Inheritance

Object Oriented Programming

Version 4.6.2 - April 2016

Inheritance

- A class can be a sub-type of another class
- The derived class contains
  - all the members of the class it inherits from
  - plus any member it defines explicitly
- The derived class can override the definition of existing methods by providing its own implementation
- The code of the derived class consists of the changes and additions to the base class

Addition

```
class Employee{
   String name;
   double wage;
   void incrementWage(){…}
}
```

```
class Manager extends Employee{
   String managedUnit;
   void changeUnit(){…}
}
```

```
Manager m = new Manager();
m.incrementWage(); // OK, inherited
```

Override

```
class Vector{
   int vect[];
   void add(int x) {…}
}
```

```
class OrderedVector extends Vector{
   void add(int x){…}
}
```

Inheritance and polymorphism

```
class Employee{
 private String name;
 public void print(){
   System.out.println(name);
 }
}
class Manager extends Employee{
   private String managedUnit;

   public void print(){ //override
      System.out.println(name); //un-optimized!
      System.out.println(managedUnit);
   }
}
```

```
Employee e1 = new Employee();
Employee e2 = new Manager();
e1.print(); // name
e2.print(); // name and unit
```

Inheritance and polymorphism

```
Employee e1 = new Employee();
Employee e2 = new Manager(); //ok, is_a
e1.print(); // name
e2.print(); // name and unit
```

Why inheritance

- Frequently, a class is merely a modification of another class. Inheritance minimizes the repetition of the same code
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding a new functionality in the base class automatically adds it in the subclasses too
  - Less chances of different (and inconsistent) implementations of the same operation

Why inheritance

- Often we need to treat in a similar way different objects,
  - Polymorphism allow feeding algorithms with different objects
  - Dynamic binding allow accomodating different behavior behind the same interface

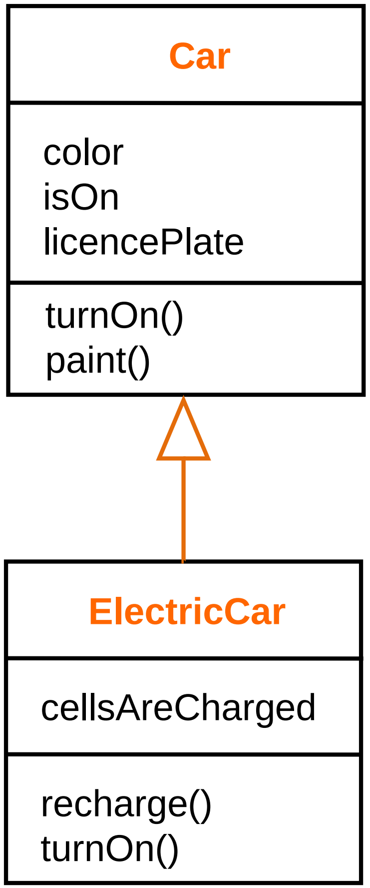
Terminology

- Class one above
  - Parent class
- Class one below
  - Child class
- Class one or more above
  - Superclass, Ancestor class, Base class
- Class one or more below
  - Subclass, Descendent class

Inheritance in a few words

- Subclass
  - Inherits attributes and methods defined in base classes
  - Can modify inherited attributes and methods (*override*)
  - Can add new attributes and methods

Inheritance syntax: **extends**




```
class Car {

  String color;
  boolean isOn;
  String licencePlate;

  void paint(String color) {
    this.color = color;
  }

  void turnOn() {
    isOn=true;
  }
}
```

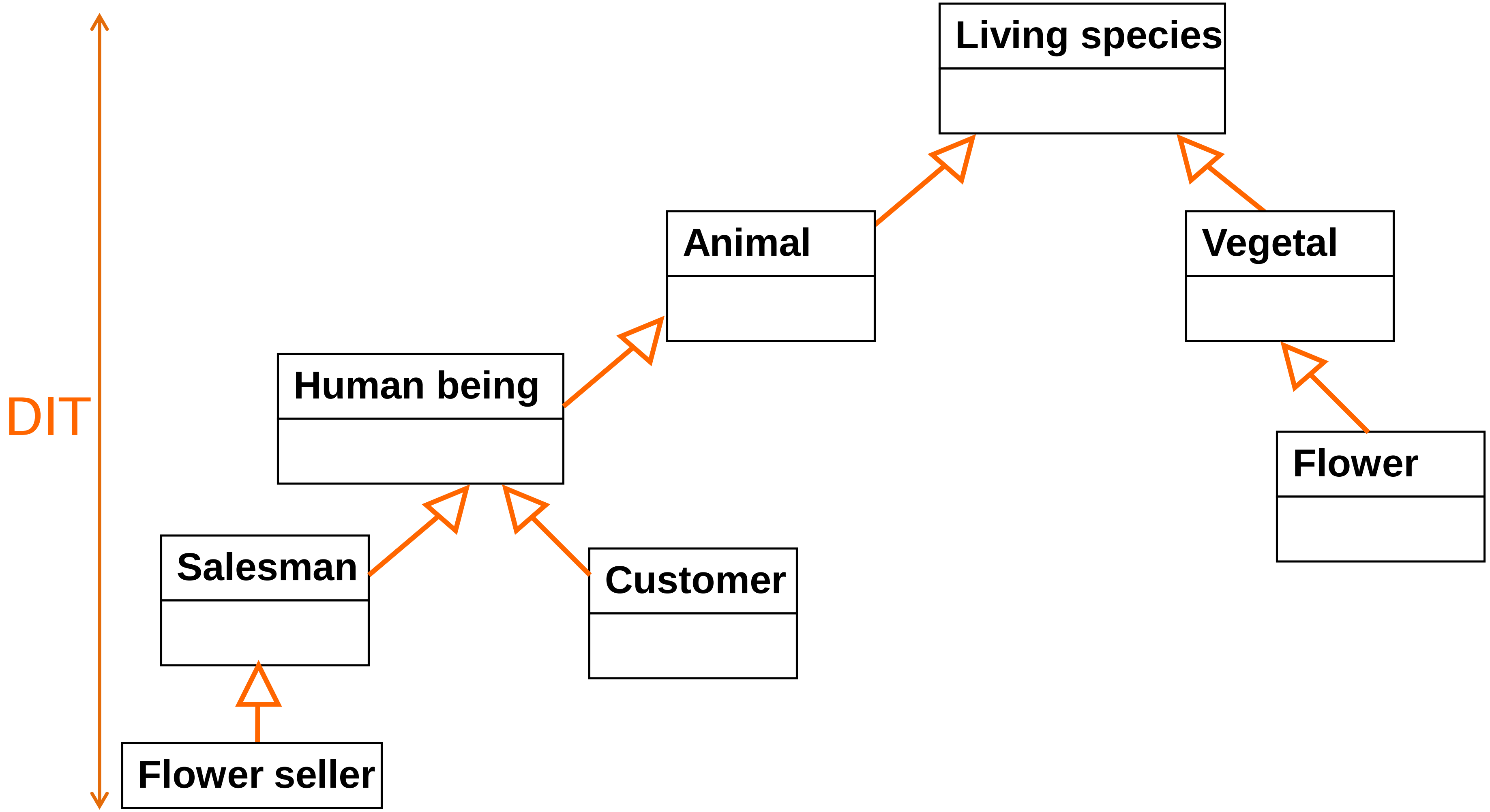
```
class ElectricCar extends Car{

  boolean cellsAreCharged;

  void recharge() {
    cellsAreCharged = true;
  }

  void turnOn() {
    if(cellsAreCharged )
      isOn=true;
  }
}
```

Inheritance tree



Depth of Inheritance Tree

- Too deep inheritance trees put at risk the understandability of the code
- DIT $\leq$ 5
  - Empirical limit

CASTING

Types

- Java is a strictly typed language, i.e., each variable has a type

```
float f;
f = 4.7;    // legal
f = "string";// illegal
Car c;
c = new Car(); // legal
c = new String();// illegal
```

