Insertion is more complex than a hash table
  - Time $O(\log n)$ if the set has n elements

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

  ```
  boolean add(E element);
  E remove();
  ```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

  ```
  boolean offer(E element);
  E poll();
  ```

  - Return `false` or `null`, respectively, if not possible

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error

- Gentler versions of add(), remove()

```
boolean offer(E element);
E poll();
```

  - Return false or null, respectively, if not possible

- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

  ```
  boolean add(E element);
  E remove();
  ```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error
- Gentler versions of add(), remove()

  ```
  boolean offer(E element);
  E poll();
  ```

  - Return false or null, respectively, if not possible
- Inspect the head, no update

  ```
  E element(); // Throws exception
  E peek();    // Returns null
  ```

- Interface Deque, double ended queue

  ```
  boolean addFirst(E element);
  boolean addLast(E element);
  boolean offerFirst(E element);
  boolean offerLast(E element);
  E pollFirst();
  E pollLast();
  E getFirst();
  E getLast();
  E peekFirst();
  E peekLast();
  ```

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

  ```
  boolean add(E element);
  E remove();
  ```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

  ```
  boolean offer(E element);
  E poll();
  ```

  - Return `false` or `null`, respectively, if not possible
- Inspect the head, no update

  ```
  E element(); // Throws exception
  E peek();    // Returns null
  ```

- Interface `Deque`, double ended queue

  ```
  boolean addFirst(E element);
  boolean addLast(E element);
  boolean offerFirst(E element);
  boolean offerLast(E element);
  E pollFirst();
  E pollLast();
  E getFirst();
  E getLast();
  E peekFirst();
  E peekLast();
  ```

- Interface `PriorityQueue`
  - `remove()` returns highest priority item

The Queue interface

- Ordered, remove front, insert rear
- Queue interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, add() flags an error
  - If queue empty, remove() flags an error
- Gentler versions of add(), remove()

```
boolean offer(E element);
E poll();
```

  - Return false or null, respectively, if not possible
- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

- Interface Deque, double ended queue

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

- Interface PriorityQueue
  - remove() returns highest priority item
- Concrete implementations
  - LinkedList — implements Queue
  - ArrayDeque — circular array Deque

Maps

- The Collection interface abstracts properties of grouped data
    - Arrays, lists, sets, . . .
    - But not key-value structures like dictionaries

Maps

- The Collection interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Key-value structures come under the Map interface
  - Two type parameters
  - K is the type for keys
  - V is the type for values
  - get(k) fetches value for key k
  - put(k,v) updates value for key k

```
public interface Map<K,V>{
  V get(Object key);
  V put(K key, V Value);

  boolean containsKey(Object key);
  boolean containsValue(Object value);
  ...
}
```

Maps

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Key-value structures come under the `Map` interface
  - Two type parameters
  - `K` is the type for keys
  - `V` is the type for values
  - `get(k)` fetches value for key `k`
  - `put(k,v)` updates value for key `k`

```
public interface Map<K,V>{
  V get(Object key);
  V put(K key, V Value);

  boolean containsKey(Object key);
  boolean containsValue(Object value);
  ...
}
```

- As expected, keys form a set
  - Only one entry per key-value
  - Assigning a fresh value to existing key overwrite the old value
  - `put(k,v)` returns the previous value associated with `k`, or `null`

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views

- Can now iterate through a `Map`

```
Set<String> keys = strmap.keySet();
for (String key : keys) {
  do something with key
}
```

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views

- Can now iterate through a `Map`

```
Set<String> keys = strmap.keySet();
for (String key : keys) {
  do something with key
}
```

- Use `entrySet()` to operate on key and associated value without looking up map again

```
for (Map.Entry<String, Employee> entry :
                       staff.entrySet()){
  String k = entry.getKey();
  Employee v = entry.getValue();
  do something with k, v
}
```

Concrete implementations of `Map`

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

Concrete implementations of `Map`

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

`TreeMap`

- Similar to `TreeSet`
- Use a balanced search tree to store keys and values
- Iterator over `keySet()` will process keys in sorted order

Concrete implementations of Map

HashMap

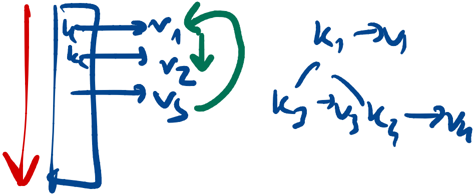
- Similar to HashSet
- Use a hash table to store keys and values
- No fixed order over keys returned by keySet()

