ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of January 26, 2013

Abstract

- Examples of polymorphism:
  - Object: equals, toString;
  - Generic data structures: Pair.

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

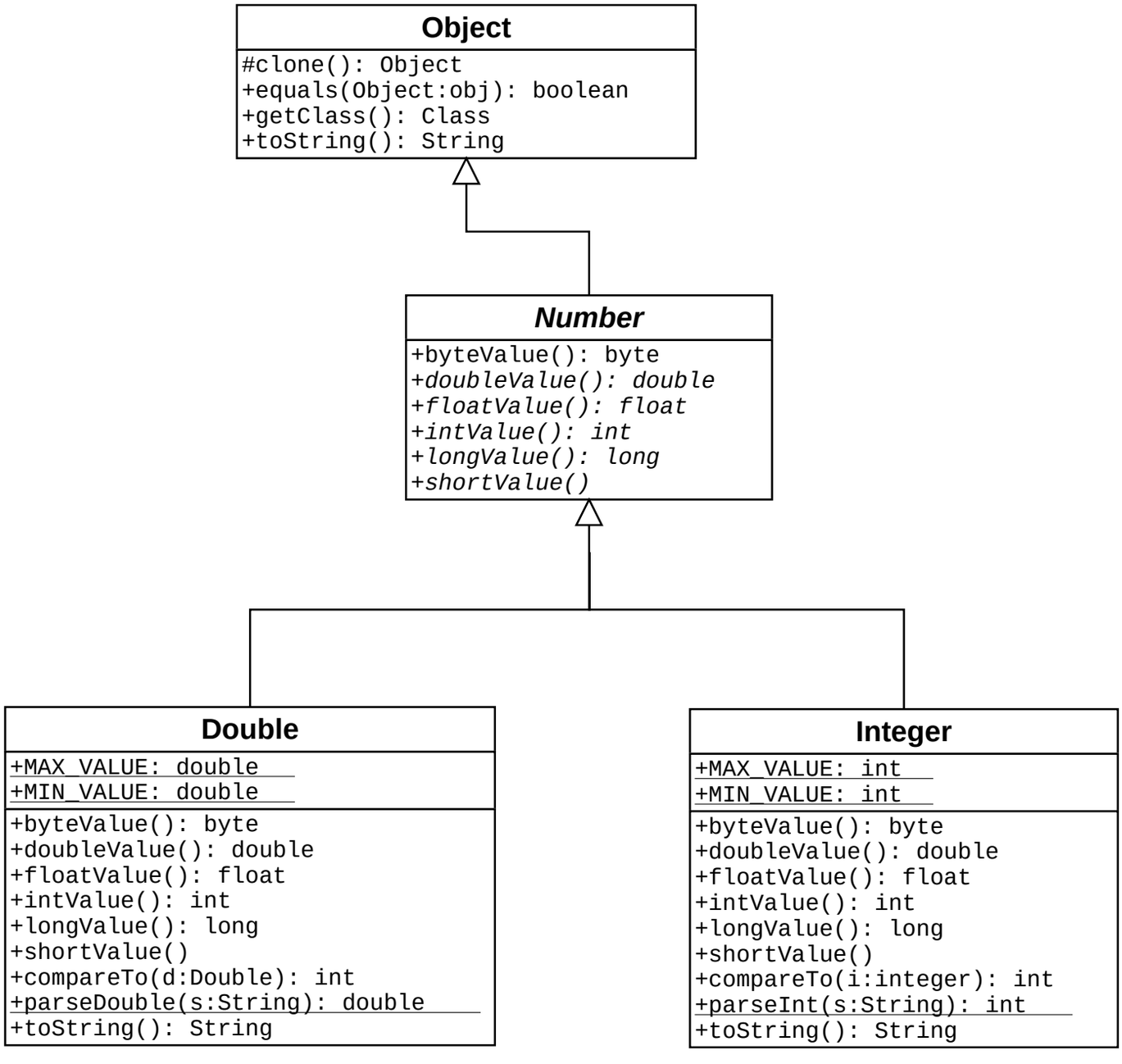
Object

What about this?

```
Object o;
```

A reference of type **Object** can be used to designate any object!