Cast

- Type conversion
  - explicit or implicit

```
int i = 44;

float f = i;
// implicit cast 2c -> fp

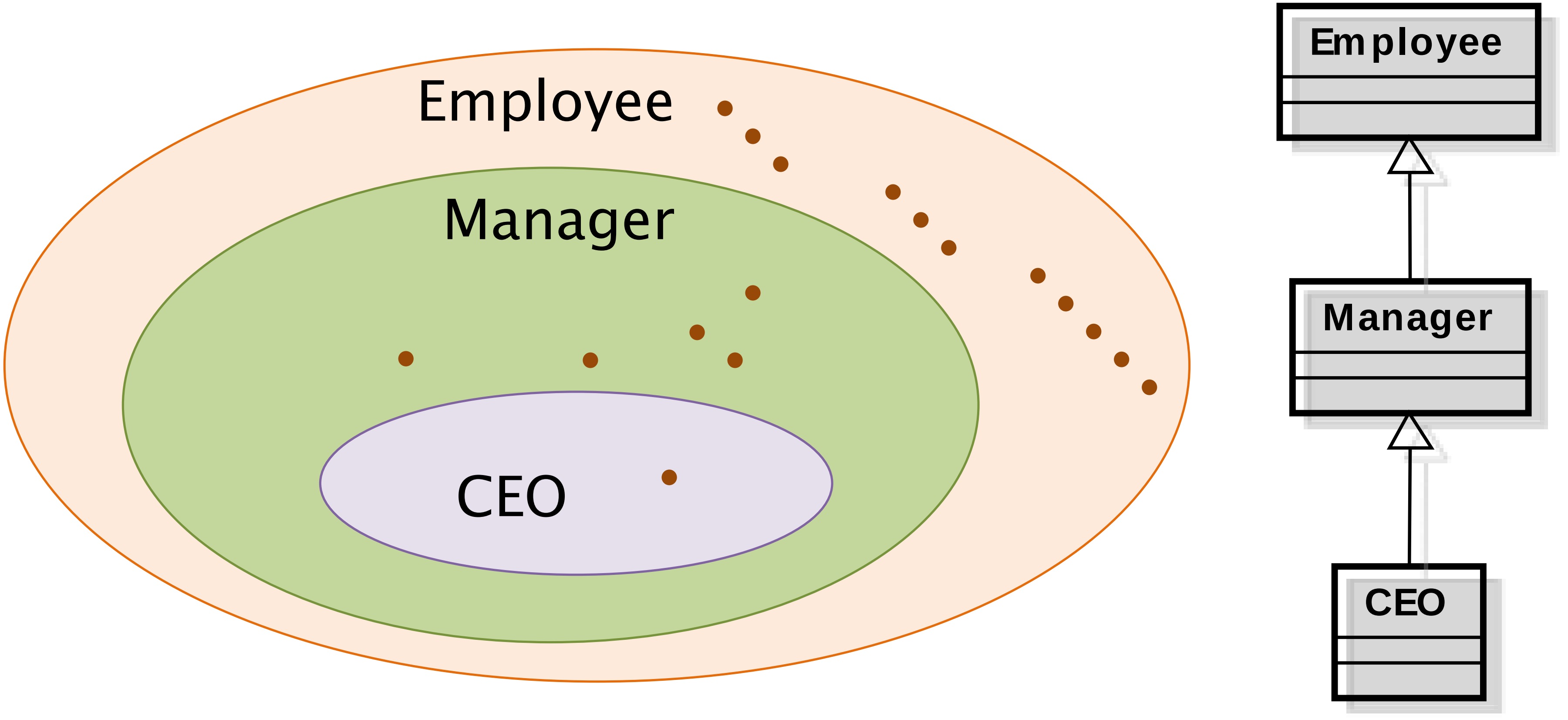
f = (float) 44;
// explicit cast
```

Cast – Generalization

- Things change slightly with inheritance
- Normal case…


```
Employee e = new Employee("Smith",12000);
Manager m = new Manager("Black",25000,"IT");
```

Generalization



Upcast

- Assignment from a more specific type (subtype) to a more general type (supertype)
  - ```
    Employee e = new Employee(…);
    Manager m = new Manager(…);
    Employee em = m
    ```
  - $\forall$ m $\in$ Manager : m $\in$ Employee
- Upcasts are always type-safe and are performed implicitly by the compiler
  - Though it is legal to explicitly indicate the cast
    ```

Upcast

- Motivation
  - You can treat indifferently object of different classes, provided they inherit from a common class

```
Employee[] team = {
  new Manager("Mary Black",25000,"IT"),
  new Employee("John Smith",12000),
  new Employee("Jane Doe",12000)
};
```

Cast

- Reference type and object type are distinct concepts
- A reference cast only affects the reference
  - In the previous example the object referenced to by '**`em`**' continues to be of Manager type
- Notably, in contrast, a primitive type cast involves a value conversion

Downcast

- Assignment from a more general type (super-type) to a more specific type (sub-type)
  - `Manager mm = (Manager)em;`
  - $\exists$ em $\in$ Employee : em $\in$ Manager
  - $\exists$ em $\in$ Employee : em $\notin$ Manager
- Not safe by default, no automatic conversion provided by the compiler
  - **MUST** be explicit

Downcast

- Motivation
  - To access a member defined in a class you need a reference of that class type
    - Or any subclass

Syntax Error: The method getDepartment() is undefined for the type Employee

```
Employee emp = staff[0];
s = emp.getDepartment();
Manager mgr = (Manager)staff[0];
s = mgr.getDepartment();
```

Downcast – Warning

- The compiler trusts any downcast
- The JVM at run-time checks type consistency for all reference assignments

```
mgr = (Manager)staff[1];
```

`ClassCastException`:
Employee cannot be cast to Manager

Down cast safety

- Use the **`instanceof`** operator

  `aReference instanceof aClass`

  - Returns true if the object referred to by the reference can be cast to the class
    - i.e. if the object belongs to the given glass or any of its subclasses

```
if(staff[1] instanceof Manager){
   mgr = (Manager)staff[1];
}
```

POLYMORPHISM AND DYNAMIC BINDING

Polymorphism

- A reference of type T can point to an object of type S if-and-only-if
  - S is T or
  - S is a subclass of T

```
Car myCar;
myCar = new Car();
myCar = new ElectricCar();
```

Polymorphism

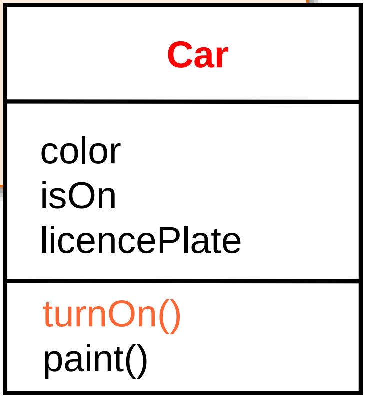
```
Car[] garage = new Car[4];
garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] = new ElectricCar();
garage[3] = new Car();
for(Car a : garage){
   a.turnOn();
}
```

Static type checking

- The compiler performs a check on method invocation on the basis of the reference type

```
for(Car a : garage){
   a.turnOn();
}
```

Does the type of **a** (i.e. **Car**) provide method **turnOn()**?

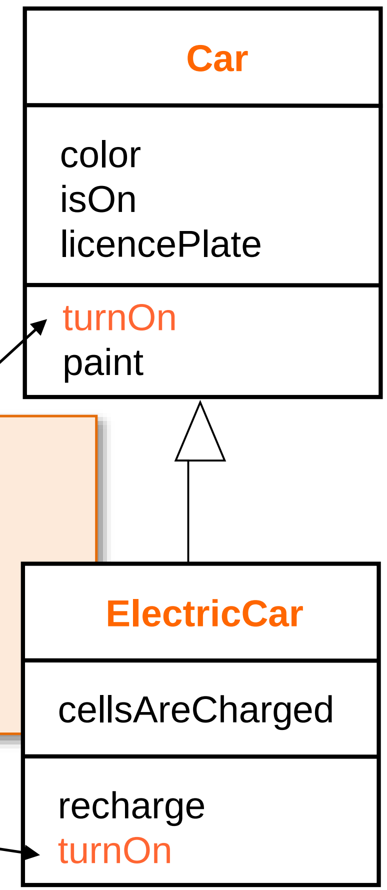


Dynamic Binding

- Association message - method
  - Performed by JVM at run-time
- Constraint
  - Same signature

