# Java: generics

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 10, 9 February 2023

# 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`


Simula

```
for e in Eventqueue:
    e.simulate()
```

# 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

[a] → [a]

[a → b]

[a] → [a] ?

# 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)

# 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);
}
```

always "works"

# 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];
  }
}


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

```
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 {...}
}
```

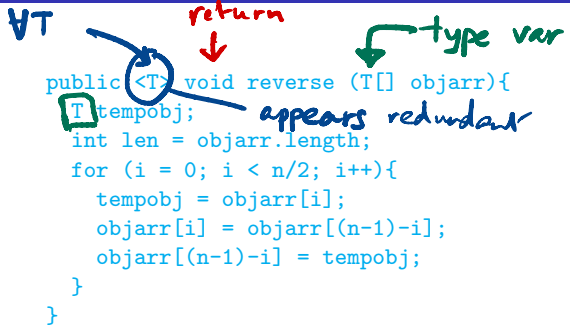
```
LinkedList list = new LinkedList();
Ticket t = new Ticket();
Date d = new Date();
list.insert(t);
list.insert(d);
...
```

# Java Generics

- Use type variables

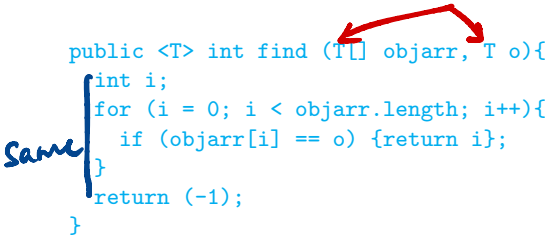
# Java Generics

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



# 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



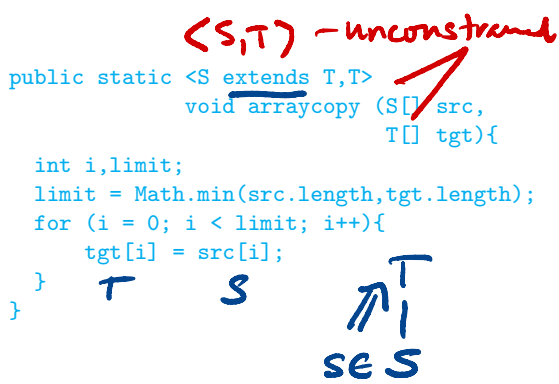
# Java Generics

- 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



# 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

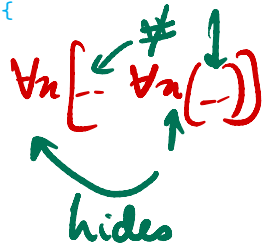
```
public class LinkedList<T>{
    ...
}

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){...}
```

```
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

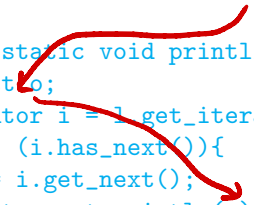
- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`

```
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 define variables of a wildcard type

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

LinkedList<?> l;
```

# Wildcards

- Can define variables of a wildcard type

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

LinkedList<?> l;
```

- But need to be careful about assigning values

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

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

# Wildcards

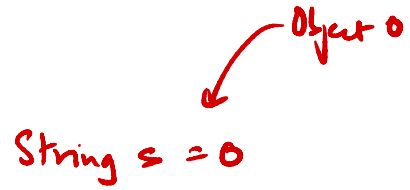
- Can define variables of a wildcard type

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

LinkedList<?> l;
```

- But need to be careful about assigning values

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

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();
    trt.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();
    trt.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();
    trt.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

int x = 5;

# 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
```

Employee e = new Manager()