Java: callbacks, iterators, generics

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 8, 6 February 2025

Interfaces

- An interface is a purely abstract class
  - All methods are abstract
- A class implements an interface
  - Provide concrete code for each abstract function
- Classes can implement multiple interfaces
  - Abstract functions, so no contradictory inheritance
- Interfaces describe relevant aspects of a class
  - Abstract functions describe a specific "slice" of capabilities
  - Another class only needs to know about these capabilities

private class

Interfaces express relevant capabilities

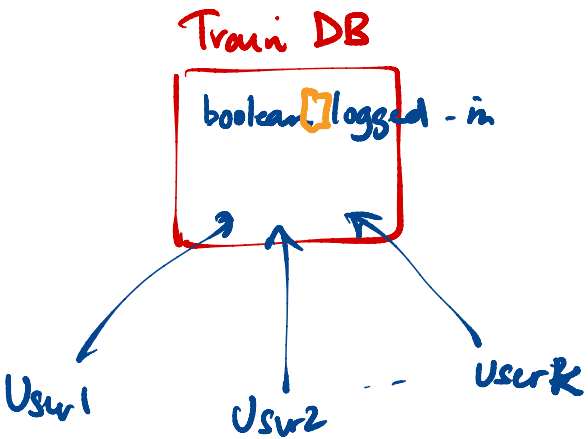
- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant
- Describe the relevant functions supported by `Comparable` objects through an interface
- However, we cannot express the intended behaviour of `cmp` explicitly

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

```
public interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

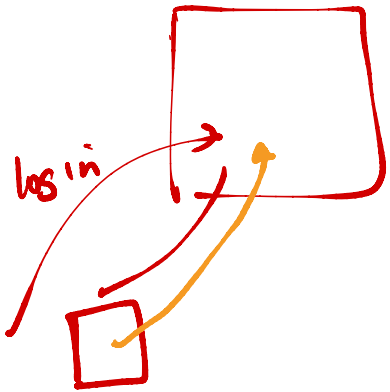
Interactions with state

- Connect database query to logged in status of the user



Interactions with state

- Connect database query to logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`



Interactions with state

- Connect database query to logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?

Interactions with state

- Connect database query to logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?
- Use an interface!
  - Interface describes the capability of the object returned on login

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
      qobj = new QueryObject();
      return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
      ...
    }
  }
}
```

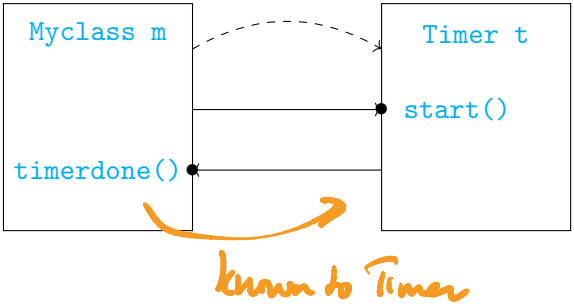
Interactions with state . . .

- Query object allows unlimited number of queries
- Limit the number of queries per login?
- Maintain a counter
  - Add instance variables to object returned on login
  - Query object can remember the state of the interaction

```
public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    private int numqueries;
    private static int QLIM;

    public int getStatus(int trainno, Date d){
      if (numqueries < QLIM){
        // respond, increment numqueries
      }
    }
  }
}
```

Implementing a call-back facility

- `Myclass m` creates a `Timer t`
- Start `t` to run in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!
- `Timer t` notifies `Myclass m` when the time limit expires
  - Assume `Myclass m` has a function `timerdone()`



Implementing callbacks

- Code for `Myclass`

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ..
  }

  public void timerdone(){...}
}
```

Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

Owner

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ..
    owner.timerdone();
    // I'm done
  }
}
```

Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel
- `Timer` specific to `Myclass`
- Create a generic `Timer`?