```
for(Car a : garage){
   a.turnOn();
}
```

message methods



Dynamic binding procedure

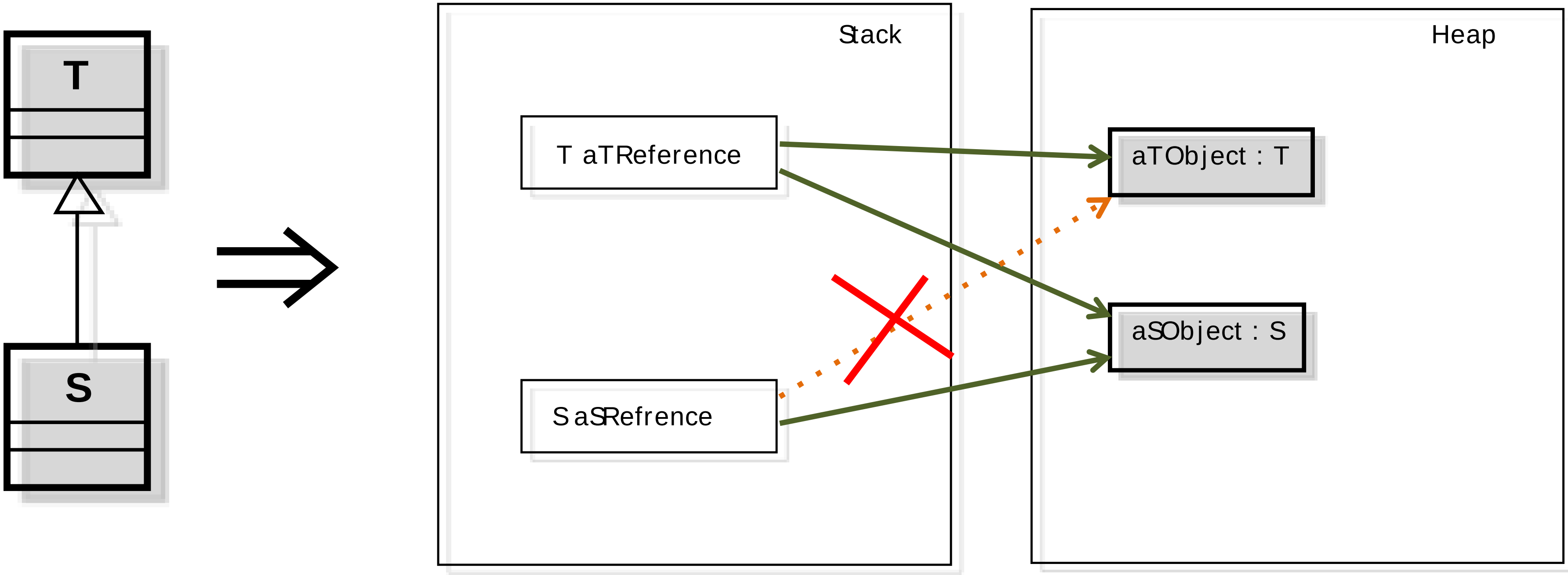
- The JVM retrieves the effective class of the target object
- If that class contains the invoked method it is executed
- Otherwise the parent class is considered and the previous step is repeated
- Note: the procedure is guaranteed to terminate
  - The compiler checks the reference type class (a base of the actual one) defined the method

Why dynamic binding

- Several objects from different classes, sharing a common ancestor class
- Can be treated uniformly
- Algorithms can be written for the base class (using the relative methods) and applied to any subclass

Substitutability principle

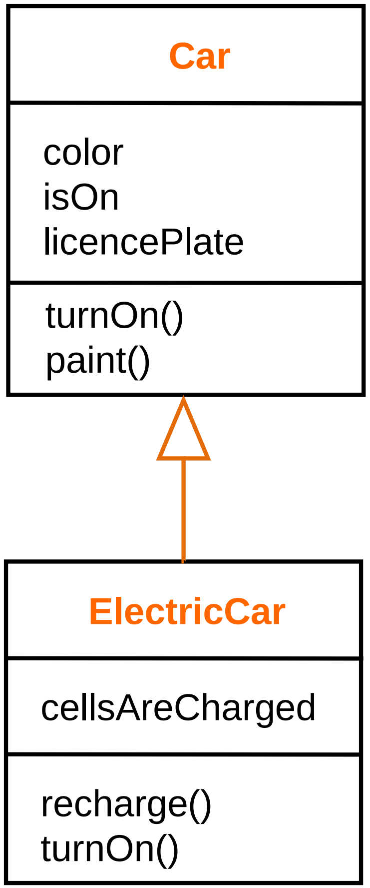
- If S is a subtype of T, then objects of type T may be replaced with objects of type S
  - A.k.a. Liskov Substitution Principle (LSP)



Masked overridden methods

- The presence of an override practically masks the method in the base class
  - Makes it invisible
- This behavior might be a problem if we wish to re-use the original overridden methods inside a subclass

Super (reference)




```java
class Car {

  String color;
  boolean isOn;
  String licencePlate;

  void paint(String color) {
    this.color = color;
  }

  void turnOn() {
    isOn=true;
  }
}
```

```java
class ElectricCar extends Car{

  boolean cellsAreCharged;

  void recharge() {
    cellsAreCharged = true;
  }

  void turnOn() {
    if(cellsAreCharged )
      super.turnOn();
  }
}
```

Super (reference)

- **`this`** is a reference to the current object
- **`super`** is a reference to the parent class

Attributes redefinition

```
class Parent{
    protected int attr = 7;
}
```

```
class Child extends Parent{
    protected String attr = "hello";

    void print(){
      System.out.println(super.attr);
      System.out.println(attr);
    }
}
```

Improper override

- A method override must use exactly the original method signature
  - Might widen visibility
- A slightly different method is not an override and therefore not considered in the dynamic binding procedure
- Annotation `@Override`
  - Inform the compiler that a method is intended as an override

VISIBILITY (SCOPE)

Example

```
class Employee {
  private String name;
  private double wage;
}
```

```
class Manager extends Employee {

  void print() {
    System.out.println("Manager" +
                        name + " " + wage);
  }
}
```

Not visible

Protected

- Attributes and methods marked as
  - public are always accessible
  - private are accessible from within the declaring class only
  - protected are accessible from within the class and its subclasses

In summary

	Method in the same class	Method of other class in the same package	Method of subclass	Method of class in other package
`private`	✓			
package	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

INHERITANCE AND CONSTRUCTORS

Construction of child's objects

- Since each object "contains" an instance of the parent class, the latter must be initialized
- Java compiler automatically inserts a call to default constructor (w/o parameters) of the parent class
- The call is inserted as the first statement of each child constructor

Construction of child objects

- Execution of constructors proceeds top-down in the inheritance hierarchy

- In this way, when a method of the child class is executed (constructor included), the super-class is completely initialized already

Example

```
class ArtWork {
  ArtWork() {
     System.out.println("ctor ArtWork"); }
}
```

```
class Drawing extends ArtWork  {
  Drawing() {
     System.out.println("ctor Drawing"); }
}
```

```
class Cartoon extends Drawing {
  Cartoon() {
     System.out.println("ctor Cartoon"); }
}
```

Example (cont’d)

```
Cartoon obj = new Cartoon();
```

```
ctor ArtWork
ctor Drawing
ctor Cartoon
```

A word of advice

- Default constructor “disappears” if custom constructors are defined

```
class Parent{
  Parent(int i){}

}
class Child
extends Parent{ }
// error!
```

```
class Parent{
  Parent(int i){}
  Parent(){} //explicit
}
class Child
extends Parent { }
// ok!
```

Super

- If you define custom constructors with arguments
- and default constructor is not defined explicitly

➔ the compiler cannot insert the call automatically

- The arguments cannot be inferred

Super

- The child class constructor must call the right constructor of the parent class, explicitly
- Use **`super()`** to identify constructors of parent class
- Must be the first statement in child constructors

Example

```
class Employee {
  private String name;
  private double wage;
       ???
  Employee(String n, double w){
    name = n;
    wage = w;
  }
}
```

```
class Manager extends Employee {
  private int unit;

  Manager(String n, double w, int u) {
    super();   ERROR !!!
    unit = u;
  }
}
```

Example

```java
class Employee {
  private String name;
  private double wage;

  Employee(String n, double w){
    name = n;
    wage = w;
  }
}
```

```java
class Manager extends Employee {
  private int unit;

  Manager(String n, double w, int u) {
    super(n,w);
    unit = u;
  }
}
```

Final method

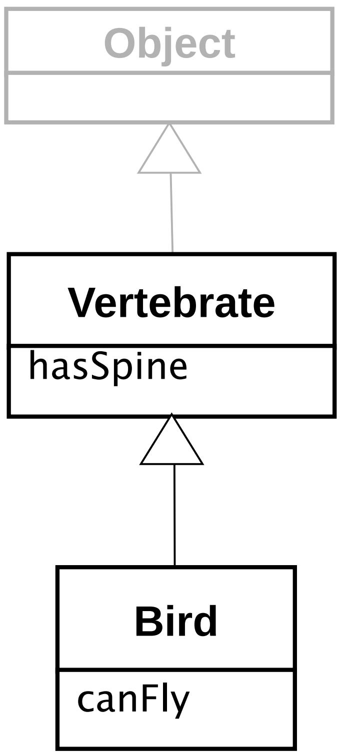
- The keyword `final` applied to a method makes it not overridable by subclasses
  - When methods must keep a predefined behavior
  - E.g. method provide basic service to other methods

Not an antinomy:
in Java there is a class called "Object"

OBJECT CLASS

Class `Object`

- **`java.lang.Object`**
- All classes are subtypes of Object



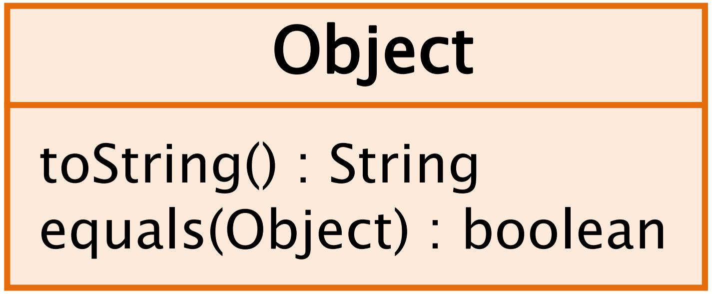
extends Object Implicitly

