Génie Logiciel et Gestion de Projets

Object-Oriented Programming
An introduction to Java

Roadmap

- History of Abstraction Mechanisms
- Learning an OOPL
- Classes, Methods and Messages
- Inheritance
- Polymorphism

History of Abstraction Mechanisms

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Abstraction

- Abstraction is the purposeful suppression, or hiding of some details of a process or artifact, in order to bring out more clearly other aspects, details and structure.

History of Abstraction

The function centered view	Functions and Procedures
The data centered view	Modules Abstract Data Types
The service centered view	Object-Oriented Programming

Abstraction Mechanisms in OOP languages

- Classes are as ADTs in a service-centered view
- Message Passing and Method binding bring polymorphism leading to more readable code
- Class Hierarchies and Inheritance bring code sharing resulting in increased functionality and reduction of code size.
- Inheritance and Polymorphism together allow for tailoring shared code.

Parnas Principle

- For modules:
  - One must provide the intended user with all the information needed to use the module correctly and nothing more
  - One must provide the implementor with all the information needed to complete the module and nothing more
- For objects:
  - A class definition must provide the intended user with all the information necessary to manipulate an instance of a class correctly and nothing more.
  - A method must be provided with all the information necessary to carry out its given responsibilities and nothing more.

Learning an OOPL

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

- Class definition
  - is like a type definition
  - specifies the data and the behavior (methods) to be associated with this data,
  - does not create a value (object);
  - default values for data can be specified;
  - visibility of data and behavior can be specified
- Object instantiation
  - the creation of a new instance of a class, i.e. a value (object)
  - initial values for the data fields can be passed

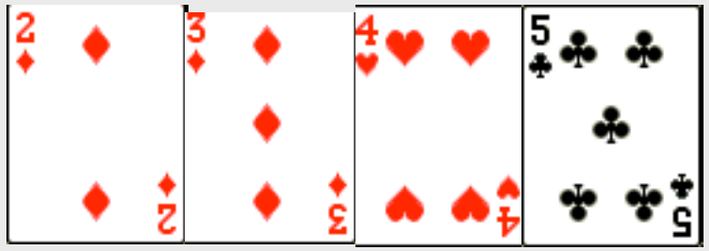
- Messages
  - have a receiver (the object the message is sent to),
  - a message selector (some text indicating the message being sent), and arguments.
- Class hierarchies
  - are defined when in a class definition one or more parent classes can be specified (single <> multiple inheritance)
- Inheritance
  - is the property that instances of a child class or subclass can access both data and behavior (methods) associated with a parent class or superclass
- Method binding or method look-up
  - is a mechanism for determining the method to be used in response to the message being sent

Classes, Methods and Messages

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Card Example

- A card belongs to a suit, has a rank and a color.
- A card can be face up or face down on the table (screen) and can be flipped from one position to the other
- A card can be displayed upon and erased from the screen
- A magic card is a special kind of card, its rank can be changed



Class Definition

Define Class
<name>
<data definitions>
<method declarations and definitions>

Card
suit: {diamond, club, heart, spade} rank : integer faceUp : boolean color : {red, black}
Flip() <some code> Draw(W : Window, X : Position, Y : Position) <some code> Erase() <some code>

define class:
Card
....

Java (version 5)

- Java is not a dialect of C++, there are superficial surface similarities but the underlying differences are substantial.
- Java does not have pointer, reference, structure, and union types, goto statement, function definition or operator overloading.
- Java uses garbage collection for memory management, introduces the concept of interfaces, introduces packages and package visibility, supports multithreading and exception handling.
- Java is a portable language due tot he use of a virtual machine.