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Generic timer — use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

Generic timer — use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

- Modify `Myclass` to implement `Timerowner`

```
public class Myclass
   implements Timerowner{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

Generic timer — use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

- Modify `Myclass` to implement `Timerowner`
- Modify `Timer` so that `owner` is compatible with `Timerowner`

```
public class Myclass
   implements Timerowner{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
      implements Runnable{
  // Timer can be
  // invoked in parallel
  private Timerowner owner;

  public Timer(Timerowner o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Linear list

- A generic linear list of objects

Linear list

- A generic linear list of objects
- Internal implementation may vary

Linear list

- A generic linear list of objects
- Internal implementation may vary
- An array implementation

```
public class Linearlist {
  // Array implementation
  private int limit = 100;
  private Object[] data = new Object[limit];
  private int size;  // Current size

  public Linearlist(){ size = 0; }

  public void append(Object o){
    data[size] = o;
    size++;
    ...
  }
  ...
}
```

Linear list

- A generic linear list of objects
- Internal implementation may vary
- An array implementation
- A linked list implementation

```
public class Linearlist {
  private Node head;
  private int size;

  public Linearlist(){ size = 0; }

  public void append(Object o){
    Node m;

    for (m = head; m.next != null; m = m.next){}
    Node n = new Node(o);
    m.next = n;

    size++;
  }
  ...
  private class Node (...}
}
```

Iteration

- Want a loop to run through all values in a linear list

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this
- We don't have public access . . .

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this
- We don't have public access . . .
- . . . and we don't know which implementation is in use!

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iterators

- Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

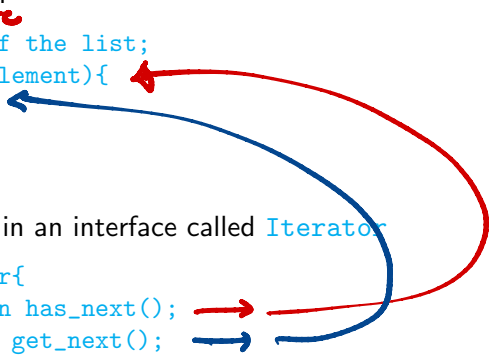
Iterators

- Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

- Encapsulate this functionality in an interface called `Iterator`

```
public interface Iterator{
  public abstract boolean has_next();
  public abstract Object get_next();
}
```



Iterators

- How do we implement `Iterator` in `Linearlist`?

Iterators

- How do we implement `Iterator` in `Linearlist`?
- Need a "pointer" to remember position of the iterator

Iterators

- How do we implement `Iterator` in `Linearlist`?
- Need a "pointer" to remember position of the iterator
- How do we handle nested loops?

```
for (i = 0; i < data.length; i++){
  for (j = 0; j < data.length; j++){
     ... // do something with data[i] and data[j]
  }
}
```

Iterators

- Solution: Create an `Iterator` object and export it!

Iterators

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{

  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }

 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return(it);
  }
 }
```

Iterators

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{

  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }

 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return(it);
  }
 }
```

- Definition of `Iter` depends on linear list

Iterators

- Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator();

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
```

Iterators

- Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator();

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
```

- For nested loops, acquire multiple iterators!

```
Linearlist l = new Linearlist();
...
Object oi,oj;
Iterator i,j;

i = l.get_iterator();
while (i.has_next()){
  oi = i.get_next();
  j = l.get_iterator();
  while (j.has_next()){
    oj = j.get_next();
    ... // do something with oi, oj
  }
}
...
```

Iterators

- Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator();

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
```

- The new Java `for` over lists implicitly constructs and uses an iterator

```
for (type x : a)
  do something with x;
}
```

- For nested loops, acquire multiple iterators!