```
class Vertebrate {
   …
}
```

```
class Bird extends Vertebrate{
   …
}
```

Class `Object`

- **Each instance can be seen as an Object instance** (see Collection)
- Class **`Object`** defines some services, which are useful for all classes
- Often, they are overridden in sub-classes



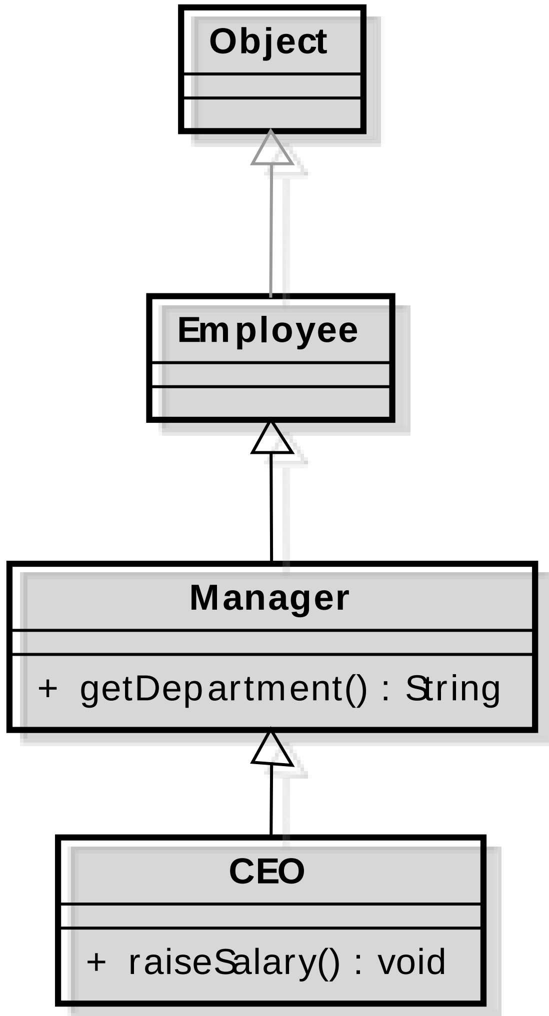
Objects' collections

- References of type **`Object`** play a role similar to **`void*`** in C

```
Object [] objects = new Object[3];
objects[0]= "First!";
objects[2]= new Employee("Luca","Verdi");
objects[1]= new Integer(2);
for(Object obj : objects){
  System.out.println(obj);
}
```

Wrappers must be used instead of primitive types

Company Employees



Upcast to Object

- Each class is either directly or indirectly a subclass of **`Object`**
- It is always possible to upcast any instance to Object type (see Collection)

```
AnyClass foo = new AnyClass();
Object obj;
obj = foo;
```

`Object` class methods

- **`hashCode()`**
  - Returns a unique code
- **`toString()`**
  - Returns string representation of the object
- **`equals()`**
  - Checks if two objects have same contents
- **`clone()`**
  - Creates a copy of the current object
- **`finalize()`**
  - Invoked by GC upon memory reclamation

Object.toString()

- **`toString()`**
  - Returns a string representing the object contents
  - The default implementation returns:

  `ClassName@#hash#`
  - Es:

  `org.Employee@af9e22`

Object
toString() : String equals(Object) : boolean

Object.equals()

- **equals()**
  - Tests equality of values
  - Default implementation compares references:

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

  - Must be overridden to compare contents, e.g.:

```
public boolean equals(Object o){
Student other = (Student)o;
return this.id.equals(other.id);
}
```

Object
toString() : String equals(Object) : boolean

The `equals()` Contract

- It is reflexive: **`x.equals(x) == true`**
- It is symmetric: **`x.equals(y)==y.equals(x)`**
- It is transitive: for any reference values x, y and z
- if **`x.equals(y)==true && y.equals(z)==true`** => **`x.equals(z)==true`**
- It is **consistent**: for any reference values x and y, multiple invocations of **`x.equals(y)`** consistently return true (or false), provided that no information used in equals comparisons on the object is modified.
- **`x.equals(null) == false`**

The `hashCode()` contract

- **`hashCode()`** must consistently return the same value, if no information used in **`equals()`** is modified.
- If two objects are equal for **`equals()`** method, then calling **`hashCode()`** on the two objects must produce the same result
- If two objects are unequal for **`equals()`** method, then calling **`hashCode()`** on the two objects *may* produce distinct results.
  - producing distinct results for unequal objects may improve the performance of hash tables

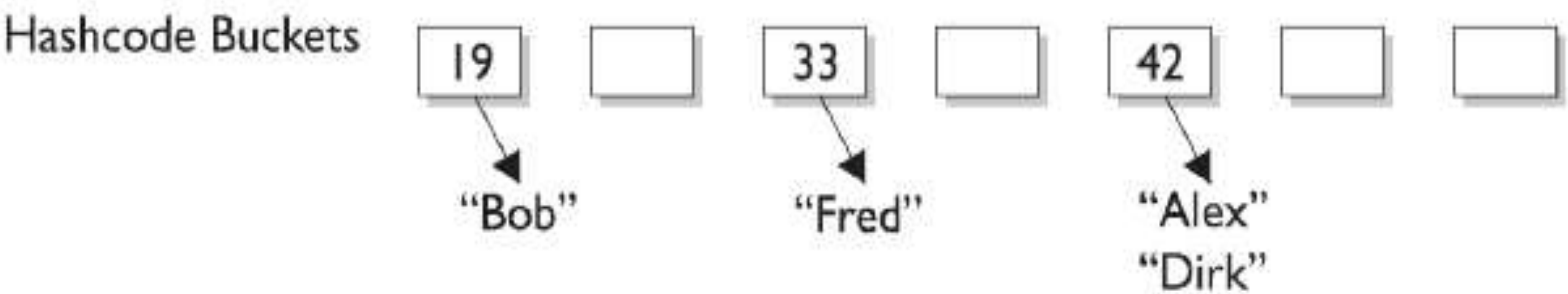
`hashCode()` vs. `equals()`

Condition	Required	Not Required (but allowed)
`x.equals(y) == true`	`x.hashCode()== y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y)== true`
`x.equals(y) == false`		-
`x.hashCode() != y.hashCode()`	`x.equals(y)== false`	

`hashCode` example

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) +I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + (D)	= 33

HashMap Collection



System.out.print(Object)

- *`print`* methods implicitly invoke **`toString()`** on all object parameters

```
class Car{ String toString(){…} }
Car c = new Car();
System.out.print(c);  // same as...
... System.out.print(c.toString());
```

- Polymorphism applies when **`toString()`** is overridden

```
Object ob = c;
System.out.print(ob);//Car's toString() called
```

Variable arguments– example

```
static void plst(String pre, Object...args){
    System.out.print(pre + ", ");
    for(Object o:args){
        if(o!=args[0]) System.out.print(", ");
        System.out.print(o);
    }
    System.out.println();
}
public static void main(String[] args) {
    plst("List:", "A", 'b', 123, "hi!");
}
```

ABSTRACT CLASSES

Abstract class

- Often, a superclass is used to define common behavior for many children classes
- Though some methods have no obvious implementation in the superclass
- The behavior is left partially unspecified
- The superclass cannot be instantiated

Abstract modifier

- The **abstract** modifier marks the class as non-complete
- The modifier must be applied to all incomplete method and to the class

```
public abstract class Expression {

   // to be implemented in child classes
   public abstract double evaluate();
}
```

No method body

Abstract modifier