Java Card class

```
class Card {
  final static public int red = 0;     // static values
  final static public int black = 1;
  final static public int spade = 0;
  final static public int heart = 1;
  final static public int diamond = 2;
  final static public int club = 3;

 private boolean faceUp;          // data fields
 private int rankValue;
 private int suitValue;

  public Card (int sv, int rv){    // constructor
   suitValue = sv;  rankValue = rv; faceUp = true;
  }
```

Java Card Class (cont'd)

```
public  boolean isFaceUp(){
  return faceUp; }          // access attributes

public  int  rank(){
 return rankValue; };

public  int  suit(){
 return suitValue; };

 public  int color(){
  if (suit() == heart || suit() == diamond)
      then return red;
     else return black;}

 public  void flip() { faceUp = ! faceUp; } //actions
 public  void draw( Graphics g, intx, int y)
…};
}
```

Visibility

- Visibility modifiers to control visibility thus manipulation of data fields and methods
- Public features can be seen and manipulated by anybody (external/interface/service view)

```
public void flip(){
   faceUp = !faceUp;}
```

- Private features can be manipulated only within class (internal/implementation view)

```
private int suitValue;
```

Constructor

- A constructor is a function that is implicitly invoked when a new object is created.
- In Java a constructor is a function with the same name as the class.

```
public Card (int sv, int rv){ // constructor
  suitValue = sv;
  rankValue = rv;
  faceUp = true;}
```

Getter and Setters

- Getter and setter methods (or accessors and mutators) to control how data is accessed and modified.

```
public  boolean isFaceUp(){
   return faceUp;}

public  int  rank(){
  return rankValue;}
```

Class Data Fields

- Class data fields that are shared in common amongst all instances

```
public class CountingClass {
  private static int count; //shared by all

  CountingClass(){
    count = count + 1;
  }

 static {count = 0;}

 public int getCount(){
  return count;
 }
```

Constant Data Fields

- Constant or immutable data fields to guarantee no changes will occur.
- Notice how symbolic constants are defined in Java:

```
final static public int heart = 1;
final static public int diamond = 2;
final static public int club = 3;
```

- static means that all instances share the same value, one per class.
- final means it will not be reassigned.

Class Methods

- Static methods which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in `ClassName.methodName(args)`
- You can also refer to static methods with an object reference like: `instanceName.methodName(args)` but this is discouraged because it does not make it clear that they are class methods.
- Class methods cannot access instance variables or instance methods directly, they must use an object reference. Also, class methods cannot use the **this** keyword as there is no instance for **this** to refer to.

Enumerations

- new in Java 5
- enum declaration defines a full-fledged class
  - possibility to add methods and fields to enum type,
  - implement interfaces.

```
public enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}

public class Card{
private final Rank rank;
private final Suit suit;...}
```

```
public enum Planet {

MERCURY (3.303e+23, 2.4397e6),
 VENUS   (4.869e+24, 6.0518e6),
 EARTH   (5.976e+24, 6.37814e6),
 MARS    (6.421e+23, 3.3972e6),
 JUPITER (1.9e+27,   7.1492e7),
 SATURN  (5.688e+26, 6.0268e7),
 URANUS  (8.686e+25, 2.5559e7),
 NEPTUNE (1.024e+26, 2.4746e7),
 PLUTO   (1.27e+22,  1.137e6);

 private final double mass;   // in kilograms
 private final double radius; // in meters
 Planet(double mass, double radius) {
     this.mass = mass;
     this.radius = radius;
 }
 public double mass()   { return mass; }
 public double radius() { return radius; }

 // universal gravitational constant  (m^3 kg^-1 s^-2)
 public static final double G = 6.67300E-11;

 public double surfaceGravity() {
     return G * mass / (radius * radius);
 }
 public double surfaceWeight(double otherMass) {
     return otherMass * surfaceGravity();
 }
}
```

Interfaces

- An interface is like a class, but it provides no implementation. Later, another class can declare that it supports the interface, and it must then give an implementation.

```
public interface Storing {
      void writeOut (Stream s);
      void readFrom (Stream s);
};

public class BitImage implements Storing {
      void writeOut (Stream s) {
        // ...
      }
      void readFrom (Stream s) {
// ...}
};
```

Java Classes and Methods

- There is no preprocessor or global variables. Symbolic values are created by declaring and initialising a data field with the keywords final and static.
- The keyword static used on data fields indicates that only one copy of the data field exists which is shared by all instances.
- The keyword static on a method indicates that this method exists even when no instance of the class exist.
- A Java program needs one class with a main static method.
- Method implementations must be provided directly in the class definition.
- The public and private keywords are applied individually to every instance variable and method definition.

