Java: generics

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 7, 30 January 2024

Java Generics

- Use type variables

Java Generics

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

∀T (but not scalar types)

```
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;
  }
}
```

[a] → a

[a] → b

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);
}
```

Also need ==, but Object defines this

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];
  }
}
```

Polymorphic data structures

- A polymorphic 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

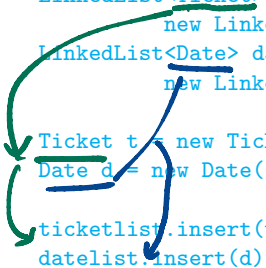
- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`

```
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);
```



Polymorphic data structures

- Be careful not to accidentally hide a type variable

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

$\forall x (\ldots x \ldots \wedge (\forall x (\ldots)$

```
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`

```
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!

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[]
```

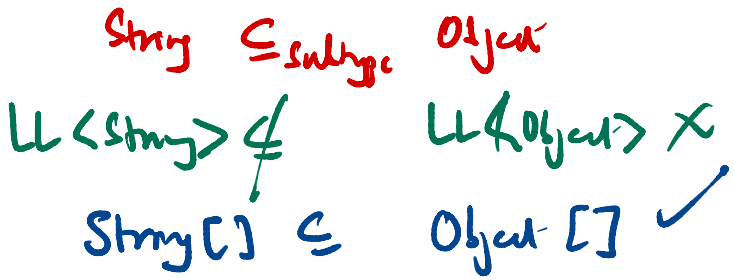
- But ...

```
...
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()`

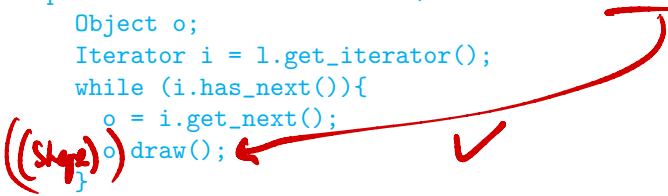
Bounded wildcards

- 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();
    ((Shape) 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
```

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

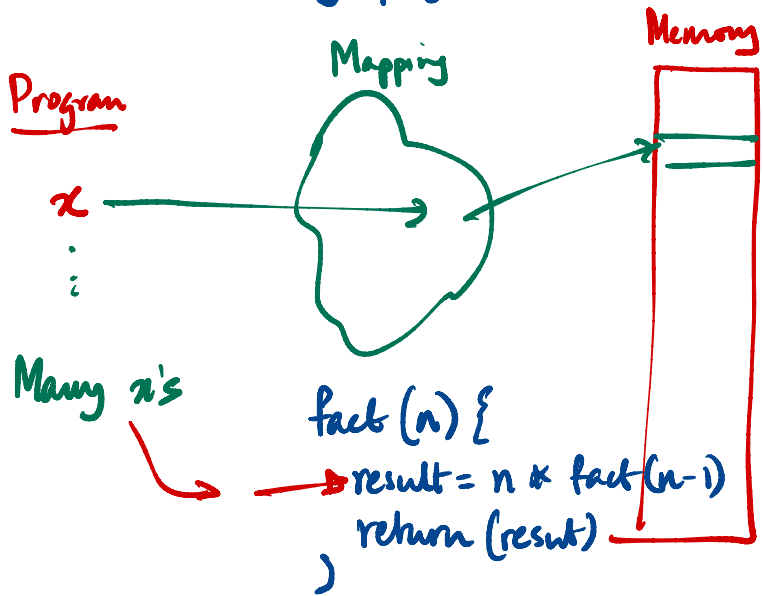
```
var b = false;  // boolean

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

var d = 2.0;  // double

var f = 3.141f; // float

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

Generics implementation → Type Erasure

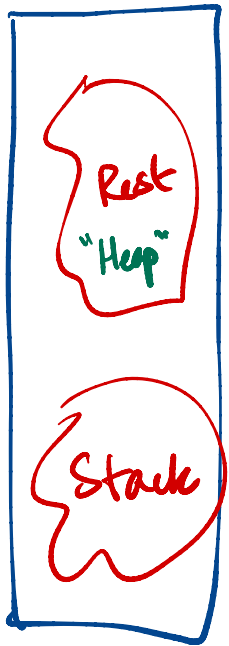
X<T> → X<Object>

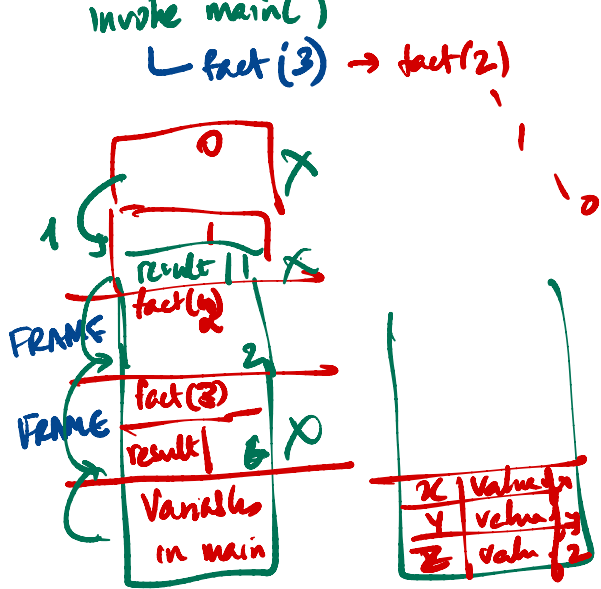
X<? extends Shape> → X<Shape>

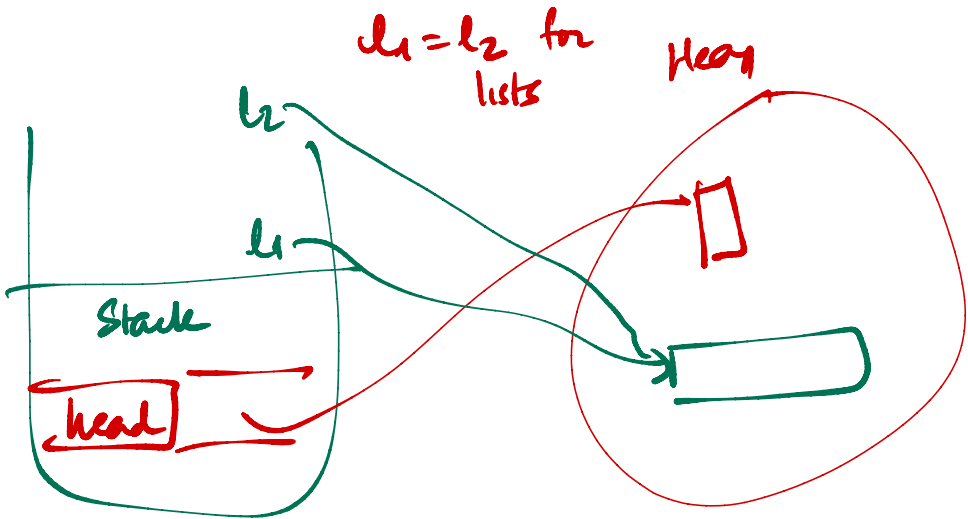
How memory is used by programs



Memory


Rest
"Heap"
Stack


invoke main()
fact(3) → fact(2)
1
0
0
1
result
fact(2)
FRAME
2
fact(3)
FRAME
result
6
Variables in main
x
y
z


l1 = l2 for lists
Heap
l2
l1
Stack
head