TreeMap

- Similar to TreeSet
- Use a balanced search tree to store keys and values
- Iterator over keySet() will process keys in sorted order

LinkedHashMap

- Remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- Iterators over both keySet() and value() enumerate in order of insertion



Concrete implementations of `Map`

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

`TreeMap`

- Similar to `TreeSet`
- Use a balanced search tree to store keys and values
- Iterator over `keySet()` will process keys in sorted order

`LinkedHashMap`

- Remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- Iterators over both `keySet()` and `value()` enumerate in order of insertion
- Can also use access order
    - Each `get()` or `put()` moves key-value pair to end of list
    - Process entries in least recently used order — scheduling, caching

Concrete implementations of Map

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

`TreeMap`

- Similar to `TreeSet`
- Use a balanced search tree to store keys and values
- Iterator over `keySet()` will process keys in sorted order

`LinkedHashMap`

- Remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- Iterators over both `keySet()` and `value()` enumerate in order of insertion
- Can also use access order
  - Each `get()` or `put()` moves key-value pair to end of list
  - Process entries in least recently used order — scheduling, caching
- Similarly, `LinkedHashSet`

When things go wrong

- Our code could encounter many types of errors
  - *User input* — enter invalid filenames or URLs
  - *Device errors* — printer jam, network connection drops
  - *Resource limitations* — disk full
  - *Code errors* — invalid array index, key not present in hash table, refer to a variable that is `null`, divide by zero, . . .

When things go wrong

- Our code could encounter many types of errors
  - *User input* — enter invalid filenames or URLs
  - *Device errors* — printer jam, network connection drops
  - *Resource limitations* — disk full
  - *Code errors* — invalid array index, key not present in hash table, refer to a variable that is `null`, divide by zero, . . .
- Signalling errors
  - Return an invalid value: -1 at end of file, `null`
  - What if there is no obvious invalid value?

Exception handling

- Code that generates error raises or throws an exception

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
    - Information about the nature of the exception
    - Natural to structure an exception as an object

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
  - Extract information about the error from the exception object
  - Graceful interruption rather than program crash

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
  - Extract information about the error from the exception object
  - Graceful interruption rather than program crash
- . . . or passes the exception back up the calling chain

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
  - Extract information about the error from the exception object
  - Graceful interruption rather than program crash
- . . . or passes the exception back up the calling chain
- Declare if a method can throw an exception
  - Compiler can check whether calling code has made a provision to handle the exception

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
  - Array index out of bounds, invalid hash key, ...

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
  - Array index out of bounds, invalid hash key, ...
- Checked exceptions
  - Typically user-defined, code assumptions violated
    - In a list of orders, quantities should be positive integers

Catching and handling exceptions

- try–catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Similar to Python

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

properly

Catching and handling exceptions

- try–catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Similar to Python
- If try encounters an exception, rest of the code in the block is skipped

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- try–catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Similar to Python
- If try encounters an exception, rest of the code in the block is skipped
- If exception matches the type in catch, handler code executes

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- try–catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Similar to Python
- If try encounters an exception, rest of the code in the block is skipped
- If exception matches the type in catch, handler code executes
- Otherwise, uncaught exception is passed back to the code that called this code

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- try–catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Similar to Python
- If try encounters an exception, rest of the code in the block is skipped
- If exception matches the type in catch, handler code executes
- Otherwise, uncaught exception is passed back to the code that called this code
- Top level uncaught exception — program crash

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hiearachy
  - `catch (ExceptionType e)` matches any subtype of `ExceptionType`

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hiearachy
  - `catch (ExceptionType e)` matches any subtype of `ExceptionType`
- Catch blocks are tried in sequence
  - Match exception type against each one in turn

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple catch blocks
- Exceptions are classes in the Java class hiearachy
  - catch (ExceptionType e) matches any subtype of ExceptionType
- Catch blocks are tried in sequence
  - Match exception type against each one in turn
- Order catch blocks by argument type, more specific to less specific
  - IOException would intercept FileNotFoundException

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Generating exceptions

- When does a function generate an exception?

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds, invalid hash key, . . .