```
public class Operand extends Expression {
private double value;
public Operand(double v){
   value = v;
}
   public double evaluate() {
      return value;
   }
}
```

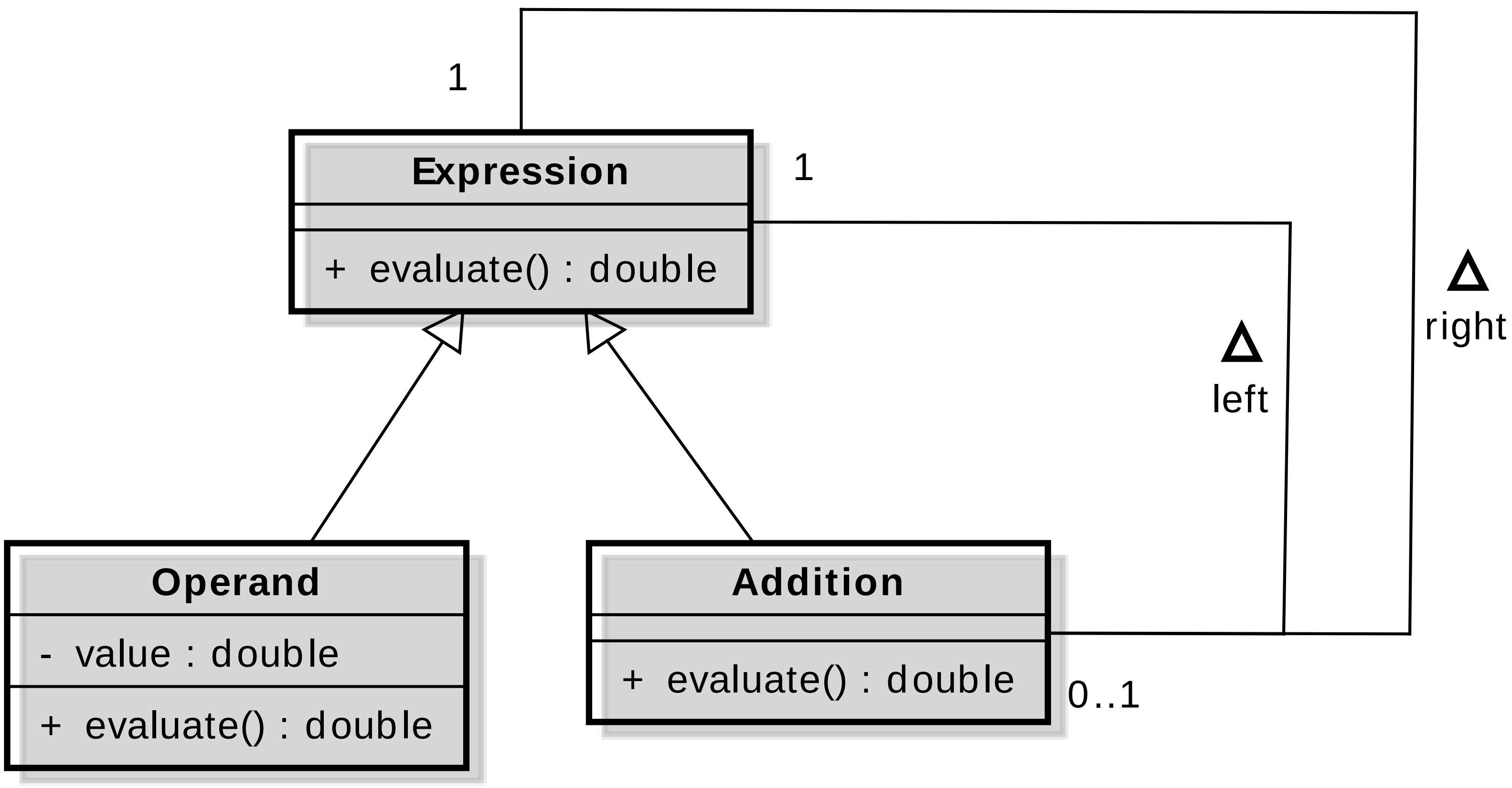
```
Expression e=new Expression();//No:abstract

Expression v=new Operand(1);// OK: concrete
```

Abstract modifier

```
public class Addition extends Expression {
private Expression left, right;
public Addition(Expression l,
                  Expression r){
  left=l; right=r;
}
   public double evaluate() {
      return left.evaluate()+right.evaluate();
   }
}
```

```
Expression s= new Addition(
                    new Operand(3),
                    v);
double res = s.evaluate();
```

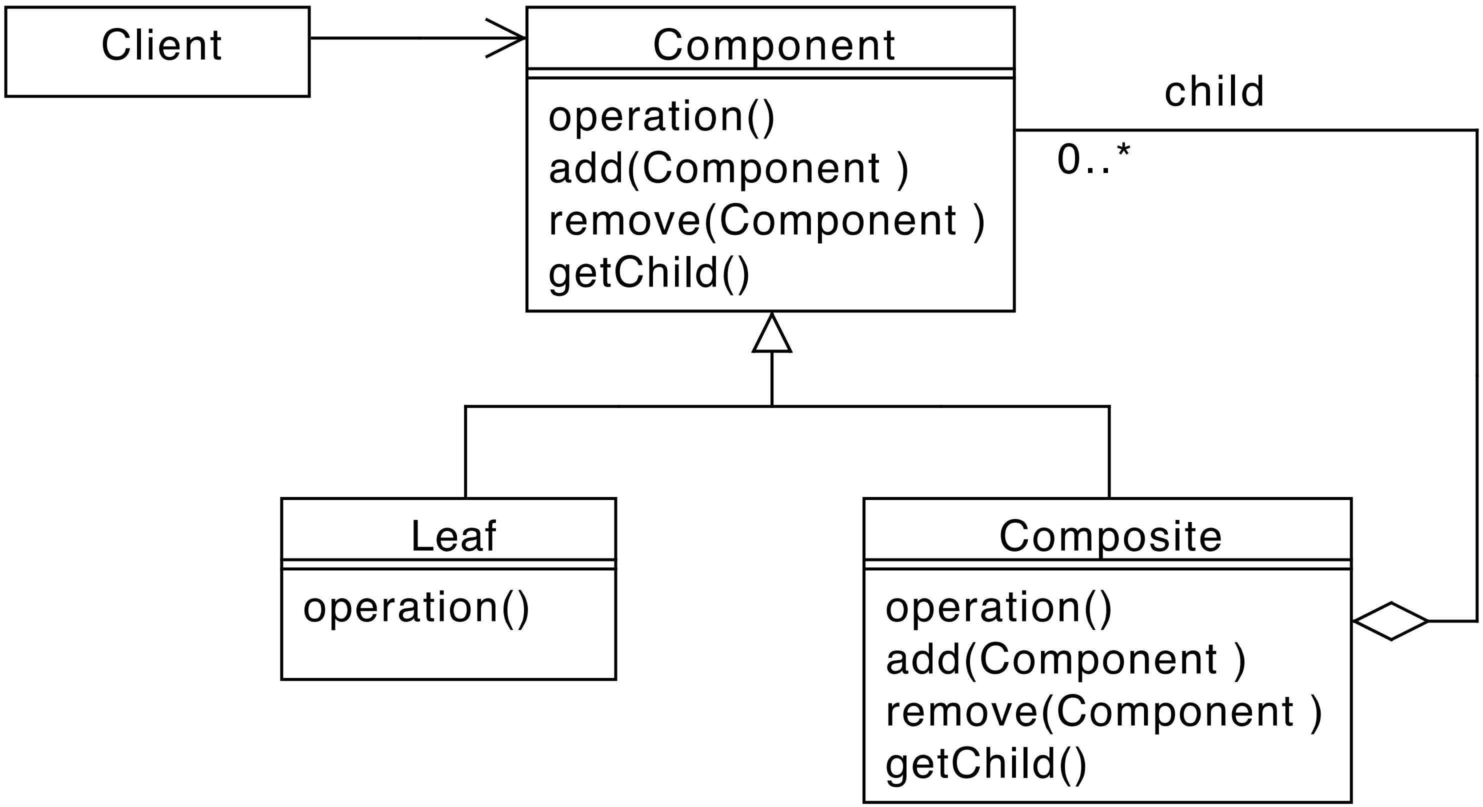
Abstract Expression Tree



Composite Pattern

- Context:
  - You need to represent part-whole hierarchies of objects
- Problem
  - Clients need to access a unique interface
  - There are structural difference between composite objects and individual objects.

Composite Pattern



Example: Sorter

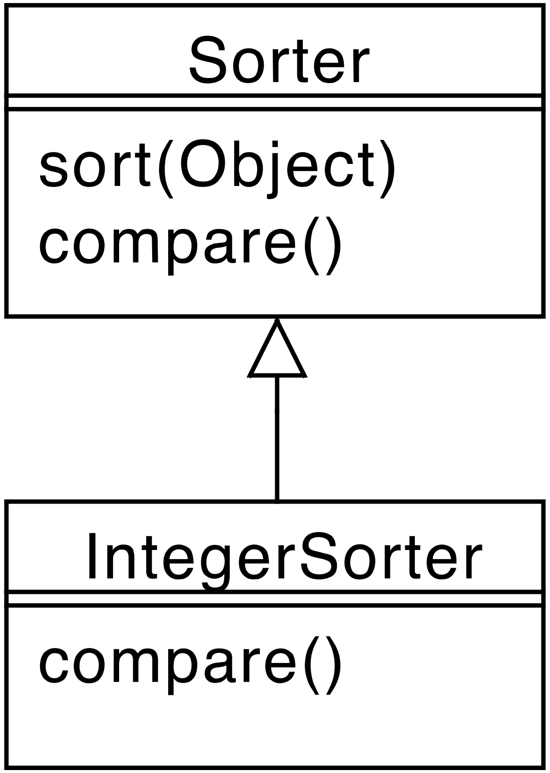
```
public abstract class Sorter {
  public void sort(Object v[]){
    for(int i=1; i<v.length; ++i)
      for(int j=1; j<v.length; ++j){
        if(compare(v[j-1],v[j])>0)
          Object o=v[j];
          v[j]=v[j-1]; v[j-1]=o;
      }
  }
  abstract int compare(Object a, Object b);
}
```

StringSorter

