03-60-440: OO Programming

Jianguo Lu

University of Windsor

Object-Oriented Programming

- Object orientation
  - Abstract data type
  - Polymorphism
  - Inheritance

Data Type and Abstract Data Type

- Data type
  - Data values
  - Operations on the data
- Abstract
  - Focus on some details while ignore others.
    - Simplified description of objects
    - Emphasis significant information only
    - Suppress irrelevant information
- Abstract Data Type
  - Focus on operations, ignore the concrete data representation.

Abstract Data Type (ADT)

- One of the most important programming concepts introduced in the 1970s.
- Separation of the use of the data type from its implementation
- Users of an ADT are concerned with the interface, but not the implementation, as the implementation can change in the future.
  - User's point of view: determines what operations can be done to a variable
  - Implementer's point of view:
    - a restriction on the kinds of values a variable can store
    - determines how much memory it requires
- This supports the principle of information hiding
  - protecting the program from design decisions that are subject to change.

Javadoc for Stack

- Method Summary
- boolean empty()
  - Tests if this stack is empty.
- E peek()
  - Looks at the object at the top of this stack without removing it from the stack.
- E pop()
  - Removes the object at the top of this stack and returns that object as the value of this function.
- E push(E item)
  - Pushes an item onto the top of this stack.

ADT Example

Stack();
Stack push(Object item);
Stack pop();
Object peek();
boolean empty();

for all [s: Stack; i: Object]

Stack().empty() = true
s.push(i).empty()=false

Stack().pop()=error
s.push(i).pop()=s;

Stack().peek()=error
s.push(i).peek()=i;

(Note that this is not the Stack in Java)

- ADT defines a data type in terms of operations instead of the data
- The implementation of Stack data type can be an Array, or a Vector, or other data structure
- This is the idea of information hiding
  - Implementation is not relevant to the user of the data type
  - Implementation can be changed without affecting other parts of the system.
- The meaning of ADT can be specified in terms of the relationships between the operators, independent of the data representation.

Algebraic specification of Stack and Queue

ADT

QUEUE

sorts: QUEUE, INT, BOOLEAN

operations:

new: --> QUEUE
add: QUEUE x INT --> QUEUE

empty: QUEUE --> BOOLEAN
del: QUEUE --> QUEUE
head: QUEUE --> INT U { error }

Semantics

empty(new()=true
emtpty(add(q,i))=false

del(New())=error
del(add(q,i))=if (empty(q)) then new() else
add(del(q),i)

head(new())=error
head(add(q,i))=if (empty(q)) then i
else head(q)

STACK

sorts: STACK, INT, BOOLEAN

operations:

new: --> STACK
push: STACK x INT --> STACK

empty: STACK --> BOOLEAN
pop: STACK --> STACK
top: STACK --> INT U { error }

Semantics

empty(new()) = true
empty(push(S, i)) = false

pop(new()) = error
pop(push(S, i)) = S

peek(new()) = error
peek(push(S,i)) = i

Polymorphism

- Kinds of polymorphism
  - Overload
  - Coercion
  - Subtype polymorphism
  - Generics

- Overriding
- Static vs. dynamic binding

Polymorphism

polymorphism

- Polymorphism: “the quality or state of being able to assume different forms”—webster
  - E.g., one species in several forms
- in computer programming, one type is able to assume different forms
  - One object can assume different forms
  - One method can assume different forms
- Types of polymorphism in programming
  - Ad hoc
    - Overloading
    - Coercing
  - Universal
    - Inclusion (subtype) polymorphism: achieves polymorphic behavior through an inclusion relation between types.
    - Parametric polymorphism (Generics): a function or datatype can be written generically so that it can deal equally well with any objects without depending on their type

Motivating example

- Operator + may be used in different ways:
  1) 1 + 2 → 3
  2) 3.14 + 0.0015 → 3.1415
  3) 1 + 3.7 → 4.7
  4) [1, 2, 3] + [4, 5, 6] → [1, 2, 3, 4, 5, 6]
  5) [true, false] + [false, true] → [true, false, false, true]
  6) "foo" + "bar" → "foobar"
- To handle these six function calls, four different pieces of code are needed:
  - in 1) integer addition must be invoked.
  - in 2), 3) floating-point addition must be invoked.
  - in 4), 5) list concatenation must be invoked.
  - in 6) string concatenation must be invoked.