```
Linearlist l = new Linearlist();
...
Object oi,oj;
Iterator i,j;

i = l.get_iterator();
while (i.has_next()){
  oi = i.get_next();
  j = l.get_iterator();
  while (j.has_next()){
    oj = j.get_next();
    ... // do something with oi, oj
  }
}
...
```

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities
  - Reverse an array/list
  - Search for an element in an array/list
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list (need to compare values)

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list (need to compare values)

[a]

a → b

Ord a ⇒ a → a

Structural polymorphism

- Use the Java class hierarchy to simulate this

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`

```
public void reverse (Object[] objarr){
  Object tempobj;
  int len = objarr.length;
  for (i = 0; i < n/2; i++){
    tempobj = objarr[i];
    objarr[i] = objarr[(n-1)-i];
    objarr[(n-1)-i] = tempobj;
  }
}
```

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`
- Polymorphic `find`
  - `==` translates to `Object.equals()`

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length; i++){
    if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`
- Polymorphic `find`
  - `==` translates to `Object.equals()`
- Polymorphic `sort`
  - Use interfaces to capture capabilities

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}

public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

Type consistency

- Polymorphic function to copy an array

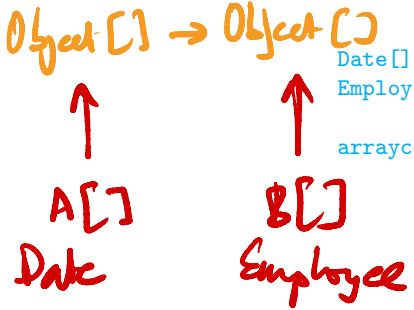
```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

fails at runtime




```
Date[] datearr = new Date[10];
Employee[] emparr = new Employee[10];

arraycopy(datearr,emparr); // Run-time error
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time
- More generally source array can be a subtype of the target array

ETktarr → Tktarr ✓

Tktarr → ETktarr ✗

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

```
public class Ticket {...}
public class ETicket extends Ticket{...}

Ticket[] tktarr = new Ticket[10];
ETicket[] etktarr = new ETicket[10];

arraycopy(etktarr,tktarr); // Allowed
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time
- More generally source array can be a subtype of the target array
- But the converse is illegal

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}


public class Ticket {...}
public class ETicket extends Ticket{...}

Ticket[] tktarr = new Ticket[10];
ETicket[] etktarr = new ETicket[10];

arraycopy(tktarr,etktarr); // Illegal
```

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval;
    ...
    return(returnval);
  }

  public void insert(Object newdata){...}

  private class Node {
    private Object data;
    private Node next;
    ...
  }
}
```

?

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval;
    ...
    return(returnval);
  }

  public void insert(Object newdata){...}

  private class Node {
    private Object data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems
  - Type information is lost, need casts

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){...}

  private class Node {...}
}
```

```
LinkedList list = new LinkedList();
Ticket t1,t2;

t1 = new Ticket();
list.insert(t1);
t2 = (Ticket)(list.head());
// head() returns an Object
```

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems
  - Type information is lost, need casts
  - List need not be homogenous!


```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){...}

  private class Node {...}
}
```

```
LinkedList list = new LinkedList();
Ticket t = new Ticket();
Date d = new Date();
list.insert(t);
list.insert(d);
...
```

Generic Programming = Structural Polym.

- Use type variables

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."

```
public <T> void reverse (T[] objarr){
  T tempobj;
  int len = objarr.length;
  for (i = 0; i < n/2; i++){
    tempobj = objarr[i];
    objarr[i] = objarr[(n-1)-i];
    objarr[(n-1)-i] = tempobj;
  }
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` . . . "
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error

```
public <T> int find (T[] objarr, T o){
  int i;
  for (i = 0; i < objarr.length; i++){
    if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error
- Polymorphic `arraycopy`
  - Source and target types must be identical

