Dynamic dispatch and polymorphism

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclasses cannot see private components of parent class
- Subclass can add more instance variables and methods

```
public class Employee{
  private String name;
  private double salary;

  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }
  public String getName(){ ... }
  public double getSalary(){ ... }

  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}

public class Manager extends Employee{
  private String secretary;
  public boolean setSecretary(name s){ ... }
  public String getSecretary(){ ... }
}
```

Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclasses cannot see private components of parent class
- Subclass can add more instance variables and methods
- Can also override methods

```
public class Employee{
  private String name;
  private double salary;

  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }
  public String getName(){ ... }
  public double getSalary(){ ... }

  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}

public class Manager extends Employee{
  private String secretary;
  public boolean setSecretary(name s){ ... }
  public String getSecretary(){ ... }
}
```

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

- What about `e.bonus(p)`? Which `bonus()` do we use?
  - Static: Use `Employee.bonus()`
  - Dynamic: Use `Manager.bonus()`

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

- What about `e.bonus(p)`? Which `bonus()` do we use?
  - Static: Use `Employee.bonus()`
  - Dynamic: Use `Manager.bonus()`
- Dynamic dispatch (dynamic binding, late method binding, ...) turns out to be more useful
  - Default in Java, optional in languages like C++ (`virtual` function)

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Recall the event simulation loop that motivated Simula to introduce objects

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Recall the event simulation loop that motivated Simula to introduce objects
- Also referred to as runtime polymorphism or inheritance polymorphism

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays
- Made possible by overloaded methods defined in class `Arrays`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Type casting

- Consider the following assignment

```
Employee e = new Manager(...)
```

Type casting

- Consider the following assignment

  ```
  Employee e = new Manager(...)
  ```

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- A simple example of reflection in Java
  - "Think about oneself"

Type casting

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`

```
((Manager) e).setSecretary(s)
```

- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- A simple example of reflection in Java
  - "Think about oneself"
- Can also use type casting for basic types

```
double d = 29.98;
int nd = (int) d;
```

Summary

- A subclass can override a method from a parent class
- Dynamic dispatch ensures that the most appropriate method is called, based on the run-time identity of the object
- Run-time/inheritance polymorphism, different from overloading
  - We will later see another type of polymorphism, structural polymorphism
  - For instance, use the same sorting function for array of any datatype that supports a comparison operation
  - Java uses the term *generics* for this
- Use type-casting (and reflection) overcome static type restrictions