Overloading

- The name “+” actually refers to four completely different functions. This is an example of overloading.
- Overloading: use same function (or method, operator) name for different functions as long as the parameters differ.
- Note the function signatures and bodies are different

  1 + 2 → 3

  "foo" + "bar" → "foobar"

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be different
  - The signature includes the number, type, and order of the parameters.
- The compiler determines which version of the method is being invoked by analyzing the parameters
- e.g., `println` method is overloaded:
  - println(String s)
  - println(int i)
  - println(double d)
- The following lines invoke different versions of the `println` method:

```
System.out.println("The total is:");
System.out.println( 12 );
```

- Constructors are often overloaded
  - Overloaded constructors provide multiple ways to initialize a new object

Overloading methods: user defined methods

Version 1

```
float tryMe (int x) {

    return x + .375;
}
```

Version 2

```
float tryMe (int x, float y) {

    return x * y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```

Whether the following program will compile?

```
public class Overloading {
    void doSomething (int k) {
      System.out.println("doSomething int method");
    }
    int doSomething(int k){
      return k;
    }
}

Overloading o=new Overloading();
o.doSomething(1);
```

Compiler will report duplicate method declaration.

Coercion

- For the following example,

  1 + 3.7 → 4.7

  “no”+1 →no1

- There is no function for adding an integer to a floating-point number (nor string to an integer).
- Since an integer can be converted into a floating-point number without loss of precision, 1 is converted into 1.0 and floating-point addition is invoked.
- When the compiler finds a function call f(a1, a2, ...) that no existing function named f can handle, it tries to convert the arguments into different types in order to make the call conform to the signature of one of the functions named f.
- This is called coercion. Both coercion and overloading are kinds of ad-hoc polymorphism.

What is the printout of the following code

- System.out.println(1/2);
  - 0
- System.out.println(1.0*1/2);
  - 0.5
- System.out.println(1/2+1.0);
  - 1.0
- System.out.println(1/2*1.0);
  - 0.0

Ad-hoc polymorphism

- Overloading and coercing are ad-hoc polymorphisms
- The name refers to the manner in which this kind of polymorphism is typically introduced: "Oh, hey, let's make the + operator work on strings, too!"
- ad-hoc polymorphism is just syntactic sugar for calling add_integer, append_string, etc., manually. One way to see it is that
  - To the user: there is only one function, but one that takes different types of input and is thus type polymorphic;
  - To the author, there are several functions that need to be written—one for each type of input—so there's essentially no polymorphism.

Parametric polymorphism

- An example of parametric polymorphism:

  [1, 2, 3] + [4, 5, 6] → [1, 2, 3, 4, 5, 6]

  [true, false] + [false, true] → [true, false, false, true]

- The reason why we can concatenate both lists of integers, lists of booleans, is that the function for list concatenation was written without any regard to the type of elements stored in the lists.

  List<T> concat (List<T> x, List<T> y)

- You could make up a thousand different new types of lists, and the generic list concatenation function would accept instances of them all.

A similar concept: overriding

- When a class redefines an inherited method, the new method *overrides* the inherited method.
  - Normally the overriding method is a more specific version for a particular derived class
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

```
public class A {
    void doSomething () {
      System.out.println("A");
    }
}
public class B extends A{
    void doSomething () {
    System.out.println("B”);
}
   A a=new A();
   a.doSomething();
```

A

```
   B b = new B ();
    b.doSomething ();
```

B

```
   A ab = new B ();
   ab.doSomething ();
```

B

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

