Programming languages Java

Kitlei Róbert

Department of Programming Languages and Compilers
Faculty of Informatics, ELTE

The program's dynamic structure

- ***control flow***, ***execution***: the order in which the program's statements are executed
  - ◇ ***entry point***: control can be transferred to ∼s from the outside
  - ◇ ***main program***: the subprogram for the whole program's entry point
    - in Java (and most languages) this is called ***`main`***
  - ◇ procedural paradigm: main calls subprograms, subprograms call other subprograms and so on
  - ◇ happens in runtime
  - ◇ can be reasoned about statically
    - optimalisation: which variables "do not interfere" and therefore can be stored in the same space in memory?
    - code analysis: to which places of the code can control flow take a value?
    - unreachable code: which code parts are outside of the control flow?
    - dead code: are there computations whose values are ignored?

Call chain

- ***call chain***: the sequence of called subprograms starting from the entry point at a given point in time
  - ◇ when the called subprogram is finished, it returns, and now the call chain is one element shorter
  - ◇ the whole program is finished when the outermost subprogram (main) is finished
- ***active*** subprogram: an element in the call chain (still executing, hasn't finished)
  - ◇ the caller subprogram can give arguments to the callee
  - ◇ subprograms can maintain local variables while they are executed
  - ◇ the innermost active subprogram ***executes statements***

Call chain

- ***activation record***: contains the arguments and local variables of an active subprogram
  - ◇ ***stack***, ***call stack***: the activation records of the call chain are usually stored in it
    - when a new subprogram is called, a new entry is added to the stack
    - when the subprogram is finished, the last entry is removed
    - ... so it really works as a stack (LIFO - Last In, First Out)
    - the stack is also used to temporarily store the values of subexpressions during expression evaluation
  - ◇ ***stack frame***: name for activation records stored in the stack
- ***recursive*** subprogram: can be part of the call chain multiple times
  - ◇ more generally: ***reentrant***: a piece of code runs in many instances
  - ◇ either calls itself directly, or control returns to it through several intermediate calls
  - ◇ when this happens, the subprogram has many ***running instance***s in the call chain
  - ◇ the program will “freeze” or “crash” when it gets to infinite recursion

Memory

- ***memory*** or ***store***: sequence of bytes accessible to the program
  - ◇ ***memory address***: the index of a given byte in the sequence
  - ◇ the stack is stored in the memory
- ***allocation***: upon instantiation, a range of bytes is assigned to the object (disjoint from other objects' allocated bytes)
  - ◇ the starting address of the range is the ***address*** of the object
  - ◇ each ***instance variable*** has allocated space inside the range
  - ◇ ***pointer***: a variable that contains a memory address
    - ▸ ***null pointer***: a dedicated value (not necessarily 0 numerically)
    - ▸ ***dereferencing*** a pointer: accessing its memory location
      - dereferencing `null` always fails (`NullPointerException`)
    - ▸ ***pointer arithmetic***: using the memory address as a number, e.g. adding or subtracting to it
      - dangerous: it is easy to misuse, and point to an invalid address
  - ◇ ***reference***: a pointer that is guaranteed to point to an object (or it can be `null`)
    - ▸ Java only has references, no pointer arithmetic

Heap

- ***heap***: stores dynamically allocated data in the memory
  - ◇ all objects in Java go here

```
void f() {
    Point p = new Point(); // steps of evaluation:
    ...
}
```

1. first, the expression `new Point()` is evaluated
   1. the runtime allocates the necessary amount of space on the heap for the object
   2. the object is initialised (more on this later)
   3. the value of the expression is a reference to the object
2. variable `p` is put in the activation record of the innermost subprogram (`f`)
3. the memory address of the reference is copied into the variable
   - from here on, `p` refers to the new object

Lifetime/Extent

- ***lifetime*** or ***extent***: the portion of execution time when a value (an object) is present/accessible in memory
  - the values on the stack (local variables, formal parameters) only live as long as the activation record exists
    - so their lifetimes end when the subprogram exits
    - their values might continue to be present in the memory, but they are considered garbage
  - objects allocated on the heap live as long as they are being referred to
    - the exit of the subprogram in which their lifetime started does not necessarily remove them
      - the subprogram could return a reference to the object
      - the subprogram could load a reference of the object into the field of another object