What are the methods declared in the class **Object**?


Object
#clone(): Object
+equals(Object:obj): boolean
+getClass(): Class
+toString(): String
Number
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
Double
+MAX_VALUE: double
+MIN_VALUE: double
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
+compareTo(d:Double): int
+parseDouble(s:String): double
+toString(): String
Integer
+MAX_VALUE: int
+MIN_VALUE: int
+byteValue(): byte
+doubleValue(): double
+floatValue(): float
+intValue(): int
+longValue(): long
+shortValue()
+compareTo(i:integer): int
+parseInt(s:String): int
+toString(): String

Object/equals

The class **Object** defines a method **equals**.

Hence, for any two references **a** and **b**, the following is always valid.

```
if ( a.equals( b ) ) { ... }
```

This is true for the predefined classes, such as **String** and **Integer**, but also for any class that you will define.

All the classes are directly or indirectly sub-classes of **Object**.

Equality

Consider two references, **a** and **b**, and the designated objects.

- comparing the **identity** of two objects — do a and b designate the same object (i.e. `a == b`);

- comparing the **content** of the designated objects — often, the user wants to know if the content of the two objects is the same, use the method **equals**; **logical equality**.

equals

Is possible to define a method **equals** that works for any pairs of objects?

The method **equals** from the class **Object** is defined as follows.

```
public boolean equals( Object obj ) {
    return ( this == obj );
}
```

Account

```
public class Account {
    private int id;
    private String name;

    public Account( int id, String name ) {
        this.id = id;
        this.name = name;
    }
}
```

Test

```
class Test {
    public static void main( String[] args ) {
        Account a, b;
        a = new Account( 1, new String( "Marcel" ) );
        b = new Account( 1, new String( "Marcel" ) );
        if ( a.equals( b ) ) {
            System.out.println( "a and b are equals" );
        } else {
            System.out.println( "a and b are not equals" );
        }
    }
}
```

What is the outcome?

“a and b are not equals”

Solution: redefine the method **equals** in the class **Account**!

Recipe for the method equals (1/4)

Use == to check that the reference parameter is not **null**.

```
public class Account {
    private int id;
    private String name;
    public Account( int id, String name ) {
        this.id = id;
        this.name = name;
    }
    public boolean equals( Object o ) {
        boolean result = true;
        if ( o == null ) { // <---
            result = false;
        } ...
        return result;
    }
}
```

Recipe of the method equals (2/4)

Use **instanceof** to check that the object designated by the parameter is of the same class as this object.

```
public class Account {
    private int id;
    private String name;
    public Account( int id, String name ) {
        this.id = id;
        this.name = name;
    }
    public boolean equals( Object o ) {
        boolean result = true;
        if ( o == null ) {
            result = false;
        } else if ( ! ( o instanceof Account ) ) { // <---
            result = false;
        } ...
        return result;
    }
}
```

Recipe of the method equals (3/4)

Since **o** is not null and designates an object of the class **Account**, let's use a variable of type **Account** to designate this object.

```
public class Account {
    private int id;
    private String name;
    public Account( int id, String name ) { ... }
    public boolean equals( Object o ) {
        boolean result = true;
        if ( o == null ) {
            result = false;
        } else if ( ! ( o instanceof Account ) ) {
            result = false;
        } else {
            Account other = (Account) o; // <---
            ...
        }
        return result;
    }
}
```

Recipe of the method equals (4/4)

For each significant field make sure they are equal. For **primitive** type use ==, for **reference** type use **equals**, but be careful with **null** references!

```
public class Account {
    private int id;  private String name;
    public Account( int id, String name ) { ... }
    public boolean equals( Object o ) {
        boolean result = true;
        if ( o == null ) {
            result = false;
        } else if ( ! ( o instanceof Account ) ) {
            result = false;
        } else {
            Account other = (Account) o;
            if ( id != other.id ) {
                result = false;
            } else if ( name == null && other.name != null ) {
                result = false;
            } else if ( name != null && ! name.equals( other.name ) ) {
                result = false;
            }
        }
        return result;
    }
}
```