```
class Parent {
   public int x = 12;
   public int get() {return x;};
}
class Child extends Parent {
   public int x = 42; // shadows variable from
   parent class
   public int get() {return x;};
}
public class Override {
   public static void main(String args[]) {
        Parent p = new Parent();
        Child c= new Child();
        System.out.println(p.x+"  " +c.x);
        p = c;
        System.out.println(p.x+"  " +c.x);
        System.out.println(p.get()+" “ +c.get());
   }
}//override.java
```

```
12  42
12  42
42  42
```

Comparing overloading and overriding

- The place to declare the methods
  - Overloading: multiple methods with the same name in the same class.
  - Overriding: methods in inheritance hierarchy.
- Signature of the methods
  - Overloading: with different signatures
  - Overriding: with the same signature
- Purpose:
  - Overloading: define a similar operation for different input parameters
  - Overriding: redefine an operation of its super class.

Polymorphism

- Overloading
- Coercion
- Subtype
- Parametric

Subtype polymorphism

Motivating example for subtype polymorphism

- Suppose that you have a Dog class and an Interrogator class:

```
class Dog {
    void talk() {
       System.out.println("Woof!");
    }
}

class Interrogator {
static void makeItTalk(Dog subject){
          subject.talk();
    }
}
```

- Interrogate the dog:

```
Dog dog=new Dog();
Interrogator.makeItTalk(dog);
```

> “Woof”

- Then you have a Cat class that you also want to interrogate:

```
class Cat {
      void talk() {
         System.out.println("Meow.");
      }
  }

Cat cat = new Cat();
Interrogator.makeItTalk(cat);
```

> “Meow”

A solution not very good—use overloading

```
class Interrogator {
      static void makeItTalk(Dog subject) {
          subject.talk();
       }
   static void makeItTalk(Cat subject) {
          subject.talk();
       }
}
```

- Interrogate the dog and cat:

```
Dog d=new Dog();
Cat c=new Cat();
Interrogator.makeItTalk(d);
Interrogator.makeItTalk(c);
>"Woof"
>"Meow"
```

- Related concepts in this program
  - *Overloading:* MakeItTalk is an overloaded method
  - *Static binding:* The binding of the method name to the method definition is static, i.e., at compile time.

Problem with this approach

- What if there are other types of subjects you want to interrogate?

```
class Bird extends Animal {
   void talk() {
      System.out.println("Tweet, tweet!");
   }
}

Bird b=new Bird();
Interrogator.makeItTalk(b);
```

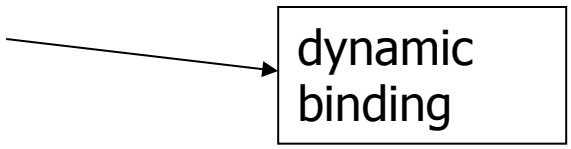
- There would be many repeated code (overloaded makeItTalk() methods) when there are many classes that you want interrogate

```
static void makeItTalk(Dog subject) { …}
static void makeItTalk(Cat subject) {…}
static void makeItTalk(Bird subject) {…}
… …
```

- There should be a better approach

Subtype polymorphism


```
abstract class Animal {
 abstract void talk();
 }

 class Dog extends Animal {
    void talk() {
       System.out.println("Woof!");
    }
 }

 class Cat extends Animal {
    void talk() {
       System.out.println("Meow.");
    }
 }

class Interrogator {
   static void makeItTalk(Animal subject) {
       subject.talk();
    }
}
```



- Dynamic binding

  subject.talk()

  - At compile time, compiler doesn't know which class of object is passed to makeItTalk()
  - JVM decides at runtime which method to invoke based on the class of the object.