Garbage collection

- in other languages, it is possible to explicitly ***deallocate*** objects
  - ◇ in Java, only allocation is possible
- ***garbage collection*** (GC): from time to time, the runtime system detects and deallocates objects that are inaccessible
  - ◇ it can be built into the language (e.g. Java)
    - usually, all objects are under its jurisdiction
    - it may be (partially) turned off in some languages
  - ◇ for other languages, they can be added on
- reason to use: manual memory management can be difficult
  - ◇ can be difficult to tell when an object has to be deallocated
    - it is very bad to deallocate an object still in use
  - ◇ it's easy to forget about deallocation → memory leak

Garbage collection: reference counting

- ***reference counting***: for each object, we store the number of references to it
  - it is fast: only a counter has to be increased/decreased
    - $+1$: `new`, pointing a new reference to the object
    - -1: pointing a reference away from the object
  - it's not good enough: circular references are never deallocated this way
- a number of ways exist that improve the basic algorithm

Garbage collection: mark and sweep

- ***tracing GC***: finds the reachable objects, and then deallocates the others
  - ***root set***: references that are in use
    - mostly, they are the references stored on the stack
    - goal: keep only objects accessible from the root set
  - two phases of the ***mark and sweep*** algorithm:
    1. starting from the elements of the root set, the GC traverses the graph of references
       - the found objects are ***mark***ed
    2. the objects that are left unmarked are reclaimed in the end (***sweep***)
- naïve implementation: the program cannot run during marking ("stop the world")
  - if it was allowed to do so, it could make changes in the reference graph, and invalidate the markings
  - improvements: incremental, parallel, generational, realtime

Reference types

- ***strong reference***: Java reference types covered so far
  - ◇ if an object is reachable via a strong reference, GC cannot reclaim it
- ***soft reference*** and ***weak reference***: if all references to an object are such, there is no guarantee that it will stay in memory
  - ◇ e.g. used for caching purposes
  - ◇ the GC may reclaim it, e.g. if the system is low on memory
  - ◇ weak references are reclaimed before soft ones are

```
import java.lang.ref.WeakReference;

WeakReference<Data> cacheData = new WeakReference<Data>(data);
Data data = cacheData.get();
         // --------------- we get a strong reference here
         //                 or null, if not present anymore
if (data != null) { /* it is alive, we can still use it */ }
```

Reference types: at extent's end

- the lifetime of an object is over when the GC removes it from memory
  - we have no influence over when it happens
- most of the time, we don't need to know when it happes, still it's possible to...
  - get notified when the object is reclaimed: its `finalize` method is invoked
    - it is possible to give the method a custom body
      - it can "revive" objects: point references towards objects scheduled for removal
      - it can call methods of "dead" objects
  - other sort of notification: ***phantom reference***
    - it cannot be used to access the object
    - when the GC schedules it for removal, it enters a `ReferenceQueue`
      - upon seeing this, we know that the object is reclaimed
    - a bit more flexible approach than `finalize`
- both ideas are dangerous, very rarely if ever useful, avoid them!

Basic notions

1. ***entity***: something different from everything else; something to be represented in the program
2. ***type***: entities grouped together by some sort of similarity
   - can be thought of as a (mathematical) set of entities
3. ***subtype***: its entities belong to another type (the ***supertype***) as well
   - with sets: $\{$entities of subtype$\} \subseteq \{$entities of supertype$\}$

- equivalents of the above in programming
  1. ***object*** $\leftarrow$ entity
  2. ***class*** $\leftarrow$ type
  3. ***inheritance*** $\leftarrow$ subtyping
- we call a programming language ***object oriented*** (OO) if it has (1.), (1. and 2.) or all three of the above
  - ◇ Java has all three

Entities

- ***property***
  - the name of a given person is John Smith, has a height of 180cm, age of 45 years
  - our level of abstraction dictates what we consider as a property
    - we could represent each cell of John Smith
- ***operation***s
  - we usually can't directly change the properties of entities in reality
  - however, we can perform operations on them
    - they can change the properties of the entities
- ***invariant***: a condition that always holds for an entity
  - e.g. neither the age, nor the height of a person can be negative
  - an invariant can involve several properties
    - e.g. all sides of a square are of the same length