Test

```
class Test {
    public static void main( String[] args ) {
        Account a, b;
        a = new Account( 1, new String( "Marcel" ) );
        b = new Account( 1, new String( "Marcel" ) );
        if ( a.equals( b ) ) {
            System.out.println( "a and b are equals" );
        } else {
            System.out.println( "a and b are not equals" );
        }
    }
}
```

Now displays:

“a and b are equals”

print

Usage:

```
System.out.print( 10 );

System.out.print( true );

System.out.print( 'c' );

System.out.print( "vote today" );

System.out.print( new Time( a, b, c ) );
```

print

```
System.out.print( ... );
```

Firstly, since the standard notation has been used, we know that **print** is the instance method of an object designated by the variable **out** of the class **System**.

The class variable **out** designates a **PrintStream** object.

print

Secondly, the class **PrintStream** provides us with several examples of overloading (*ad hoc* polymorphism).

```
public static void print( boolean b ) {
  if ( b ) {  print( "true" );  }
  else { print( "false"); }
}
public static void print( char c ) {
  print( String.valueOf( c ) );
}
public static void print( int i ) {
  print( Interger.toString( i ) )
}
...
```

One declaration per primitive type.

print

But also, a polymorphic method that can be applied to any **Object**.

```
public static void print( Object obj ) {
  if ( obj == null ) {
       print( "null" );
  else {
       print( obj.toString() ); // <---
  }
}
public static void print( String s ) {
  ...
}
```

What is this method **toString()**?

print

What is this method **toString()**?

It's defined in the class **Object** so that **obj.toString()** is valid for any reference variable **obj**.

toString() is **inherited** or **redefined**.

Let's try this on the **Account** example.

```
Account a;
a = new Account( 1, "Marcel");
System.out.print( a );
```

Displays:

```
Account@863399
```

print

Displays:

```
Account@863399
```

The method **toString** in the class **Object** is defined as follows.

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Hum . . . not so useful.

Most of the time, you'll want to overwrite the method **toString()**.

print

```
public class Account {

    private int id;
    private String name;

    public Account( int id, String name ) {
        this.id = id;
        this.name = name;
    }
    public String toString() {
        return "id = " + id + "; name = " + name;
    }
}
```

```
Account a;
a = new Account( 1, "Marcel");
System.out.print( a );
```

Displays:

```
id = 1; name = Marcel
```

instanceof and type cast

```
Shape s;
...
if ( s instanceof Circle ) {
   Circle c;
   c = (Circle) s;
   double radius = c.getRadius();
}
```

Always preceded by **instanceof**!

```
long l;
...
if ( ( Integer.MIN_VALUE <= l ) && ( l <= Integer.MAX_VALUE ) ) {
   int i;
   i = (int) l;
   ...
}
```

instanceof

Given the following expression,

```
s instanceof T
```

The operator **instanceof** returns **false** if the the value of **s** is **null**, otherwise, the operator returns **true** if the class of the object designated by **s**, at runtime, is compatible with the type **T**; either the same type or the object designated by **s** is of a subclass of **T**.

instanceof and type cast

```
if ( s instanceof Circle ) {
   double radius = ( (Circle) s ).getRadius();
}
```

Data structures and polymorphism

Let's consider developing a simple class to store a pair of specific objects, say **Time** objects.

What are the instance variables? What are the methods?

Time Pair

```
public class Pair {
  private Time first;
  private Time second;
  public Pair( Time first, Time second ) {
    this.first = first;
    this.second = second;
  }
  public Time getFirst( ) {
      return first;
  }
  public Time getSecond( ) {
      return second;
  }
}
```

⇒ `new Pair( new Time( 14, 30 ), new Time( 16, 0 ) );`

Shape Pair

```
public class Pair {
  private Shape first;
  private Shape second;
  public Pair( Shape first, Shape second ) {
    this.first = first;
    this.second = second;
  }
  public Shape getFirst( ) {
      return first;
  }
  public Shape getSecond( ) {
      return second;
  }
}
```

⇒ `new Pair( new Circle( 0, 0, 0 ), new Rectangle( 1, 1, 1, 1 );`

Object and "generic" data structures

Write a class that can be used to store a pair of objects.