```
Animal animal=new Dog();
Interrogator.makeItTalk(animal);
```

```
static void makeItTalk(Dog subject) { …}
static void makeItTalk(Cat subject) {…}
static void makeItTalk(Bird subject) {…}
```

Make it more polymorphic

- Suppose that there are another hierarchy of classes that also need to be interrogated:

```
class Clock { …}

class AlarmClock extends Clock {
    public void talk() {
        System.out.println(“Beep!");
    }
}


Interrogator.makeItTalk(alarmClock);
```

- First attempt:

  class AlarmClock extends Animal

  – It lost the inheritance from Clock class

  – Logically AlarmClock is not a subclass of Animal

- Second attempt:

  class AlarmClock extends Animal, Clock

  – Multiple inheritance is not allowed in Java

Using Interface

- Interface: A collection of constants and abstract methods that cannot be instantiated.
- A class implements an interface by providing method implementations for each of the abstract methods defined in the interface.

```
interface Talkative {
    void talk();
}

class AlarmClock extends Clock implements Talkative {
    public void talk() {
        System.out.println("Beep!");
    }
}
```

Must implement the abstract method(s) in interface

Make it more polymorphic

```
interface Talkative {
        void talk();
}


abstract class Animal implements
    Talkative {
    …
}


class Clock {
    … }


class AlarmClock extends Clock
    implements Talkative {
    public void talk() {
         System.out.println(“Beep!");
    }
}
```

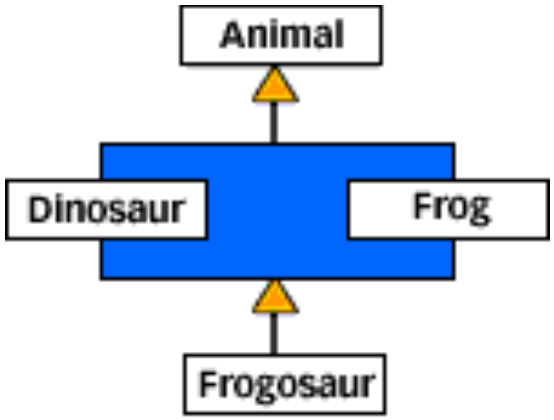
- Now the same makeItTalk() method can interrogate AlarmClock

```
AlarmClock ac = new AlarmClock();
Interrogator.makeItTalk(ac);
```

- This is also an example of multiple inheritance simulated by interface
  - Sometimes called interface multiple inheritance

Multiple inheritance


```
abstract class Animal {
   int age;
   abstract void talk();
}

class Frog extends Animal {
   float age;
   void talk() {   System.out.println("Ribit, ribit."); }
}

class Dinosaur extends Animal {
   void talk() {
      System.out.println("Oh I'm a dinosaur and I'm OK...");
   }
}

class Frogosaur extends Frog, Dinosaur { …
}

Animal animal = new Frogosaur();
animal.talk();

animal.age;
```



Incorrect in Java

Problems with multiple inheritance

- Regarding methods:
  - It isn't clear whether the runtime system should invoke Frog's or Dinosaur's implementation of talk().
    - Will a Frogosaur croak "Ribbit, Ribbit." or sing "Oh, I'm a dinosaur and I'm okay..."?
- Regarding instance variables:
  - When a variable in defined in both Frog and Dinosaur, which copy of the variable -- Frog's or Dinosaur's -- would be selected?

```
Animal animal = new Frogosaur();
animal.talk();

animal.age;
```

Problem with multiple inheritance regarding methods

- Two methods in two super classes can have the same name and signature
- but they can have a different code body.
- Which code body does the subclass inherit?