Entities/objects/values

- representation in OO programming languages
  - ◇ object ← entity
  - ◇ data field ← property
  - ◇ method ← operation
  - ◇ invariant: not supported by most programming languages
    - ▸ they are available in research languages
      - the compiler can check/guarantee e.g. that the elements of a list are always in order
    - ▸ ***assert***: used to check conditions during runtime
      - a much weaker tool, but it is much easier to use
- the objects are the values usable in the program
  - ◇ … besides primitive types
  - ◇ it would be more convenient if all values were objects, but using primitive types is much more efficient

Types/classes

- ***class definition***
  - introduces the name of the class
  - `.java` source files mostly contain class definitions

```
class Point { int x; int y; }
```

- ***instantiation***: creation of an object (based on a class)
  - the created object is an ***instance*** of the class
  - ***instance variable***: for all data fields, a variable is created that belongs to the object
    - it can be referred to using the name of the field
  - the instantiation expression has a side effect (the object is created)
    - its return value is a reference to the object

```
new Point()  // creates an object of type Point
             // the parentheses are mandatory
(new Point()).x // accesses field x of the newly created point
new Point().x   // the outer parentheses are optional
```

Equality of references

- each instantiation creates a new object that is different from all previous ones
  - for objects, the operator == checks whether its operands refer to the same object

```
Point p  = new Point();    // (1)
Point p2 = new Point();    // (2)
System.out.println(p == p2); // false, (1) and (2) differ
System.out.println(new Point() == new Point());
  // false, it's the same as before, just without variables

String s  = readString();     // let us suppose we get   (3)
String s2 = readString();     // the string "abc" twice  (4)
System.out.println(s == s2);  // false because of the above
System.out.println("abc" == "abc");
   // true! String is special, the compiler optimises it
   //       and only one String object is created here
System.out.println("abc" == new String("abc")); // false
```

final modifier

- ***immutable*** primitive types: the value of the variable cannot be changed
    - convention: the variable name is uppercase

```
final int MAX_LINE_COUNT = 200;
MAX_LINE_COUNT = 300;                  // wrong
```

- immutable reference types: cannot be modified to refer to another object (or `null`)
    - that is, the pointer (with the memory address) is immutable
    - .. but the data of the object can change!

```
class C { int x; void setX(int x) { this.x = x; } }
final C c = new C();
c.x = 62;        // compiles
c.setX(51);      // compiles
c = new C();     // compilation error
```

Initialisation: instance variables

- instance variables cannot be left empty: they will get a value even if there's none in the source code

```
class Data { boolean b; int x; double d; String s; }
```

- the same, with the implicit initial values:

```
class Data {
    boolean b = false; int x = 0; double d = 0.0;
    String  s = null;  // its initial value is not ""
                       // because String is a reference type!
}
```

- it's bad practice to not write down the initial values if they coincide with the implicit ones
  - ◇ writing the value down indicates that we wish the field to get the value
  - ◇ an initial value can also be missing by mistake

Static variables

- ***static*** field: there is always exactly one instance of it, and that belongs to the class
  - ◇ the instances don't own an independent copy of it
  - ◇ it exists before any instances of the class are made
  - ◇ it is stored on the heap

```
class C {
    static int si;
           int d;
}

C cobj = new C();   // an object is created    (1)
C.si;    // OK: can use class name to refer to static fields
C.d;     // wrong: d would refer to an instance variable
cobj.si; // OK: objects can also refer to static fields
cobj.d;  // OK: refers to field d of cobj (that is, object (1)
```

Static variables: initial values

- gets its value when the class is loaded
  - this happens before the first use of the class
  - gets an implicit value like instance variables if no value is specified
- the initialiser expression can make use of the static variables that appear before it in the source code
  - instance initialiser expressions can use static variables

```
class C {
    // can use maxIdx although maxIdx is declared only later
    final int IDX = ++maxIdx;              // instance variable

    static final int START  = 12;          // static variable
    static       int maxIdx = START;       // static variable
}
```

Initialiser block

- ***initialiser block***: a block inside the class, runs upon instance initialisation
  - it can provide values for instance variables that require a longer computation
  - the order of instance variables and initialiser blocks is important
  - ***static initialiser block***: a similar tool on the instance level

```
class C {
    static final int    VSN = .....;
    static final String INIT_NAME;

    static { if (VSN >= 3)     INIT_NAME = "VSN3";
             else              INIT_NAME = "VSN_OLD";   }

    final List<String> names = ......;
    final String name;

    {   if (names.contains("X"))     name = INIT_NAME + "X";
        else                         name = INIT_NAME;         }
```