```
public static <T> void arraycopy (T[] src,
                                  T[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` . . . "
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error
- Polymorphic `arraycopy`
  - Source and target types must be identical
- A more generous `arraycopy`
  - Source and target types may be different
  - Source type must extend target type

```
public static <S extends T,T>
              void arraycopy (S[] src,
                              T[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Two type variable with some constraint

Polymorphic Sorting

public <T implements Comparable> sort (T[] ...)

∴ T.comp() exists

Polymorphic data structures

- A polymorphic list

Parametrized list

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`
- Instantiate generic classes using concrete type

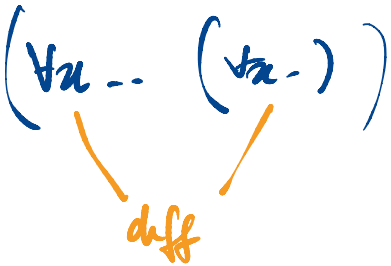
```
public class LinkedList<T>{
    ...
}

LinkedList<Ticket> ticketlist =
          new LinkedList<Ticket>();
LinkedList<Date> datelist =
          new LinkedList<Date>();

Ticket t = new Ticket();
Date d = new Date();

ticketlist.insert(t);
datelist.insert(d);
```

Concrete type for T

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```




```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

different (arbitrary) T

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```

- `T` in the argument of `insert()` is a new `T`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```

- `T` in the argument of `insert()` is a new `T`
- Quantifier `<T>` masks the type parameter `T` of `LinkedList`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```

- `T` in the argument of `insert()` is a new `T`
- Quantifier `<T>` masks the type parameter `T` of `LinkedList`
- Contrast with

```
public <T> static void
   arraycopy (T[] src, T[] tgt){...}
```

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

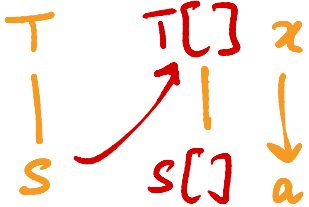
Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
// Not OK.  ticketarr[5] refers to an ETicket!
```



Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
// Not OK.  ticketarr[5] refers to an ETicket!
```

- A type error at run time!

"Bad thing"

Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
// Not OK.  ticketarr[5] refers to an ETicket!
```

- A type error at run time!
- Java array typing is covariant
  - If `S extends T` then `S[] extends T[]`

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`


```
public class LinkedList<T>{...}

public static void printlist(LinkedList<Object> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<Object> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- How can we get around this limitation?

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

```
public class LinkedList<T>{...}

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

```
public class LinkedList<T>{...}

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- `<T>` is a type quantifier: *For every type* `T`, ...

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

```
public class LinkedList<T>{...}

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- `<T>` is a type quantifier: *For every type* `T`, ...
- Note that `T` is not actually used inside the function
  - We use `Object o` as a generic variable to cycle through the list

Wildcards

- Instead, use ? as a wildcard type variable

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

Wildcards

- Instead, use ? as a wildcard type variable

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

- ? stands for an arbitrary unknown type

Wildcards

- Instead, use ? as a wildcard type variable

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

- ? stands for an arbitrary unknown type
- Avoids unnecessary type variable quantification when the type variable is not needed elsewhere

Wildcards

- Can declare wildcard instance of a generic class, but usability is limited

```
public class LinkedList<T>{...}

LinkedList<?> l;
```

Wildcards

- Can declare wildcard instance of a generic class, but usability is limited

```
public class LinkedList<T>{...}

LinkedList<?> l;
```

- Can extract an iterator from wildcard instance of `l` ...

```
LinkedList<?> l = new LinkedList<String>();
i = l.get_iterator();
```

Wildcards

- Can declare wildcard instance of a generic class, but usability is limited

```
public class LinkedList<T>{...}

LinkedList<?> l;
```

- Can extract an iterator from wildcard instance of `l` ...