```
class StringSorter extends Sorter {
  int compare(Object a, Object b){
    String sa=(String)a;
    String sb=(String)b;
    return sa.compareTo(sb);
  }
}
```

```
Sorter ssrt = new StringSorter();
String[] v={"g","t","h","n","j","k"};
ssrt.sort(v);
```

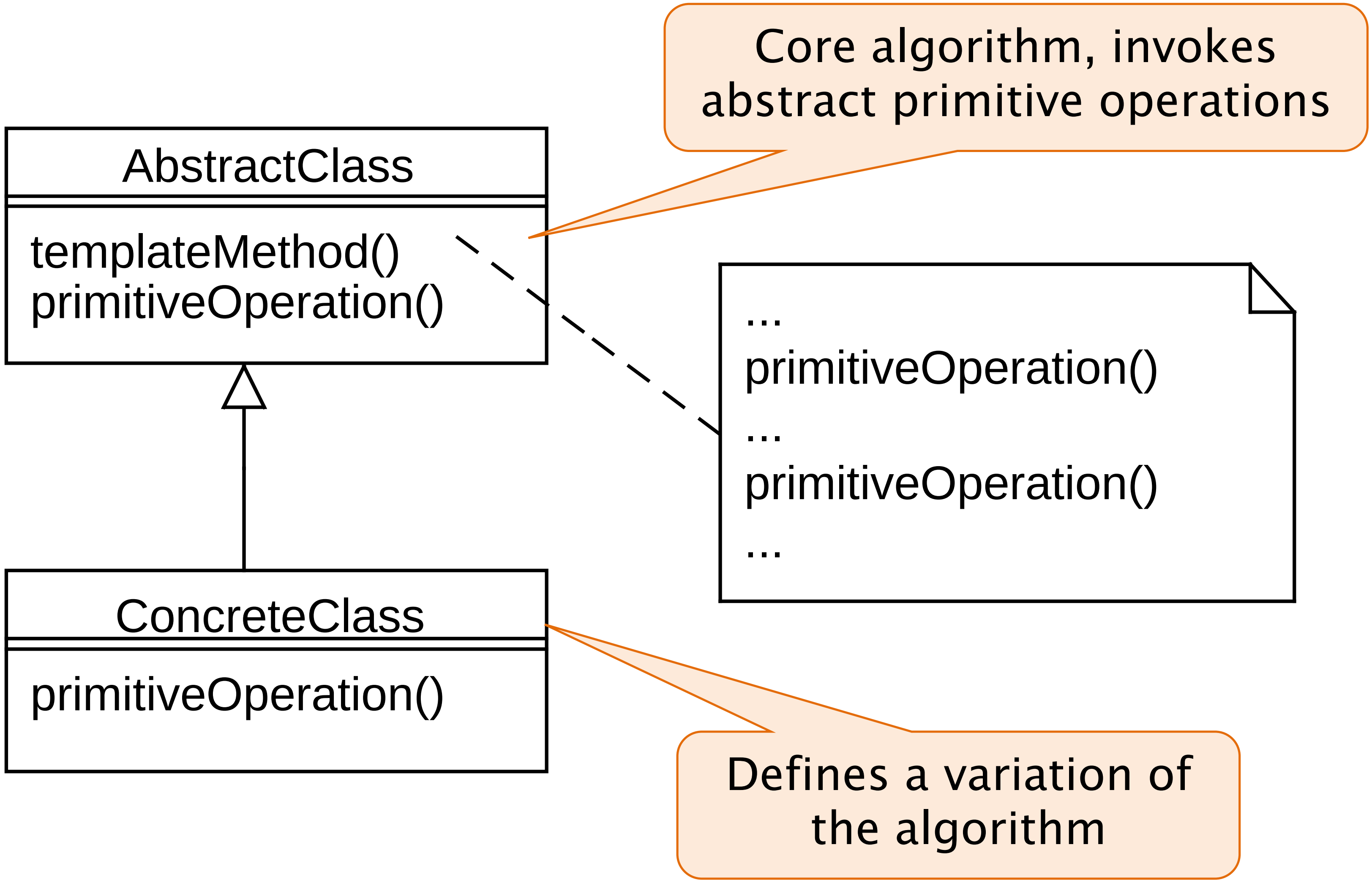
Template Method Example



Template Method Pattern

- Context:
  - An algorithm/behavior has a stable core and several variation at given points
- Problem
  - You have to implement/maintain several almost identical pieces of code

Template Method



INTERFACES

Java interface

- Special type of class where
  - Methods are implicitly abstract (no body)
  - Attributes are implicitly static and final
  - Members are implicitly public
- Defined with keyword **`interface`**
  - Instead of **`class`**
- Cannot be instantiated
  - i.e. no **`new`**
- Can be used as a type for references
  - Similar to abstract class

Interface implementation

- Class **implements** interfaces
  - A class must implement all interface methods unless the class is abstract
  - Interfaces are similar to abstract classes with only abstract methods

```
interface Iface {
void method();
}
```

```
class Cls implements Iface {
void method(){
   // . . .
}
}
```

Interfaces and inheritance

- An interface can extend another interface, cannot extend a class

```
interface Bar extends Comparable {
  void print();
}
```

interface

- An interface can extend multiple interfaces

```
interface Bar extends Orderable, Comparable{
  ...
}
```

interfaces

Class implementations

- A class can extend only one class
- A class can implement multiple interfaces

```
class Person
   extends Employee
   implements Orderable, Comparable {…}
```

Purpose of interfaces

- Define a common "interface"
  - Allows alternative implementations
- Provide a common behavior
  - Define a (set of) method(s) to be called by algorithms
- Enable Behavioral parameterization
  - Encapsulate behavior in an object passed as parameter
- Enable Communication decoupling
  - Define a set of callback method(s)

Alternative implementations

- Complex numbers

```
public interface Complex {
    double real();
    double imaginary();
    double modulus();
    double argument();
}
```

  - Can be implemented using either Cartesian or polar coordinates storage

Alternative implementations

- Context
  - Same module can be implemented in different ways by distinct classes with variations of:
    - Storage type or strategy
    - Processing
- Problem
  - The classes should be usable interchangeably
- Solution
  - Interface provides a set of methods with a well defined semantics and functional specification
  - Distinct classes can implement it

Common behavior: sorting

- Class **`java.utils.Arrays`** provides the static method **`sort()`**

```
int[] v = {7,2,5,1,8,5};
Arrays.sort(v);
```

- Sorting object arrays requires a way to compare two objects:
  - **`java.lang.Comparable`**

Comparable

- Interface **java.lang.Comparable**

```
public interface Comparable{
   int compareTo(Object obj);
}
```

- Semantics: returns
  - a negative integer if **this** precedes **obj**
  - 0, if **this** equals **obj**
  - a positive integer if **this** follows **obj**

Note: simplified version, actual declaration uses generics

Comparable

```
public class Student
        implements Comparable {
  int id;
  public int compareTo(Object o){
    Student other = (Student)o;
    return this.id - other.id;
  }
}
```

Common behavior idiom

- Context
  - An algorithm require its data to provide a predefined set of common operations
- Problem
  - The algorithm should work on a diverse set of classes
- Solution
  - Interface provides the set of required methods
  - Classes implement the interface and provide methods that are used by the algorithm

Common behavior: iteration

- Interface **`java.lang.Iterable`**

```
public interface Iterable {
  Iterator iterator();
}
```

- The class implementing `Iterable` can be the target of a *foreach* construct
  - Use the Iterator interface

Note: simplified version, actual declaration uses generics

Common behavior: iteration

- Interface `java.util.Iterator`

```
public interface Iterator {
  boolean hasNext();
  Object next();
}
```

- Semantics:
  - Initially before the first element
  - **`hasNext()`** tells if a next element is present
  - **`next()`** returns the next element and advances by one position

Note: simplified version, actual declaration uses generics

`Iterable` example

```
class Random implements Iterable {
  private int[] values;
  public Random(int n, int min, int max){ … }
  class RIterator implements Iterator {
    private int next=0;
    public boolean hasNext() {
        return next < values.length; }
    public Object next() {
        return new Integer(values[next++]);}
  }
  public Iterator iterator() {
    return new RIterator();
  }
}
```

`Iterable` example

- Usage of an iterator with for-each