Constructor

- ***constructor***: code that runs upon object initialisation
  - ◇ its main goal: to set the invariants of the object
    - in other words: to make a valid object
    - to do that, the constructor sets the instance variables of the object
  - ◇ its name has to match that of the class

```java
class Point {
    int x;
    int y;

    Point(int x, int y) { this.x = x; this.y = y; }
}
```

Constructor

- if the code for the class doesn't contain a constructor, the compiler provides a ***default*** constructor
  - ◇ it has no parameters
  - ◇ its body is (almost) empty
  - ◇ its visibility is the same as that of its class

```
class Point {                               public class Point {
    // the compiler makes such a constructor if none is given
    Point() {                                   public Point() {
      /* "almost" empty body */                     /* "almost" empty */
    }                                           }
}                                           }
```

- no autogeneration if at least one constructor is present
  - ◇ of course, the programmer can choose to make a no-arg constructor

```
public class Point { public Point(int x) { ... } }

new Point();  // compilation error, no no-arg constructor!
```

Constructor

```
class Point {
    // if this is the only constructor, it cannot be
    // directly instantiated from outside the class
    private Point() { }

    static Point createPoint() { // ... only using a method
        return new Point();      // from which it is visible
    }
}
```

- methods can also take the name of the class, but it's a bad idea

```
class Point {
    // attention: this is not a constructor!
    public void Point() { }
        // ---- because of the specified return type
}
```

Constructor

- constructors can call other constructors
  - for that, the keyword `this` is used
  - it can only appear once, as the very first statement

```
public class Point {
    int x;
    int y;

    public Point(int x, int y) { this.x = x; this.y = y; }
    public Point()             { this(0, 0); }
}
```

- ***overloading***: more than one meaning is associated with a name
  - disambiguation is based on the number and types of arguments

```
new Point();        // calls the 0-arg constructor
new Point(3, 5);    // calls the (int, int) constructor
```

Parameter passing

- how are arguments passed upon method/constructor invocation?
  - ◇ ***call-by-value***: the parameter takes the value of the argument
  - ◇ ***call-by-sharing***: the argument is a reference (a pointer), this is taken by the parameter
    - the referred object is thus aliased
- in Java, primitive types are passed by value, and reference types are passed by sharing

```
class Point {
    int x;
    int y;

    Point(int x, int y) { this.x = x; this.y = y; }
      // -----  -----  receives both arguments by value
    Point(Point p)      { this(p.x, p.y); }
                              // ---  --- calls by value
      // ------- receives the argument by sharing
}
```

Encapsulation

- ***encapsulation***: the object is responsible for its state
  - ◇ goal: the invariants have to be preserved
    - the object's state should be changeable from the outside only through method calls
  - ◇ throughout its lifetime, only the object's code should handle its data
    1. upon instantiation, the constructor makes sure the invariant is set
    2. as the state can only change through method invocations, methods have a "contract"
       a. the method can assume that the invariants hold when the method is called
       b. by the time the method ends, it should get the object into a state where the invariants hold
- how can the encapsulation be broken?
  - ◇ if the fields are accessible from the outside (public)
  - ◇ if the object provides access to part of its representation to the outside

Encapsulation: messing up

```
class Point {  private int[] coords = { 0, 0 };
               int  getX()       { return coords[0]; }
               void setX(int x) { coords[0] = x;     } }
```

- objects' interfaces have to be well thought out
  - ◇ what happens if we use a different getter/setter?

```
class PointN {
    private int[] coords;
    PointN(int[] coords)     { this.coords = coords; } // !!!
    int[] get()              { return coords;        } // !!!
    void  set(int[] coords) { this.coords = coords; } // !!!
}

int[] coords = { 1, 2, 3, 4, 5 };
PointN p = new PointN(coords);
coords[2] = -7;                       // wrong: the "private" field
System.out.println(p.get()[2]); // of p also refers to here!
```

Encapsulation: doing it right

- the object must not let the internally handled references get out
  - it must also not store data that is accessible from the outside

```
class PointN {
    private int[] coords;
    PointN(int[] coords)    { set(coords);                  }
    int[] get()             { return coords.clone();        }
    void  set(int[] coords) { this.coords = coords.clone(); }
}
```