Instantiation and Initialization

"make-instance" <class> <initial values>

"make-instance" Card Diamond, 9, false

"make-instance" Card Spade, 4, true

(aCard)
suit: Diamond
rank: 9
faceUp: false
color: red

(aCard)
suit: Spade
rank: 4
faceUp: true
color: black

Creation and Initialisation in Java

- Object values are created with the operator **new**; space is allocated on the heap, the pointers are always implicit.
- The syntax of the new operator requires the use of parentheses even if arguments are not necessary
- All variables of some object type are initially assigned the value **null.**
- Constructors in Java can call other constructors on the same class by using the keyword **this**; it allows to factor out common behavior

```
PlayingCard aCard; // simply names a new variable
// next expression creates the new object
aCard = new PlayingCard(Diamond, 3);
```

Message Passing

- Message passing is the dynamic process of asking an object to perform an action
- Messages have a receiver (the object the message is sent to), a message selector (some text indicating the message being sent), and arguments.

```
card mycard = new card(hearts, 9)
mycard.draw(win1,10,10)
if(mycard.faceUp?()).
```

Message Passing in Java

- Java is statically typed: requires the programmer to declare a type for each variable. The validity of a message passing expression will be checked at compile time, based on the declared type of the receiver.
- Inside a method, the receiver can be accessed by means of a pseudo-variable, called **this**.
- A message expression or data access within a method with no explicit receiver is implicitly assumed to refer to **this**.

Message Passing in Java

```
class PlayingCard {
...
public void flip () { setFaceUp( ! faceUp ); }
...
}
```

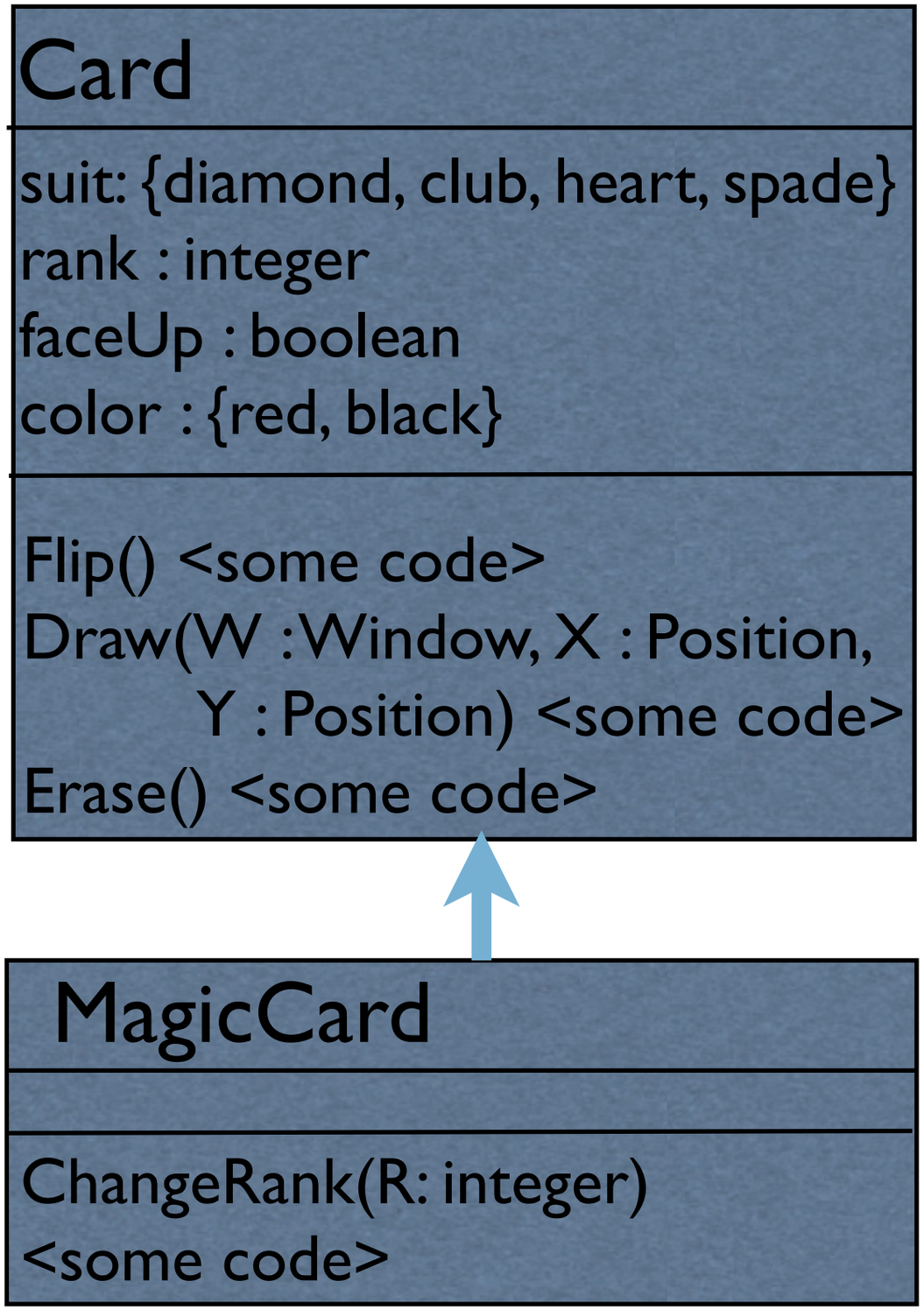
```
class PlayingCard {
...
public void flip () { this.setFaceUp( !
this.faceUp); }
...
}
```