What are the attributes of this class?

Pair

```
public class Pair {
  private ______ first;
  private ______ second;
  public Pair( ______ first, ______ second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( ______ first, ______ second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( Object first, Object second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {  // type of the return value?
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair( Object first, Object second ) {
    this.first = first;
    this.second = second;
  }
  public Object getFirst( ) {
      return first;
  }
  public Object getSecond( ) {
      return second;
  }
}
```

Pair

How would you use this class.

```
Pair p;

String a;
a = "King'';
String b;
b = "Edward";

p = new Pair( a, b );

a = p.getFirst();
b = p.getFirst();
```

Problems?

Pair

```
Pair p;

String a;
a = "King'';
String b;
b = "Edward";

p = new Pair( a, b );

a = (String) p.getFirst();
b = (String) p.getFirst();
```

Object is more general than **String**! Whenever, an element is removed from a generic data structure, the reference must be cast to the original type of the object.

Generic data structures before Java 1.5

Before Java 1.5, creating a "generic" data structure always involved declaring attributes of type **Object**.

The **advantage** of generic data structures is that a single implementation can be used in a variety of contexts (any object that can be created in Java can be stored in such data structure).

The **drawback** is that the type of the return value of the access methods (getters) is **Object**, consequently requiring type casting.

No static type checking! Errors might occur at runtime!

Generic data structures before Java 1.5

```
Time t;
Pair p;
...
t = (Time) p.getFirst();
```

This is dangerous because the compiler can no longer do one of its important tasks, validating the type of the expressions, hence runtime errors are more likely to occur.

Generic data structures and 1.5

Java 1.5 offers a new concept called **Generics** introducing class parameters.

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

This declaration (the parameter) represents the type of the objects to be stored in a **Pair**.

Generics

The value of the parameter is specified in the type declaration.

```
Pair<String> name;
Pair<Integer> range;
```

as well as in the format of the expression for creating an object.

```
name = new Pair<String>( "Hilary", "Clinton" );
```

```
Integer min;
min = new Integer( 0 );
Integer max;
max = new Integer( 100 );

range = new Pair<Integer>( min, max );
```

Generics

What has been achieved?

Type casts are no longer necessary.

```
Pair<Integer> range;
range = new Pair<Integer>( new Integer( 0 ), new Integer( 100 );
Integer i;
i = range.getFirst();
```

Generics

Now, a **Pair** declared using the following type declaration **Pair**<**Integer**> can only be used to store **Integer** objects.

The statement,

```
range.setFirst( "Voila" );
```

causes a compile-time error.

```
Test.java:20: setFirst(java.lang.Integer)
in Pair<java.lang.Integer> cannot be applied to (java.lang.String)
        range.setFirst( "Voila" );
              ^
1 error
```

Generics

Similarly, a reference variable that has the following type **Pair**<**Integer**> cannot be used to designate objects of type (class) **Pair**<**String**>.

The statement,

```
range = new Pair<String>( "Hilary", "Clinton" );
```

causes the following compile-time error,

```
Test.java:22: incompatible types
found   : Pair<java.lang.String>
required: Pair<java.lang.Integer>
        range = new Pair<String>( "Hilary", "Clinton" );
                ^
1 error
```

Generics

It's a win/win situation.

A data structure, such as **Pair**, has a unique implementation (is implemented once) and yet can be used in several contexts (to store objects from any class).

All this, without sacrificing the important type validation step done at compile-time.

Generic and Parameterized Types

"A **generic type** is a type with formal type parameters. A **parameterized type** is an instantiation of a generic type with actual type arguments."

Generic and Parameterized Types

Defining a generic type.

A **generic type** is a **reference type** that has one or more type parameters.

A **generic type** is a **class** that has one or more type parameters.