- ***shallow copy***: only the outmost references are separated
  - used on arrays, the `clone` method makes such a copy
- ***deep copy***: when shallow copying is not enough, all internal references are copied

```
Point[][] ptss  = { { new Point(1, 2) } };
Point[][] ptss2 = ptss.clone();   // ptss[0]    != ptss2[0]
ptss[0][0].x = 100;               // ptss[0][0] == ptss2[0][0]
System.out.println(ptss2[0][0].x);// 100, b/c of shallow copy
```

Method: local variables

- if we want to use the object later on, we can introduce a ***local variable***
  - ◇ we can also have the instance variable of another object refer to it
  - ◇ ***initialisation***: the first time a value is assigned to a variable
    - here we look at the initialisation of local variables in methods

```
Point p;     // declaration statement without initialisation
Point p2 = new Point();// declaration with initialisation (1)
int   x  = p.x;        // compilation error, p is uninitialised
p2   = new Point();    // assignment, (1) is unreachable now (2)
p    = new Point();    // assignment                          (3)
p2   = p;              // (2) is unreachable too,
                       // p and p2 both refer to (3)
p2.x = 5;              // field x in (3) is assigned the value 5
p.y  = p2.x;           // fields x and y in (3) both become 5
```

Initialisation of local variables

- local variables of methods must be initialised before use

```
Point p;          // a local variable in a method
int   v = p.x;   // compile error, p is uninitialised
```

- initialisation may be done in a statement separate from its declaration
  - ◇ it's better to do initialisation together with the declaration, if possible

```
Point p; // the value for this variable cannot be set here
if (condition) { p = new Point(1,2); }  // because it depends
else           { p = new Point(3,4); }  // on a condition
int v = p.x;  // p can be used here: it already has a value
```

- the compiler checks whether both paths assign to the variable
  - ◇ the compiler cannot check complex conditions

```
Point p;
if (alwaysTrue()) { p = new Point(1,2); }
int v = p.x;  // variable p might not have been initialized
```

Initialisation: variables

- variables could be initialised with `null`

```
Point p = null; // this is not a good idea, usually
int   v = p.x;  // throws NullPointerException in runtime
```

- `null`: according to its inventor (Tony Hoare): "billion-dollar mistake"
  - all variables can take `null`
  - ... so it can appear everywhere where we expect a valid object
    - it can spread very far
    - it's hard to find out where it has come from
  - it is not compulsory to check for `null` when using a variable
  - very dangerous, avoid it if you can!
- languages tend to include compulsory checking more and more
  - Java 8: for a reference type `T` there is `java.util.Optional<T>`

Declaration

- ***declaration***: something (e.g. variable, method, class) is assigned a name
- ***scope***: the part of the code where the declaration is valid
  - it is almost always determined in compile time (lexical scoping)
    - ... when it is dynamic (e.g. Lisp, Perl), much harder to understand
  - ***hiding***, ***masking***: redeclaration of a name inside its scope
    - the name is now used in its new meaning, for the old one, it is necessary to use a qualifier
    - ***visibility***: a name is not hidden

```
class C {
    int x;            // (1)
    void f(int x) {   // (2)
        System.out.println(x);      // refers to (2)
        System.out.println(this.x); // refers to (1)
    }
    boolean g() { return  x == this.x; }  // always true
                  //  ---   ------ no hiding,
                  //               both are the same
}
```

Scope: block

- groups statements
- limits the visibility of variables declared inside
- most commonly used as the body of `if`s and loops
  - ◇ the body of functions is technically not a block statement (although surrounded by braces)
- another use: you can limit the visibility of variables

```
{
    int sumOfTenNumbers = 0;
    for (int i = 0; i < 10; ++i) {
        sumOfTenNumbers += inputNumberFromUser();
    }
    // sumOfTenNumbers is visible here
}

// sumOfTenNumbers is not usable anymore
```

Scope: hiding

- Java: local variables (including formal arguments) can hide the name of fields
  - ... but they cannot hide other local variables

```
class C {
    int x;              // (1)
    void f(int x) {  // (2)
        int x; // invalid: (2) is a local variable
               //              with the same name

        int y; // (3)
        {
            int y; // invalid because of (3)
        }
    }
}
```