```
LinkedList<?> l = new LinkedList<String>();
i = l.get_iterator();
```

- ... but cannot add elements to `l`

```
LinkedList<?> l = new LinkedList<String>();
l.add(new Object()); // Compile time error
```

  - Compiler cannot guarantee the types match

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`

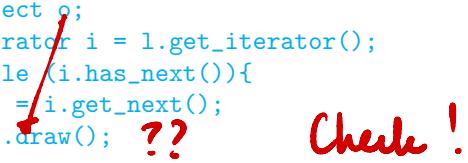
Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`
- All subclasses override `draw()`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`
- All subclasses override `draw()`
- Want a function to draw all elements in a list of `Shape` compatible objects

```
public static void drawAll(LinkedList<? extends Shape> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    o.draw();
  }
}
```



Bounded wildcards

- Copying a `LinkedList`, using a wildcard

```
public static <? extends T,T>
       void listcopy (LinkedList<?> src,
                      LinkedList<T> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    tgt.add(o);
  }
}
```

Bounded wildcards

- Copying a `LinkedList`, using a wildcard

```
public static <? extends T,T>
       void listcopy (LinkedList<?> src,
                      LinkedList<T> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    tgt.add(o);
  }
}
```

- Can reverse the constraint, using `super`

```
public static <T,? super T>
       void listcopy (LinkedList<T> src,
                      LinkedList<?> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    tgt.add(o);
  }
}
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed
- An alternative approach is to do type inference

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed
- An alternative approach is to do type inference
- Derive type information from context. For instance, `s` should be `String`

```
s = "Hello, " + "world";
```

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed
- An alternative approach is to do type inference
- Derive type information from context. For instance, `s` should be `String`

```
s = "Hello, " + "world";
```

- Propagate type information: now `t` is also `String`

```
t = s + 5;
```

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```

  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```

  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```

- More ambitious?

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```

  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```

- More ambitious?
  - If `x.bonus()` is legal, `x` must be `Manager` rather than `Employee`

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}

...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```

  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```

- More ambitious?
  - If `x.bonus()` is legal, `x` must be `Manager` rather than `Employee`
- Keep track of and validate type obligations

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}

...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference

- Assume program is type-safe, derive most general types compatible with code
  - Use information from constants to determine type
  - Propagate type information based on already inferred types

```
public class Employee {...}

public class Manager extends Employee {
    ...
    public double bonus (...) {...}
}

...

public static f(Employee x){
   ...
   double d = x.bonus(...);
     // x must be a Manager?
   ...
}
```

Type inference

- Assume program is type-safe, derive most general types compatible with code
  - Use information from constants to determine type
  - Propagate type information based on already inferred types
- Typing judgements should ideally be made at compile-time, not at run-time
  - Static analysis of code

```
public class Employee {...}

public class Manager extends Employee {
    ...
    public double bonus (...) {...}
}

...

public static f(Employee x){
   ...
   double d = x.bonus(...);
     // x must be a Manager?
   ...
}
```

Type inference

- Assume program is type-safe, derive most general types compatible with code
  - Use information from constants to determine type
  - Propagate type information based on already inferred types
- Typing judgements should ideally be made at compile-time, not at run-time
  - Static analysis of code
- Balance flexibility with algorithmic tractability

```
public class Employee {...}

public class Manager extends Employee {
    ...
    public double bonus (...) {...}
}

...

public static f(Employee x){
   ...
   double d = x.bonus(...);
     // x must be a Manager?
   ...
}
```

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value

```
var b = false;  // boolean

var s = "Hello, world";  // String
```

3.6 → double

3.6f → float

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value
- Be careful about format for numeric constants

```
var b = false;  // boolean

var s = "Hello, world";  // String

var d = 2.0;  // double

var f = 3.141f; // float
```

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value
- Be careful about format for numeric constants
- For classes, infer most constrained type
  - `e` is inferred to be `Manager`
  - `Manager` extends `Employee`
  - If `e` should be `Employee`, declare explicitly

```
var b = false;  // boolean

var s = "Hello, world";  // String

var d = 2.0;  // double

var f = 3.141f; // float

var e = new Manager(...);  // Manager
```