```
class Frog extends Animal {
   void talk() {
      System.out.println("Ribit, ribit.");
    }
}

class Dinosaur extends Animal {
   void talk() {
      System.out.println("Oh I'm a dinosaur and I'm OK...");
   }
}
```

Problems with variables in multiple inheritance


```
Class A {
  int x;
}
Class B {
  String x;
}

Class S extends A, B{
  x= 10;
  x="inheritance";
}
```

- The compiler won't be able to know which x to inherit in class S

Why Interface solves the problems

```
Interface A {
  static final int x=10;
}
Interface B {
  static final String x="test";
}

Class S implements A, B{
  A.x
  B.x
}
```

- Interface declares constants only
- the compiler does not generate bytecode instructions to allocate memory for that variable
- If two different interfaces declare a constant with the same name, but with a different type and/or a different initial value, and if a class implements both interfaces but does not access either constant, the compiler does nothing; there is no problem.
- if a constant name appears in the subclass, the compiler requires the interface's name (followed by a period character) to prefix that name. Hence there is no ambiguity.
- C# removed multiple inheritance from C++.

Why interface solves the problem: same method name


```
public interface A {
  public void m();
}

public interface B {
  public void m();
}
public class C implements A, B {
public void m() {
   System.out.println(“test");
  }
}
```

- Interface can’t specify the code body. Instead, the class that implements the interface supplies the code body.
- Hence there is no ambiguity for the same method name.

Whether the following multiple interface inheritance is correct?

```
public interface A {
    public int m2(int x);
}

public interface B {
    public String m2(int x);
}

public class C implements A, B {

    public int m2(int x){
       return x;
    }

    public String m2(int x){
        return new String(x+1);
    }
}
```

- an incorrect multiple interface inheritance.
- If it were allowed, c.m2() would have difficulty in selecting the implementation.
- This is also an example of *strong typing*: the compiler tries to find as many potential errors as possible.

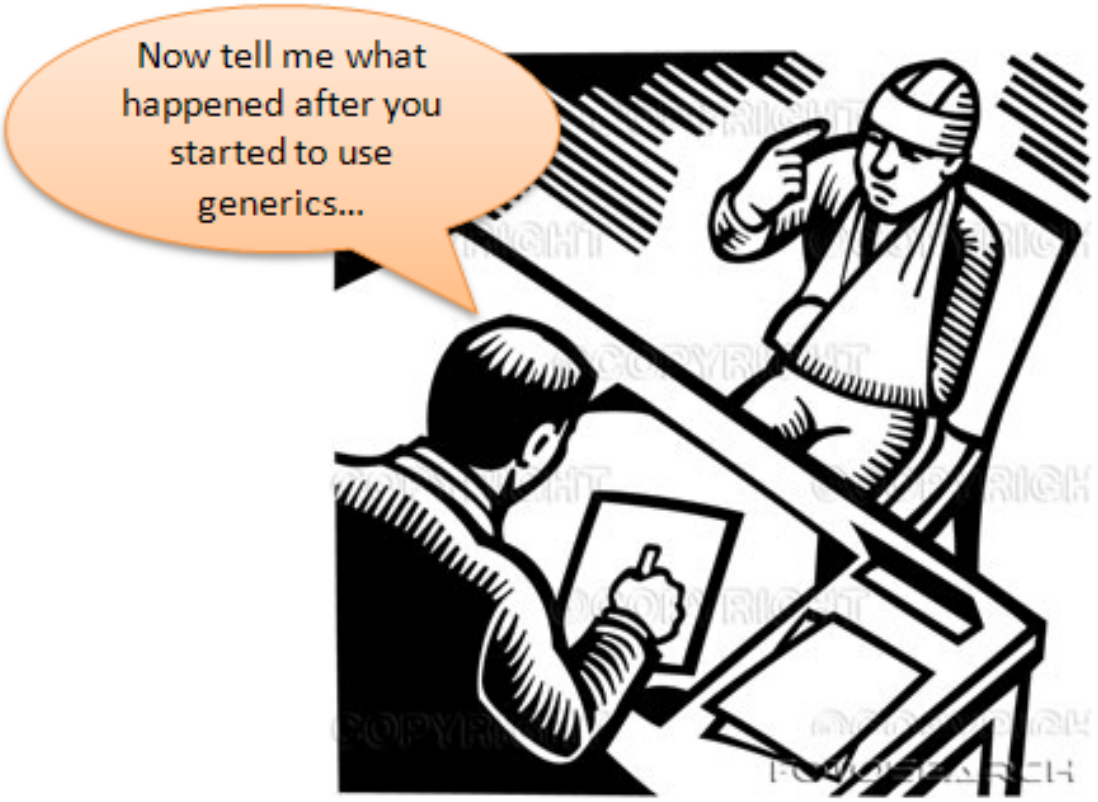
Review

- Polymorphism
  - Overloading
  - Coercion
  - Subtype
    - Polymorphism achieved by inheritance
    - Introduced dynamic binding
    - Why multiple inheritance not a good idea
    - More polymorphism by multiple (interface) inheritance
  - Parametric polymorphism

Questions?

Generics


Java
Generics
O'REILLY


Now tell me what happened after you started to use generics...

Generics

- What is generic programming (also called parametric polymorphism)
  - a function or datatype can be written *generically* so that it can deal equally well with any objects without depending on their types.
  - e.g., a function concat that joins two lists independent of element type

    concat: [T] × [T] → [T]

    List<T> concat (List<T> x, List<T> y)
  - The type of concat is parameterized by T for all values of T.
  - Variables can denote types.
    - A leap forward!
  - A *generic type* is parameterized by one or more formal type parameters

    Vector<T> // generic type
  - The actual type parameters are supplied when the generic type is instantiated
  - A *parameterized type* represents a set of types depending on the actual type parameter provided

    Vector <String>

Generics and programming languages

- Java 5 introduced generics
- Generics has a long history in programming theory and practice
  - CLU, Ada, Eiffel
  - C++ has similar concept called template
  - C# also has generics
  - Java 5 followed the idea in C#.
- Advantages:
  - Catch type errors at compile time.
  - Make code more readable.
- Can be tricky to use
  - One of the most controversial new features of Java 5.

Why generics: program w/o generics

1. Vector v = new Vector();
2. v.add("test");
3. String a = (String) v.get(0);
4. Integer b = (Integer) v.get(0);

- Problems with the above code:
  - Conceptually, we want to express a vector of something, such as a vector of String. However, the language does not support this expression. We can only say a vector of Objects. Consequently,
  - Error not caught by compiler: The above code will compile fine but will throw a runtime exception (java.lang.ClassCastException) when you execute it. This is obviously a serious problem and should be caught as early as possible.
  - Syntactically, it is cumbersome to cast types from time to time, such as in line 3) and 4)

Why generics: program with generics

- Now, let's rewrite the above code fragment using generics:

