G51PRG: Introduction to Programming Second semester

Lecture 5

Natasha Alechina
School of Computer Science & IT
nza@cs.nott.ac.uk

Previous lecture

- Overriding methods
- Shadowing/hiding fields
- Polymorphism

Plan of the lecture

- ***final*** keyword
- Casting objects
- More on polymorphism
- Object class
- Wrapper classes

Overriding methods; *final* keyword

- Class B extends A
- B inherits methods from A
- B may override (change implementations of) methods from A.
- Some methods cannot be overridden:
  - private methods;
  - static methods;
  - methods declared as ***final*** .
- A class declared as ***final*** cannot be extended.

final keyword

- You already saw ***final*** used in declarations of variables which cannot be changed in the program (e.g. constants).

```
final double pi = 3.14;
…
pi = 4.0;
```

will cause a compiler error.

- Another use of ***final*** is to prevent overriding methods/changing class definitions.
- Reasons for doing this: preventing errors, efficiency and (later in the lecture) security.

Casting

- Casting is explicitly converting from one data type to another data type.
- For example, we can cast an int as a double or a double as an int (loosing information in the latter case).
- We can also cast one Object type as another.

Widening conversions (basic types)

- Widening conversions: from a type which uses less memory to a type which uses more memory. For example, from byte to int, from int to long, float or double. No information gets lost.

```
char a = `a`;
int i = (int) a;
```

- May even skip casting:

```
int i = a;
```


Narrowing conversions (basic types)

- Narrowing conversions: from a type which uses more memory to a type which uses less. Information may get lost, so in general such casts should be avoided.

```
char c = (char) 10000000000000000;
// … bad things happen
```


Casting objects

- Suppose class B extends A
- Casting up the case hierarchy: a B type object as an A type object - like a widening conversion - easy.

```
B bobj = new B(10);
A aobj = (A) bobj;
// don't even need explicit cast
A aobj = bobj; // is OK
```

- Casting down, an A type object (parent) as a child - like a narrowing conversion, tricky. Information may get lost.

Polymorphism: general rule

- Class B extends A
- all Bs are a special kind of As
- you can use Bs where As are expected.

Using Pixels where Points are expected

```
Pixel pixel = new Pixel(5,6, Color.red);
Point p = pixel;
```

- Normally you assign an object of type Point to a variable of type Point.
- But you can also assign a Pixel to a Point type variable.
- Here we omit casting because pixel is promoted up the class hierarchy (Point is a parent class and pixel is promoted to Point).

Using Points where Pixels are expected

```
Pixel pixel = new Pixel(5,6, Color.red);
Point [] points = new Point[4];
points[0] = pixel;
Pixel q = (Pixel) points[0];
```

- To the compiler `q = points[0]` would look like you are trying to assign a less informative object (Point) to a more informative type variable (Pixel). That's why you need to put in an explicit cast.
- The compiler does not keep track of the ***actual*** type of objects, for example that points[0] is in fact a Pixel.
- The runtime system does.

Runtime errors

```
Point [] points = new Point[4];
points[0] = new Point(5,6);
Pixel q = (Pixel) points[0];
```

- This will pass the compiler, same as the code before.
- The runtime system will throw ClassCastNotSupportedException.
- Casting “real” Pixel to a Pixel will work.
- Casting a Point to a Pixel will not work.

Declared type and runtime type.

```
Point p = new Pixel(4,5,Color.blue);
```

- declared type of p is Point. Actual type is Pixel.

```
Point[] points = new Point[4];
points[0] = new Pixel(5,6, Color.red);
```

- declared type of **`points[0]`** is Point. Actual type is Pixel.
- Declared type A: declaration of variable type is A; being returned by a method whose return type is A. Compiler checks declared types.
- Actual (runtime) type: which constructor was used to construct the object. If constructor was B, then actual type is B.

When casting succeeds

```
B newobj = (B) obj;
```

- Casting up the class hierarchy (B to B or up to A) always succeeds.
- Casting down the class hierarchy (A down to B) will succeed at run time if **`obj`** ’s actual type is B or a more specific type.
- Otherwise the casting will fail (throw an exception).

Which implementation is used?

- If we look at Pixels and Points again, they both have clear() method with different implementation (one sets color field to null and another one does not).
- Which implementation is run if declared type of an object does not match the actual type?
- When invoking a method on an object, ***actual class*** of the object governs which implementation is used:

```
Point p = new Pixel(5,6);
p.clear()
```

will use Pixel's version of clear().

Dynamic binding

- To find out which implementation to use, runtime system has to keep track of the object's real type (the class lowest in the hierarchy of which the object is an instance).
- This is useful, but introduces an overhead.

Using *final* for efficiency reasons

- If a method is declared final then there is no need to look for its correct implementation at run time.
- Type checks become faster and can be done at compile time. For final methods sometimes invocation can be replaced with the actual body of the method (inlining), e.g.:

```
System.out.println(rose.getName());
```

- can be replaced with

```
System.out.println(rose.name);
```

- if we know that **`getName()`** just returns the value of the name field. Same for final fields: replace **`pi`** with 3.14.