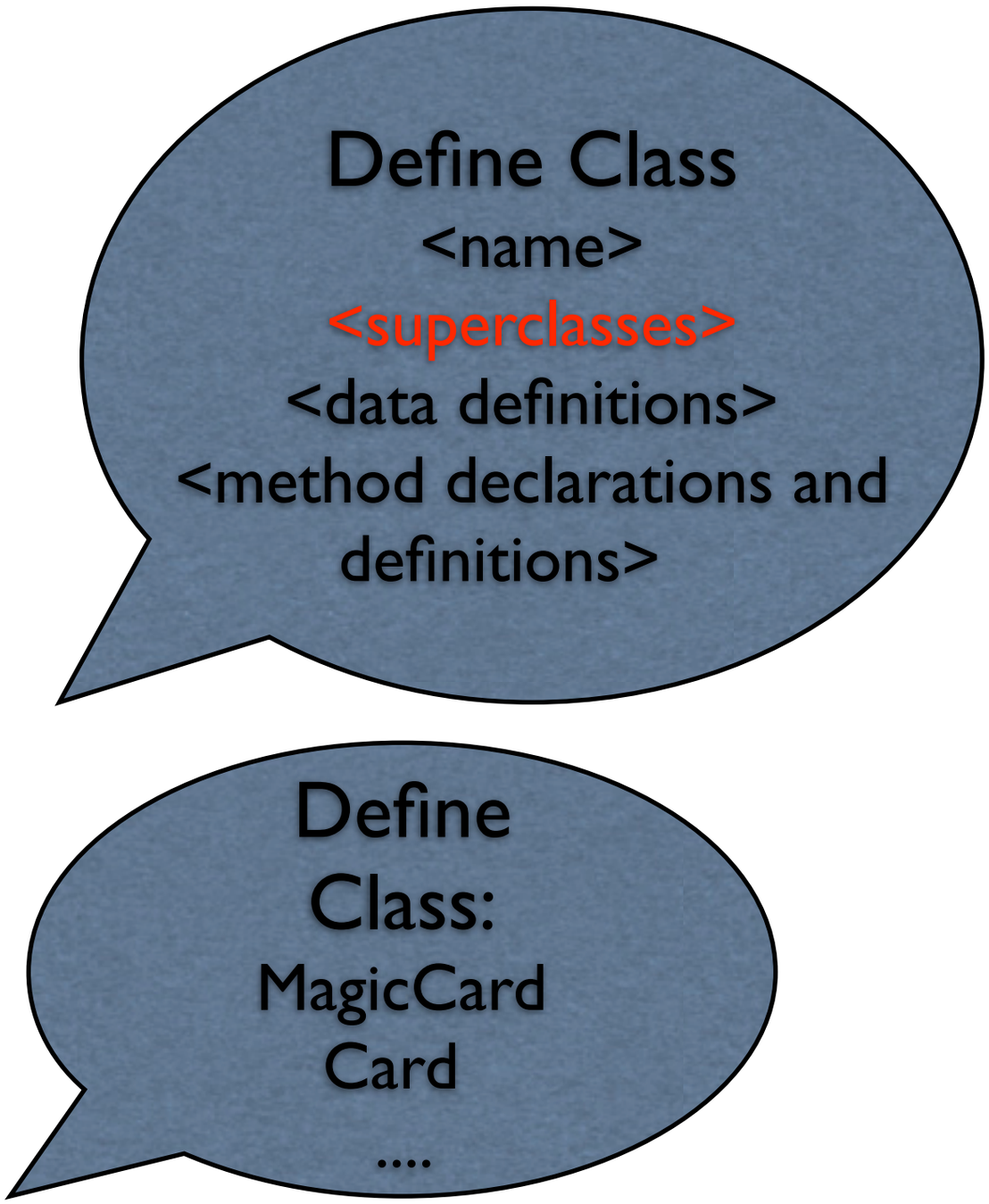
Inheritance

Roadmap

- Meanings of Inheritance
- Syntax to describe Inheritance
- Various forms of Inheritance
- Java modifiers and inheritance
- Java constructors and inheritance

Class Hierarchies





Practical Meaning

- Data members defined in the parent are part of the child.
- Behaviour defined in the parent is part of the child.

- The child is an *extension* of the parent:
  The behaviour of a child class is strictly larger than the behaviour of the parent.
- The child is a *contraction* of the parent
  The child class can override behaviour to make it fit a specialized situation.

Java Syntax

- Subclasses are declared using the keyword **extends**

```
class TablePile extends CardPile { ... };
```

- All classes are derived from a single root class Object; if no superclass is mentioned, Object is assumed

```
class CardPile extends Object { ... };
class CardPile { ... };
```

Substitutability

- The type given in a declaration of a variable may not match the type associated with a value the variable is holding.
- *Substitutability through interfaces*

```
class CannonWorld extends Frame{
...
private class FireButtonListener implements
ActionListener{
   public void actionPerformed(ActionEvent e){
...}
}

public CanonWorld(){
...
fire.AddActionListener(new FireButtonListener());}}
```