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds, invalid hash key, . . .
- Code calls another function that generates an exception

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds, invalid hash key, . . .
- Code calls another function that generates an exception
- Your code detects an error and generates an exception
  - `throw` a checked exception

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length

Notifying checked exceptions

- Example: you write a method `readData()`
    - Header line provides length of data
        - `Content-Length: 2048`
    - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
    - `EOFException`, subtype of `IOException`
    - “Signals that EOF has been reached unexpectedly during input”

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
  - `EOFException`, subtype of `IOException`
  - “Signals that EOF has been reached unexpectedly during input”
- Create an object of exception type and `throw` it

```
throw new EOFException();
```

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
  - `EOFException`, subtype of `IOException`
  - "Signals that EOF has been reached unexpectedly during input"
- Create an object of exception type and `throw` it

```
throw new EOFException();
```

- Can also pass a diagnostic message when constructing exception object

```
String errormsg = "Content-Length:" + contentlen + ", Received: " + rcvdlen;
throw new EOFException(errormsg);
```

Throwing exceptions . . .

- How does caller know that `readData()` generates `EOFException`?

Throwing exceptions ...

- How does caller know that `readData()` generates `EOFException`?
- Declare exceptions thrown in header

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- How does caller know that `readData()` generates `EOFException`?
- Declare exceptions thrown in header
- Can throw multiple types of exceptions

```
String readFile(String filename)
    throws FileNotFoundException,
           EOFException { ... }
```

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- How does caller know that `readData()` generates `EOFException`?
- Declare exceptions thrown in header
- Can throw multiple types of exceptions

```
String readFile(String filename)
    throws FileNotFoundException,
           EOFException { ... }
```

- Can throw any subtype of declared exception type

```
String readFile(String filename)
    throws IOException { ... }
```

  - Can throw `FileNotFoundException`, `EOFException`, both subclasses of `IOException`

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception

```
String readData(Scanner in)
  throws EOFException {
 ...
 while (...) {
  if (!in.hasNext()) {
   // EOF encountered
   if (n < len) {
    String errmsg = ...
    throw new EOFException(errmsg);
   }
  ...
 }
 return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception
- Need not advertise unchecked exceptions
  - `Error`, `RunTimeException`

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception
- Need not advertise unchecked exceptions
  - `Error`, `RunTimeException`
- Should not normally generate `RunTimeException`
  - Fix the error or report suitable checked exception

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Customized exceptions

- Don’t want negative numbers in a `LinearList`

Customized exceptions

- Don't want negative numbers in a LinearList
- Define a new class extending Exception

```
public class NegativeException extends Exception{

  private int error_value;
   // Negative value that generated exception

  public NegativeException(String message, int i){
    super(message);  // Appeal to superclass
    error_value = i; // constructor to set message
  }

  public int report_error_value(){
    return error_value;
  }
}
```

Customized exceptions

- Don't want negative numbers in a LinearList
- Define a new class extending Exception
- Throw this from LinearList
    - Note that add advertises the fact that it throws a NegativeException

```
public class NegativeException extends Exception{
   ...
}

public class LinearList{
  ...
  public add(int i) throws NegativeException{
    ...
    if (i < 0){
      throw new NegativeException("Negative input",i);
    }
    ...
  }
}
```

More on catching exceptions

- Can extract information about the exception

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  String errormsg = e.getMessage();
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`

```
try {
  ...
  access database
  ..
}
catch (SQLException e){
  ...
  String errormsg =
     "database error" + e.getMessage();
  throw new ServletException(errormsg);
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`
- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`, `initCause()`

```
try {
  ...
  access database
  ..
}
catch (SQLException e){
  ...
  String errormsg =
     "database error" + e.getMessage();
  throw new ServletException(errormsg);
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`
- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`, `initCause()`
- Add information when you chain exceptions

```
try {
  ...
  access database
  ..
}
catch (SQLException e){
  ...
  String errormsg =
     "database error" + e.getMessage();
  ServletException newe =
     new ServletException(errormsg);
  newe.initCause(e);
  throw newe;
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`
- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`, `initCause()`
- Add information when you chain exceptions
- Retrieve information when you catch exception

```
try {
  ...
}
catch (ServletException e){
  ...
  Throwable original = e.getCause();
  ...
}
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled `finally`

```
try{
     ...
   }

catch (ExceptionType1 e){...}

catch (ExceptionType2 e){...}

finally{
     ...
  // Always executed, whether try
  // terminates normally or
  // exceptionally. Use for clean up.
}
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled `finally`
- Different scenarios

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled `finally`
- Different scenarios
  - No error — 1,2,5,6

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Cleaning up resources

- When exception occurs, rest of the try block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled finally
- Different scenarios
  - No error — 1,2,5,6
  - IOException in try, no exception in catch — 1,3,4,5,6

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, ...)
- Add a block labelled `finally`
- Different scenarios
  - No error — 1,2,5,6
  - `IOException` in `try`, no exception in `catch` — 1,3,4,5,6
  - `IOException` in `try`, chained exception in `catch` — 1,3,5

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```