Using *final* for security reasons

- For example, a **`validatePassword()`** method which returns a boolean depending on whether password is correct or not, should be declared final. Otherwise someone may subclass its class to make a new version of the method which always returns true.
- Fields which final methods rely on should be also final or private, otherwise the method's behaviour can be changed by changing the fields.

Access modifiers summary

- Increasing order of access:
  - private
  - default (same package or directory)
  - protected = as default + subclasses
  - public
- When subclassing, access modifiers should only be changed to make more access.

Object class

- All classes extend the Object class and therefore inherit its methods.
- Default constructor for the Object class is Object() which sets all fields in an object to default values; reference type fields such as Strings to null, numbers to 0.
- There are two groups of methods in the Object class: general utility methods and methods which support threads.

Some utility methods of Object

- **`Object clone() throws CloneNotSupportedException`** - creates and returns a copy of this object; usually need to override it.
- **`boolean equals(Object o)`** - indicates whether some other object is equal to this one; usually need to override it.
- **`Class getClass()`** - returns the runtime class of an object.

Some utility methods of Object

- **`int hashCode()`** - returns a hash code value for the object. (Which is a unique number for this object. This is typically implemented by converting the internal address of the object into an integer.)
- **`String toString()`** - returns a string representation of the object. The **`toString()`** method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. So usually it is overridden to something more useful...

Example

```
class Person {
 String name;
 int age;
 Person(String s, int n) {
  this.name = new String(s);
  this.age = n;
 }
 public Object clone() {
  return new Person(this.name, this.age);
 }}
```

Example: remarks

- Person extends Object (don't need to put it in the class definition);
- Strings are passed by reference, that's why we duplicate them in constructor
- we override the Object's clone() method in the Person class. Note that it still returns value of type Object
- when we override a method we don't change its signature.
- Why we override it rather than write a new

`Person clone()`

- method: because lots of code written for Objects rely on them having the right clone() method.

Hashtable class

- Hash table holds data items indexed by keys .
- Key is used to access the value, just as an array index is used to access the corresponding element in the array.
- Hash table keys can be of any reference type (for example, Strings).

hash(key1)	(key1, value1)
hash(key2)	(key2, value2)

Example

indices	(key,value) pairs
hash("john")	(john, 9150001)
hash("adam")	(adam, 9510010)

Hashtable methods

- `Object put(Object key, Object value)`
- `Object get(Object key)` - returns the value
- `boolean containsKey(Object key) -` returns true or false depending on whether there is an item with such key in the table
- `boolean containsValue(Object value) -` returns true or false depending on whether there is such a value in the table
- `Object remove(Object key) -` removes item with the specified key

Example Hashtable

```
Hashtable telephones = new Hashtable();
// put some telephones in a table
telephones.put("john", "9150001");
telephones.put("adam", "9510010");
String tel1 =
(String) telephones.get("john");
// will return John's telephone number
// note that get() returns an Object so
// need to cast to String!
```


What about storing basic types?

- The trouble with Java Collections (such as Hashtable) is that they are designed to store Objects.
- Hashtable works with ***any*** objects: Strings, Points, Persons,… . But it does not have methods to store integers or doubles.
- There is a workaround though - you can use Wrapper classes.

Envelopes, or wrapper classes

- We can't cast between basic types and objects.
- What do we do if we need to use a basic type where an object is required?
- Define a new class and put the basic type value inside it as a field. That's what "wrappers" or "envelopes" do: make an object out of a basic type.
- There are Java classes for all basic types: byte, float and so on: Boolean, Character, Byte, Short, Integer, Long, Float, Double.

Two purposes of wrapper classes

- Make it possible to use basic types with classes which handle objects. For example, we cannot store ints or use them as keys in Hashtable, but can store an object of type Integer.
- Provide home for useful methods associated with the basic type. For example, Integer class has method

```
static int parseInt(String s)
```

- which given a string "2002" returns number 2002.

Some common methods of wrapper classes

- A constructor which takes the primitive type and creates an object of the type class (e.g. **`Character(char c)`**);
- **`xxxxValue()`** (where xxxx is the primitive type, e.g. **`Character.charValue()`** and **`Boolean.booleanValue()`**) returns the value of the basic type stored in the object.

Telephone example

- If we wanted to store telephones as ints:

```
Hashtable tels = new Hashtable();
tels.put("john", new Integer(9150001));
tels.put("adam", new Integer(9510010));
```

- Which way to get tel number as an int is correct?

```
int tel1 = (tels.get("john")).intValue();
int tel2 = (Integer) tels.get("john");
int tel3 =
((Integer) tels.get("john")).intValue();
```


Next lecture

- Abstract classes
- Interfaces
- Collections in Java

Summary and further reading

- Polymorphism allows to relax type checking
- Occasionally need casting. Casting to a less specific type (up the class hierarchy) is easy. Casting down may cause errors.
- Run time system keeps track of the actual type of each object and chooses appropriate method implementation.
- Sun Java tutorial:

http://java.sun.com/docs/books/tutorial/java/javaOO/index.html