Subtype and subclass

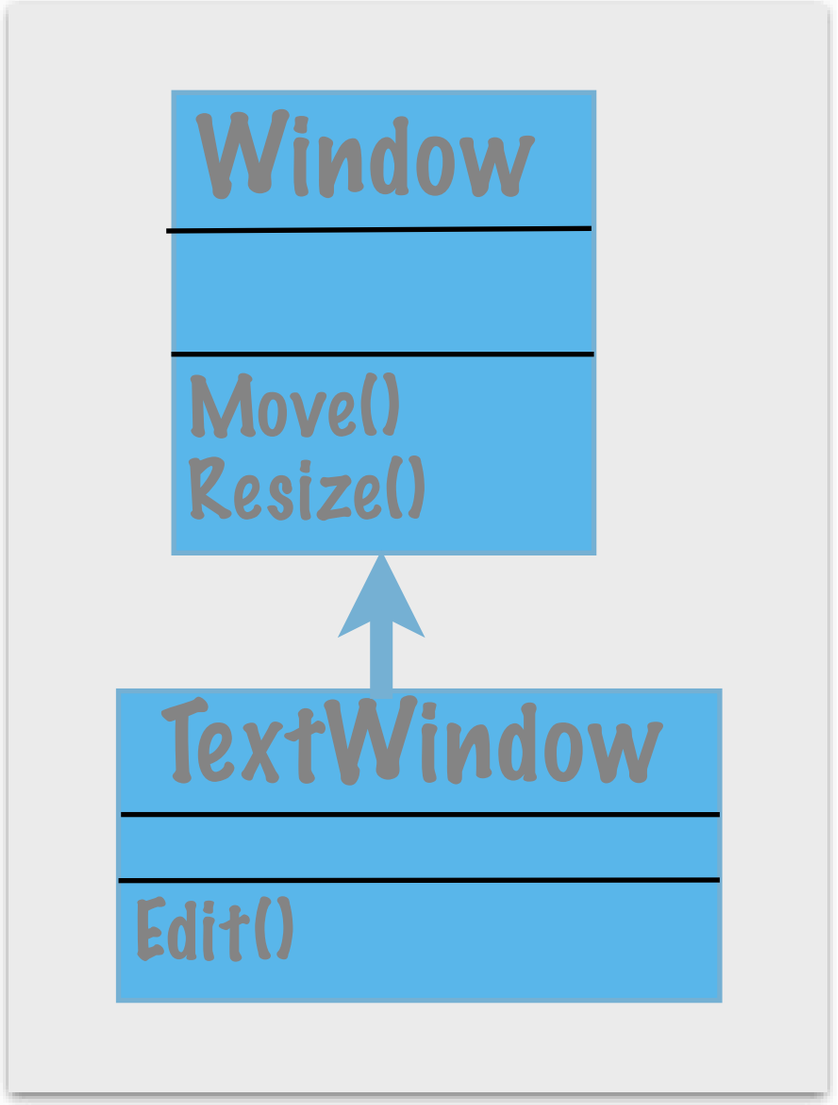
- A type B is considered to be a subtype of A if
  - an instance of B can legally be assigned to a variable declared as type A.
  - this value can then be used by the variable with no observable change in behavior.
- Subclass merely refers to the mechanism of constructing a new class using inheritance.
- It is not possible in Java to distinguish both concepts.

Specialised forms of Inheritance

- Specialization
- Specification
- Construction
- Generalisation or Extension
- Limitation
- Combination

Specialization

- The child class is a specialised form of the parent class but satisfies the specifications of the parent class completely.
- It creates a subtype.
- Is an ideal form of inheritance, good designs should strive for it.



Specification

- special case of inheritance for specialization
- guarantees that classes maintain a certain common interface
- implementation is deferred to child classes
- Java supports this idea through
  - interfaces
  - abstract classes and methods

```
public abstract class Number{

 public abstract int intValue();
 public abstract long
    longValue();
 public abstract float
   floatValue();
 public abstract double
   doubleValue();
 public byte byteValue(){
    return (byte) intValue();}
 public short shortValue(){
   return (short intValue();}
}
```

Construction

- the child class gets most of its desired functionality from the parent class only changing names of methods or modifying arguments
- forms subclasses that are not subtypes
- is a fast and easy route to new data abstractions

Example in Java Library:

```
class Stack extends Vector{
  public Object push(Object item)
   {addElement(item); return item;}

  public boolean emtpy()
   {return isEmpty();}

  public synchronized Object pop()
   {Object obj = peek();
    removeElementAt(size() - 1);
    return obj;}

  public synchronized Object peek()
   {return elementAt(size() - 1);}
}
```

Extension

- child class only adds behaviour to the parent class and does not modify or alter any of the inherited attributes
- subclasses are always subtypes

Example in Java Library:

```
class Properties extends Hashtable{
  ...
  public synchronized void
   load(InputStream in) throws
   IOException{...}

  public synchronized void
   save(OutputStream out, String
     header{...}

  public String
   getProperty(String key) {...}

  public Enumeration
   propertyNames(){...}
}
```

Limitation

- The behaviour of the subclass is smaller or more restrictive than the behaviour of the parent class.
- forms subclasses that are not subtypes
- occurs most frequently when building on a base of existing classes that should not, or cannot, be modified.