```
Random seq = new Random(10,5,10);
for(Object e : seq){
  int v = ((Integer)e).intValue();
  System.out.println(v);
}
```

Iterator pattern

- Context
  - A collection of objects has to be iterated
- Problem
  - Multiple concurrent iterations are required
  - The internal storage must not be exposed
- Solution
  - Provide an iterator object, attached to the collection, that can be advanced independently

Behavioral parameterization

```
void process(Object[] v, Processor p){
  for(Object o : v){
    p.handle(o);
  }
}
```

```
public interface Processor{
  void handle(Object o);
}
```

```
String[] v = {"A", "B", "C", "D"};
Processor printer = new Printer();
process(v, printer);
```

```
public class Printer
implements Processor{
  public void handle(Object o){
    System.out.println(o);
}}
```

Behavioral parameterization

```
void process(Object[] v, Processor p){
  for(Object o : v){
    p.handle(o);
  }
}
```

```
public interface Processor{
  void handle(Object o);
}
```

```
String[] v = {"A", "B", "C", "D"};
Processor printer = new Processor(){
  public void handle(Object o){
    System.out.println(o);
  }};
process(v,printer);
```

Anonymous inner class

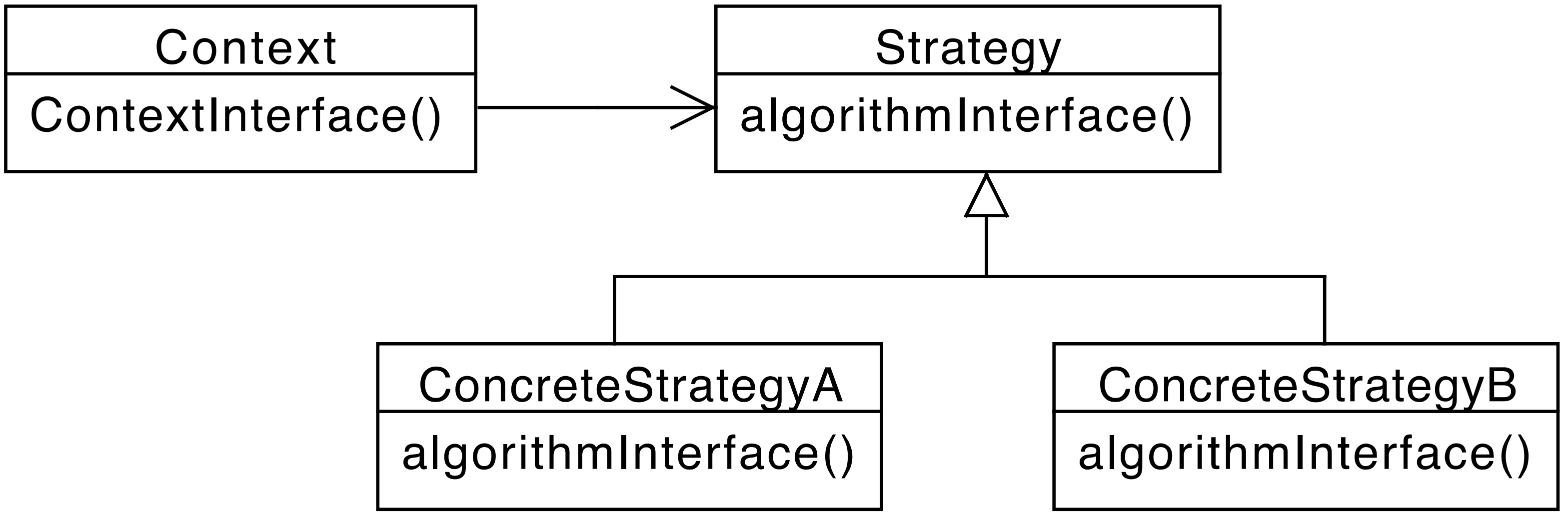
Strategy Pattern

- Context
  - Many classes or algorithm has a stable core and several behavioral variations
    - The operation performed may vary
- Problem
  - Several different implementations are needed.
  - Multiple conditional constructs tangle the code.

Strategy Pattern

- Solution
  - Embed inside a strategy object passed as a parameter to the algorithm
  - The strategy object’s class implements an interface providing the operations required by the algorithm

Strategy Pattern



Comparator

- Interface **`java.util.Comparator`**

```
public interface Comparator{
   int compare(Object a, Object b);
}
```

- Semantics (as comparable): returns
  - a negative integer if **`a`** precedes **`b`**
  - 0, if **`a`** equals **`b`**
  - a positive integer if **`a`** succeeds **`b`**

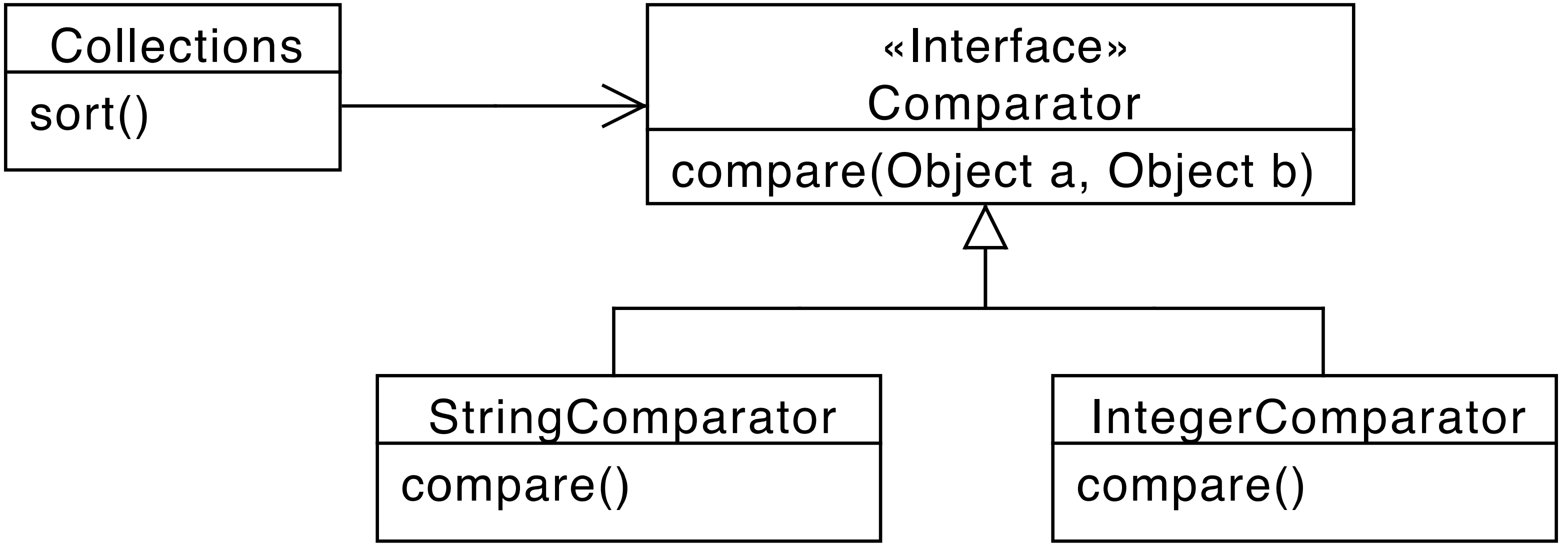
Note: simplified version, actual declaration uses generics

Comparator

```
public class StudentCmp implements Comparator{
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return a.id - b.id;
  }
}
```

```
Student[] sv = {new Student(11),
                new Student(3),
                new Student(7)};
Arrays.sort(sv, new StudentCmp());
```

Strategy Example: `Comparator`



Strategy Consequences

+ Avoid conditional statements
+ Algorithms may be organized in families
+ Choice of implementations
+ Run-time binding
- Clients must be aware of different strategies
- Communication overhead
- Increased number of objects

Comparator w/anonymous class