```
Vector <String> v = new Vector <String> ();
v.add("test");
String a = v.get(0);
Integer b = (Integer) v.get(0);
```

- That looks similar to the first fragment, except for the code in the angle brackets. The angle brackets are the syntax for providing type parameters to parameterized types.
- Compiling this program with J2SE 1.5 will give you an error.
- Errors are caught before runtime!

Generic types and parameterized types

- Generic type: defines a set of formal type parameters or type variables that must be provided for its invocation
  - Vector<T>
    - The (formal) type parameter is an unqualified identifier T
    - The type parameter T can be used as any other type in class, although it can't be used to construct new instances

      ```
      new T();   // incorrect
      ```

    - A generic type without its type parameter is called a raw type
      - Vector is the raw type of Vector<T>
- Parameterized type: a specific usage of a generic type where the formal type parameters are replaced by actual type parameters

  ```
  Vector<String>   //parameterized type
  ```

  - Primitive types are not allowed as type parameters

    ```
    Vector<int>;   //not allowed
    ```

Generic method

- Motivating example:

```
static Integer max(Integer x, Integer y) { return (x<y)?y:x; }
static Double max(Double x, Double y) { return (x<y)?y:x; }
```

- Our first try

```
static T max(T x, T y) { return (x<y)?y:x; }
```

There will be a problem when we invoke max("a", "b").