Defining a generic type

```
public class Pair<X,Y> {

    private X first;
    private Y second;

    public Pair( X a, Y b ) {
        first = a;
        second = b;
    }

    public X getFirst() { return first; }

    public Y getSecond() { return second; }

    public void setFirst( X arg ) { first = arg; }

    public void setSecond( Y arg ) { second = arg; }
}
```

Creating a parameterized type

When using generic types, here **Pair**, the **type arguments** of the **type parameters** must be provided.

```
public class Test {
    public static void main( String[] args ) {

        Pair<String, Integer> p;

        String attribut;
        attribut = new String( "height" );

        Integer value;
        value = new Integer( 100 );

        p = new Pair<String, Integer>( attribut, value );

    }
}
```

Quizz: are these valid statements?

```
Pair<String,Integer> p;
```

```
p = new Pair<String,Integer>();
```

```
p.setFirst( "session" );
```

```
p.setSecond( 12345 );
```

Quizz: are these valid statements?

```
public class T1 {
    public static void main( String[] args ) {

        Pair<String,Integer> p;

        p = new Pair<Integer,String>();

    }
}


// > javac T1.java
// T1.java:6: incompatible types
// found   : Pair<java.lang.Integer,java.lang.String>
// required: Pair<java.lang.String,java.lang.Integer>
//      p = new Pair<Integer,String>();
//             ^
// 1 error
```

Quizz: are these valid statements?

```
public class T2 {
    public static void main( String[] args ) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>();
        p.setFirst( 12345 );
        p.setSecond( "session" );
    }
}
```

```
T2.java:5: setFirst(java.lang.String) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (int)
        p.setFirst( 12345 );
         ^
T2.java:6: setSecond(java.lang.Integer) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (java.lang.String)
        p.setSecond( "session" );
         ^
2 errors
```

Quizz: are these valid statements?

```
public class T3 {
    public static void main( String[] args ) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>( "session", 12345 );
        Integer s = p.getFirst();
    }
}


// > javac T3.java
// T3.java:8: incompatible types
// found   : java.lang.String
// required: java.lang.Integer
//      Integer s = p.getFirst();
//                              ^
// 1 error
```

Generics

Generics add new tools to detect errors at compile time!

Someone (something) watching over your shoulder!

Implementation

Briefly, generics are implemented by **erasure**, which means that the type parameters are present at compile-time only.

Generic types are present at compile-time, where they are used for type checking, but are removed (erased) to produce the compiled (byte-code) representation of the program.

A.java

```
public class A {

    public static void main( String[] args ) {

        Pair<String, Integer> p;
        p = new Pair<String, Integer>( "Orange", new Integer( 1 ) );

        String s;
        s = p.getFirst();

    }
}
```

B.java

```
public class B {

    public static void main( String[] args ) {

        Pair p;
        p = new Pair( "Orange", new Integer( 1 ) );

        String s;
        s = (String) p.getFirst();

    }
}
```

Type erasure

A.java:

```
Pair<String> p;
p = new Pair<String, Integer>( "Orange", 1 );

String s;
s = p.getFirst();
```

B.java:

```
Pair p;
p = new Pair( "Orange", 1 );

String s;
s = (String) p.getFirst();
```

Type erasure

A.java and **B.java** are both producing the same byte-code. This can be seen by first compiling the two classes, then comparing the result of **javap -c** on the two classes (**javap** is the Java class file disassembler).

> "Generics implicitly perform the same cast that is explicitly performed without generics."

It comes with a guarantee:

> "Cast-iron guarantee: the implicit casts added by the compilation of generics never fail."

[NW07, page 5]

Type erasure

- Keeps everything simple and back compatible;
- Economical: only one byte-code for all its usages

Wrappers

Before Java 1.5, it was impossible to save values from primitive types directly into a generic data structure.

```
Pair p;
p = new Pair( 0, 100 );
```

Wrappers

Wrappers had to be used.

```
Pair p;
p = new Pair( new Integer( 0 ), new Integer( 100 ) );
```

Java 1.5

Java 1.5 automatic mechanisms to wrap/unwrap (box/unbox) values of primitive types in the corresponding wrapper class.

```
Pair<Integer> range;

range = new Pair<Integer>( 0, 10 );

int i;

i = range.getSecond();
```

References

[NW07] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly, 2007.