```
Student[] sv = {new Student(11),
                new Student(3),
                new Student(7)};
Arrays.sort(sv, new Comparator(){
  public int compare(Object a, Object b){
    Student sa = (Student)a;
    Student sb = (Student)b;
    return a.id - b.id;
}}
);
```

Communication decoupling

- Separating senders and receivers is a key to:
  - Reduce code coupling
  - Improve reusability
  - Enforce layering and structure

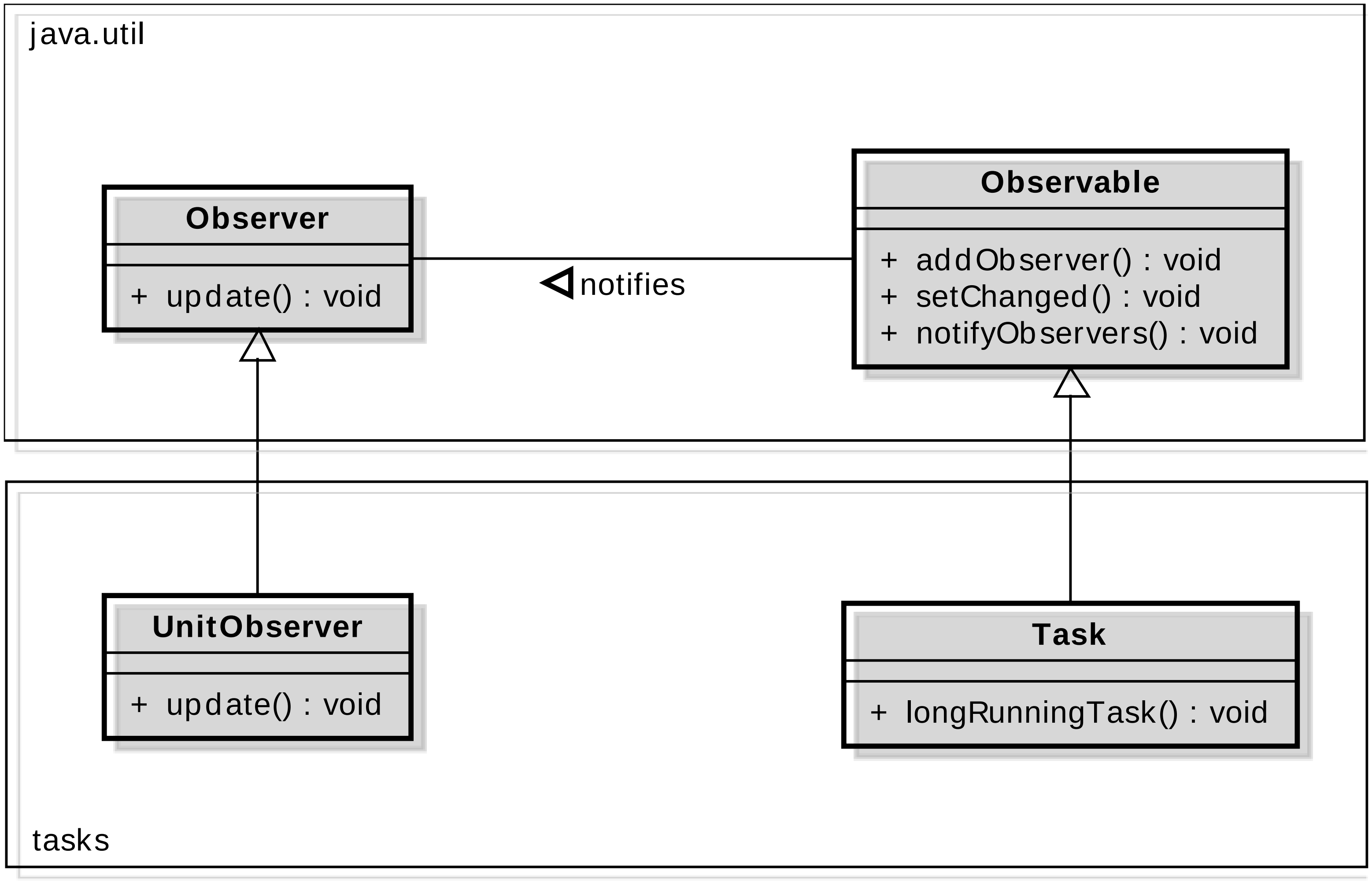
Observer-Observable

- Allows a standardized interaction between an objects that needs to notify one or more other objects
- Defined in package **`java.util`**
- Class **`Observable`**
- Interface **`Observer`**

Observer-Observable

- Class **`Observable`** manages:
  - registration of interested observers by means of method **`addObserver()`**
  - sending the notification of the status change to the observer(s) together with additional information concerning the status (event object).
- Interface **`Observer`** allows:
  - Receiving standardized notification of the observer change of state through method **`update()`** that accepts two arguments:
    - Observable object that originated the notification
    - Additional information (the event object)

Observer-Observable



Observer-Observable

- Sending a notification from an observable element involves two steps:
  - record the fact the the status of the observable has changed, by means of method **`setChanged()`**,
  - send the actual notification and provide additional information (the event object), by means of method **`notifyObservers()`**

Observer Pattern

- Context:
  - The change in one object may trigger operations in one or more other objects
- Problem
  - High coupling
  - Number and type of objects to be notified may not be known in advance

Observer Pattern

- Solution
  - Define a base Subject class that provides methods to
    - Manage observers registrations
    - Send notifications
  - Define a standard Observer interface with a method that receives the notifications

Observer – Consequences

- \+ Abstract coupling between Subject and Observer
- \+ Support for broadcast communication
- – Unanticipated updates

Default methods

- Interface method implementation can be provided for **`default`** methods
  - Cannot refer to non-static attributes
    - Since they are unknown to the interface
  - Can refer to arguments and other methods
  - Can be overridden as usual methods

Default methods motivation

- Enable adding new functionality to the interfaces of libraries and ensure compatibility with code written for older versions of those interfaces.
- Provide extra functionalities through multiple inheritance

Default method – Example

```
public interface Complex {
  double real();
  double imaginary();
  double modulus();
  double argument();
  default boolean isReal(){
    return imaginary()==0;
  }
}
```

FUNCTIONAL INTERFACES

Functional interface

- An interface containing only one regular method
  - `static` methods do not count
  - `default` methods do not count
- The semantics is purely functional
  - The result of the method is based solely on the arguments
    - i.e. there are no side-effects on attributes
  - E.g. `java.lang.Comparator`

Functional interface

- Predefined interfaces are defined in
  - **`java.util.function`**
  - Specific for different primitive types
  - Generic version (see Generics)
- The predefined interfaces can be used to define behavioral parameterization arguments
  - E.g. strategy objects

Functions (int versions)

- Function
  - `Object apply(int value)`
- Consumer
  - `void accept(int value)`
- Predicate
  - `boolean test(int value)`
- Supplier
  - `int getAsInt()`
- BinaryOperator
  - `int applyAsInt(int left, int right)`

Lambda function

- Definition of anonymous inner instances for functional interfaces

```
Processor printer =
            o -> System.out.println(o);
```

```
new Processor(){
  public void handle(Object o){
    System.out.println(o);
  }};
```

```
public interface Processor{
  void handle(Object o);
}
```

Lambda expression syntax

`parameters -> body`

- Parameters
  - None: `()`
  - One: `x`
  - Two or more: `(x, y)`
  - Types can be omitted
    - Inferred from assignee reference type
- Body
  - Expression: `x + y`
  - Code Block: `{ return x + y; }`

Type inference

- Lambda parameter types are usually omitted
  - Compiler can infer the correct type from the context
  - Typically they match the parameter types of the only method in the functional interface

Comparator w/lambda

```
Arrays.sort(sv,
   (a,b) -> ((Student)a).id -((Student)b).id
);
```

Vs.

```
Arrays.sort(sv,new Comparator(){
 public int compare(Object a, Object b){
   return ((Student)a).id -((Student)b).id;
 }});
```

Method reference

- Represent a compact representation of an instance of a functional interface that invoke single method.

```
Processor printer;
printer = System.out::println;
printer.handle("Hello!");
```

Equivalent to:

```
o -> System.out.println(o);
```

Method reference syntax

`Container::methodName`

Kind	Example
Static method	`Class::staticMethodName`
Instance method of a particular object	`object::instanceMethodName`
Instance method of an arbitrary object of a particular type	`Type::methodName`
Constructor	`Class::new`

Static method reference

- Similar to a C function
  - The parameters are the same as the method parameters

```
a,b -> Math.max(a,b)
```

```
DoubleBinaryOperator combine = Math::max;
double d=combine.applyAsDouble(1.0, 3.1);
```

```
package java.util.function;
interface DoubleBinaryOperator {
  double applyAsDouble(double a, double b);
}
```

Instance method of object

- Method is invoked on the object
  - Parameters are those of the method

```
v -> hexDigits.charAt(v)
```

```
String hexDigits = "0123456789ABCDEF";
Radix hex = hexDigits::charAt;
System.out.println("Hex for 10 : "
                        + hex.convert(10) );
```

```
interface Radix {
  char convert(int value);
}
```

Instance method reference

- The first argument is the object on which the method is invoked
  - The remaining arguments are mapped to the method arguments

```
s,i -> s.charAt(i)
```

```
StringValue f = String::charAt;
for(String e : v){
  System.out.println(f.apply(e,0));
}
```

```
interface StringValue {
   char apply(String s, int i);
}
```

Constructor reference

- The return type is a new object
  - Parameters are the constructor's parameters

```
i -> new Integer(i);
```

```
IntegerBuilder builder = Integer::new;

Integer i = builder.build(1);
```

```
interface IntegerBuilder{
    Integer build(int value);
}
```