```
static String max(String x, String y) { return x.compareTo(y)<0)?y,x;}
```

- Use compareTo(..) method

```
static T max(T x, T y){
    return (x.compareTo(y) < 0 )?y:x;
}
```

- Generic method declaration: add type parameter before returning type

```
static <T> T max(T x, T y){
    return (x.compareTo(y) < 0 )?y:x;
}
```

- Not all the types have compareTo method

```
static <T extends Comparable<T>> T max(T x, T y){
    return (x.compareTo(y) < 0 )?y:x;
}
```

- The interface Comparable<T> contains a single method that can be used to compare one object to another:
  - interface Comparable<T> { public int compareTo(T o); }
  - **Interface Comparable<T>**
  - **All Known Subinterfaces:**
    - Delayed, Name, ScheduledFuture<V>
  - **All Known Implementing Classes:**
    - Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, Character, CharBuffer, Charset, CollationKey, CompositeName, CompoundName, Date, Date, Double, DoubleBuffer, ElementType, Enum, File, Float, FloatBuffer, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType, GregorianCalendar, IntBuffer, Integer, JTable.PrintMode, KeyRep.Type, LdapName, Long, LongBuffer, MappedByteBuffer, MemoryType, ObjectStreamField, Proxy.Type, Rdn, RetentionPolicy, RoundingMode, Short, ShortBuffer, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, String, Thread.State, Time, Timestamp, TimeUnit, URI, UUID

Invocation of generic methods

- Generic method can be called like an ordinary method, without any actual type parameter.
  - The type parameter is inferred from the type of the actual parameter

```
System.out.println(max(new Integer(21),new Integer(12)));
System.out.println(max("s1","s2"));

Mammal d1=new Dog(); d1.setAge(2);
Mammal d2=new Dog(); d2.setAge(1);
System.out.println(max(d1, d2).toString());
```

- Whether the following is correct?

  max("s1", new Integer(12));

```
static <T extends Comparable<T>> T max(T x, T y){
        return (x.compareTo(y) < 0 )?y:x;
}
```

Generic class


```
public class Pair <T, S> {
      private T first;
      private S second;

  public Pair(T f, S s) { first = f;  second = s;   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }
   public String toString(){
      return "("+first.toString()+", "+second.toString()+")";
   }

}
```

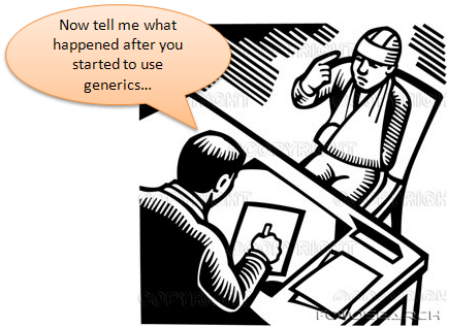
```
Pair<String, String> grade440=new Pair<String, String>("mike", "A");
Pair<String, Integer> marks440=new Pair<String, Integer>("mike", 100);
System.out.println("grade:"+grade440.toString());
System.out.println("marks:"+marks440.toString());
```

Wildcards—Why do we need to have them?


```
//Most common error in generics

public class SubtypeIterateMammls {
  static void iterateM(Vector<Mammal> ms) {
    for (Mammal m: ms)m.talk();
 }

public static void main (String[] a){
  Dog d1=new Dog();
  Dog d2=new Dog();
  Vector<Dog> dogs=new Vector<Dog>();
  dogs.add(d1);
  dogs.add(d2);
  iterateM(dogs);
}
}
```

- Suppose Dog is a subtype of Mammal
- Vector<Dog> is **not** a subtype of Vector<Mammal>
- In general, if T is a subtype of S, Vector<T> is not a subtype of Vector<S>
- This is the most counter-intuitive in generics.
- Need to change the argument type of iterateM to

  Vector<? extends Mammal>



Bounded types