Suppose(!) Set class implemented like Vector class:

```
class Set extends Vector{
  //methods addElement,
  // removeElement, contains,
  //isEmpty and size are all
  //inherited from Vector

public int indexOf(Object obj)
 {throw new
  IllegalOperation("indexOf");
 }

public int elementAt(int
                        index)
 {throw new
IllegalOperation("elementAt");
 }
}
```

Combination

- multiple inheritance
  ability to inherit from two or more parent classes
- no multiple inheritance in Java
- approximations are possible
  - e.g. new class extends an existing class and implements an interface
  - e.g. new class implements multiple interfaces. Java library: RandomAccessFile implements both the DataInput and DataOutput protocols

Java Visibility Modifiers

- A **private** feature can be accessed only within the class definition (but memory is still found in the child class, just not accessible).
- A **public** feature can be accessed outside the class definition.
- A **protected** feature can be accessed only within the class definition in which it appears or within the definition of child classes.

Java Modifiers and Inheritance

- Static data fields and methods are inherited but static methods cannot be overriden.
- Methods and classes can be declared abstract. Abstract classes cannot be instantiated. An abstract method must be overridden in subclasses.
- The modifier final used with a class indicates that the class cannot be subclassed; the modifier final used with a method indicates that the method can not be overriden.

Inheritance and Constructors

- The constructor of a child class will always invoke the constructor for the parent class.
  This invocation always takes place before the code of the constructor is executed.
- If the constructor for the parent class needs arguments, the pseudovariable **super** is used as if it were a function. If no call on **super** is made explicitly, the default constructor (the one with no arguments) is used.

```
class DeckPile extends CardPile {
  DeckPile (int x, int y, int c) {
    super(x,y);   //initialise parent
    ...       //initialise child }...}
};
```

Method Binding

- In a statically typed language we say the class of the declaration is the static class for the variable, while the class of the value it currently holds is the dynamic class.
- Should the binding for information be associated with the static class of a variable or the dynamic class??

Method Binding in Java

- Messages are always bound to methods based on the dynamic type of the receiver.
- Data fields can also be overridden in Java but in this case binding of an access is based on the static type
- Interfaces define a hierarchical organisation similar but independent from the class hierarchy. Interfaces can be used as types in variable declarations. Instances of classes that implement that interface can be assigned to these variables. The static type is then an interface type while the dynamic type is a class type. Method binding uses the dynamic type.

this and **super** sends

- Methods use:
  - **this** to start lookup in class of the receiver object.
  - **super** to start lookup in their implementor's parent

! **this** = dynamically (late) bound

super = statically bound !

this sends

- bound to the current object
- sends to **this** start the method lookup at the current object (receiver) again
- inherently dynamic
- cannot determine from reading the source code to exactly which object it will be sent to!

super sends

- start lookup in the superclass of the class method resides in
  - inherently static
    - at compilation time it is exactly known where to find the method
    - the receiver object does not change
    - the mechanism for referring to overridden methods

Polymorphism

Polymorphism in OOPL

- Polymorphic Variable
  variable that can hold different types of values during execution
- Overloading
  one name referring to two or more implementations
- Overriding
  child class redefining a method inherited from the parent class
- Templates
  Creating general tools or classes by parametrizing on types

Overloading on scope

- Overloading can be based on scope: a function f defined inside a function g is distinct from a function f defined inside a function h.
- This type of overloading is resolved by looking at the type of the receiver.
- Allows the same name to be used in unrelated classes.

Overloading on Type Signatures

- A different type of overloading allows multiple implementations in the same scope to be resolved using type signatures. A type signature is the combination of argument types and return type.

```
class Example {

 //same name, three different methods
  int sum (int a) { return a; }
  int sum (int a, int b) { return a + b; }
  int sum (int a, int b, int c) { return a + b + c; }
  }
```

- Resolution is almost always performed at compile time, based on static types, and not dynamic values.

Overloading: Redefinition

- A redefinition occurs when a child class changes the type signature of a method in the parent class.
- In Java the so-called merge model is used to resolve the name: the scope of the child is merged with the scope of the parent.

```
class Parent {
 public void example (int a)
 { System.out.println("in parent method"); }
}

class Child extends Parent {
 public void example (int a, int b)
 { System.out.println("in child method"); }
}
Child aChild = new Child(); aChild.example(3);
```

Overriding

- Overriding only occurs in the context of the parent/child relationship.
- A method in a child class overrides a method in the parent class if it has the same name and type signature. The type signatures must match.
- Overridden methods are sometimes combined together.
- Overriding is resolved at run-time, not at compile time.
- In Java overriding occurs automatically when a child class redefines a method with the same name and type signature.

Shadowing

- Java allows instance variables to be redefined, and uses shadowing. Shadowing can be resolved at compile time, does not require any run-time search.

```
class Parent {
 public int x = 12;
}
class Child extend Parent {
  public int x = 42; // shadows variable from parent
 }                                class
Parent p = new Parent();
System.out.println(p.x);
>>12
Child c = new Child();
System.out.println(c.x);
>>42
p = c; // be careful here!
System.out.println(p.x);
>>12
```

Polymorphic Variable

- A polymorphic variable is a variable that can hold values of different types during the course of execution.
  - Simple polymorphic variables
  - The receiver variable **this** holds the actual value during execution, not the static class.

Polymorphic Variable

- Pure polymorphism occurs when a polymorphic variable is used as an argument.

```
class StringBuffer {
String append(Object value){
   return append(value.toString()); } ...
}
```

- Reverse polymorphism or downcasting undo the assignment to a polymorphic variable, i.e., to determine the variables true dynamic value.

```
Parent aVariable = ...;
Child aCard;
if (aVariable instanceof Child)
  aChild = (Child) aVariable;
```

Generics

- The basic idea is to develop code by leave certain key types unspecified, to be filled in later.

```
public interface LinkedList<E>{
  void add(E x);
  Iterator<E>iterator();}
```

- Java Collections
  A collection with a generic type has a type parameter that specifies the element type to be stored in the collection

```
LinkedList list = new LinkedList();
list.add(new Integer(1));
Integer num = (Integer) list.get(0);

LinkedList<Integer> list = new LinkedList<Integer>();
list.add(new Integer(1));
Integer num = list.get(0);
```

Generic Methods

- Static methods, non-static methods, and constructors can all be parameterized in almost the same way as for classes and interfaces.

```
void printCollection(Collection<?> c){
for(Object o:c) {System.out.println(o);}
}
```

- There are three types of wildcards:
  - `? extends Type`: denotes a family of subtypes of type `Type`.
  - `? super Type`: denotes a family of supertypes of type `Type`

```
public static <T extends Comparable<? super T>>
        void sort(List<T> list){ ... }
```

  - `?`: denotes the set of all types or any

Generics and Inheritance

- Consider the following code snippet:

```
List<Driver> ld = new ArrayList<Driver>(); //1
List<Person> lp = ld; //2
```

  Is it legal assuming that Driver is a subtype of Person?

- `lp.add(new Person()); // 3`
  `Driver d = ld.get(0); // 4: attempts to assign a Person to a Driver!`

- In general, if `Foo` is a subtype (subclass or subinterface) of `Bar`, and `G` is some generic type declaration, it is not the case that `G<Foo>` is a subtype of `G<Bar>`.

References

- Thimothy Budd. An Introduction to Object-Oriented Programming, third edition. Addison-Wesley. 2002.
- Thimoty Budd. Understanding Inheritance, Chapter 8 in Understanding Object-Oriented Programming in Java (updated version). Addison-Wesley. http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/slides/chap08/under.pdf
- Gilad Bracha. Generics in the Java Programming Language. 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- M. Naftalin and P. Wadler. Java Generics and Collections. O'Reilly Media, Inc., Oct. 2006. https://java-generics-book.dev.java.net/