```
public class SubtypeIterateMammls{
   static void iterateMammals(Vector < ? extends Mammal > ms) {
        for (Mammal m: ms)m.talk();
   }
   public static void main(String[] a){
      Dog d1=new Dog(); d1.setName("Pluto");
      Dog d2=new Dog(); d2.setName("Smart");

      Vector<Dog> dogs=new Vector<Dog>();
      dogs.add(d1);
      dogs.add(d2);

      iterateMammals(dogs);
  }
}

   static <T extends Comparable<T> > T max(T x, T y){
        return (x.compareTo(y) < 0 )?y:x;
   }
```

Type specifications:	
<? extends T>	any subtype of T
<? super T>	any supertype of T
<?>	any type

Why List<String> can not be subtype of List<Object>

```
List<String> ls = new ArrayList<String>(); // certainly legal
List<Object> lo = ls; //legal if list<String> is a subtype of List<Object>
lo.add(new Object()); // no problem!
String s = ls.get(0); // attempts to assign an Object to a String!
```

- Whether the code can compile correctly?
  - Answer: no. Because List<String> is not a subtype of List<Object>
- Substitution Principle:
  - a variable of a given type may be assigned a value of any subtype of that type,
    - E.g. Animal a =new Dog();
  - a method with a parameter of a given type may be invoked with an argument of any subtype of that type.
    - e.g., aVectorOfObject.add("aString");

- Type A is a subtype of B if A extends or implements B
- Some subtype examples
  - Integer is a subtype of Number
  - Double is a subtype of Number
  - ArrayList<E> is a subtype of List<E>
  - List<E> is a subtype of Collection<E>
  - Collection<E> is a subtype of Iterable<E>
- List<Integer> is NOT a subtype of List<Number>
- Subtyping properties
  - Subtype is transitive
    - A is a subtype of B, B is a subtype of C, A is a subtype of C
  - Substitution principle

- Substitution principle
  - a variable of a given type may be assigned a value of any subtype of that type,
    - List<Integer> ints = new ArrayList<Integer>();
  - a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

- If List<Number> were a supertype of List<Integer> …

```
List<Integer> ints = new ArrayList<Integer>();
ints. add(1);
List<Number> nums = ints;
        // this would be ok if List<Number> is a supertype of List<Integer>
nums. add(3. 14);
Integer x=ints.get(0);
x=ints.get(1);
```

Generic sorting

- Task: write an insertion sorting program

Implementation of Java generics

- The compiler ensures that the parameterized types is used correctly so that errors are caught at compile time;
- No generic information is available at runtime.
- For example, List<Integer> will be converted to the non-generic type List, which can contain arbitrary objects.
- The compile-time check guarantees that the resulting code is type-correct.

Polymorphism

- Overloading
- Coercion
- Inclusion/subtype polymorphism
- Parametric polymorphism (generics)

A puzzler

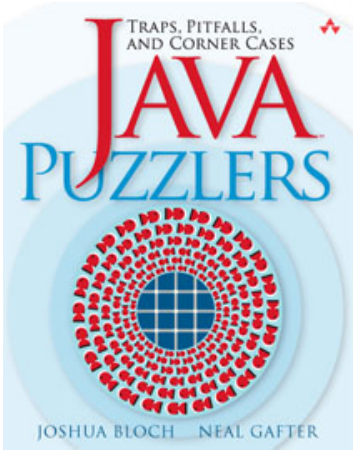
```
import java.util.*;
public class ShortSet  {
    public static void main(String[] args) {
        Set <Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
                s.add(i);
                s.remove(i - 1) ;
                                    //int-value expression
        }
        System.out.println(" Size: " + s.size());
    }
}
```

What is the output?

It compiles and prints a result

Short/short: 16 bit integer

Size: 100



What is the type of i-1?

i: short

1: int

by type coercion, we have int

By autoboxing, we have Integer

Why does it compile correctly?

HashSet <E>

add(E e)

remove(Object o)