Scope: hiding: getter/setter

- hiding is often used in `setter` methods

```
class Point {
    private int x;
    private int y;

    public int getX() { return /*this.*/x; }
    public int getY() { return /*this.*/y; }

    public void setX(int x) {
                  // ----- the name of the formal argument
                  //       can be the same as that of a local
               this.x = x;
            // ------      refers to the instance variable
            //         --- refers to the formal argument
    }
}
```

Access modifier

- ***access modifier*** or ***visibility modifier***: decreases the scope of a class/field/method/constructor
  - `public`: such methods form the ***interface*** of an object: they are the services that an object provides

```
package p1;
public class A {
    private   int priv;
              int pkg;  // package private
    protected int prot; // rarely used
    public    int publ;
}
class B { A a;    int f(){return a.prv+a.pkg+a.prt+a.pub; }}
                                   // XXXXX
=====================================================
package p2;                        // XXXXX XXXXX
class C extends A{int f(){return   prv+  pkg+  prt+  pub; }}
class D { A a;    int f(){return a.prv+a.pkg+a.prt+a.pub; }}
                                   // XXXXX XXXXX XXXXX
```

Access modifier: classes

```
package p1;
public class A { }   // public
class B        { }   // package private
class CanAccessBoth {
    void f() {
        new A(); // OK
        new B(); // OK
    }
}

=============================================================

package p2;
class CanOnlyAccessPublic {
    int f() {
        new A(); // OK
        new B(); // error: B is not public in p1;
                 // cannot be accessed from outside package
    }
```

Method parts

- method declaration

```
class Point { int x; int y;
                // body          -----------------------
    void move(int dx, int dy) { x += dx; y += dy; }    }
 //     ---- ---     ---       signature
 //          ---------------   formal argument list
 // ----                       type of return value
 //     ----                   name (identifier) of method
 // ---------------------------- header
```

- ***instance method***: operates on an instance
  - ◇ unlike ***instance variable***s, methods are not stored inside instances

```
Point p = new Point(); // (1)
p.move(3, 5*4+8); // instance method "move" is called on (1)
  //  -------- (actual) argument list for the call
     // the argument expressions are evaluated;
     // these values are passed into the formal arguments
```

Method: this

```
class Point {
    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }

    void move(/* Point this, */ int dx, int dy) {
           // ---------------- implicit formal argument
        this.x += dx;
        this.y += dy;
     // ------         refers to the fields
     //           ---- refers to the formal arguments
    }
}
```

- all instance methods have an implicit `this` argument

```
Point p = new Point();  // (1) is created, p refers to it
p.move(3, 5);           // in the call, "this" refers to (1)
```

Method: chaining

```
class Point {
    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) { this.x = x; }

    Point setY(int y) { this.y = y; return this; }
 // -----                               -----------
}
```

- ***method chaining***: code in the above style can be used like this:

```
Point p = new Point();
p.setX(3).setY(-2).move(11, 5);
```

Method: return value

- the return type is part of the declaration of all methods
  - for functions (pure subprograms), this is really a type
  - procedures (impure subprograms) do not return values, this is indicated with the keyword ***void***
- formal arguments (parameters) act as local variables in the method during its execution

```
class C {
    void f(int x) { if (x>5) return; System.out.println(x); }
                                // ------ "return" in a void method
                                //         can only stand alone

    int g(int a, int b) {
        return a*a+b/2;
            // ------- in case of a non-void return value
            //         an appropriately typed expression
            //         has to appear here
    }
}
```

Method: static method

- static methods do not have an implicit this argument

```
class C {
    static void f(int n) { ... }
}
```

- they can be called on objects and also on the name of the class

```
C c = new C();
c.f(53);
new C().f(1);
C.f(-9);
```

- static method can only call another static method
  - ◇ ... except if an object reference is available
  - ◇ a well-known static method: main

Inheritance

- ***Inheritance***, ***subclassing***

```
class A {
    int  a;
    void f(char c) { ... }
}

class B extends A {
    int b;
    int g(String s) { ... }
}

class C extends B { ... }
```

- `A` is the ***parent class*** of `B` (`B` cannot have other parents)
- `B` is a ***child class*** of `A` (`A` can have other children)
  - `A` and `B` are ***superclass***es (or ***base class***es or ***ancestor***s) of `C`
  - `C` is a ***derived class*** (or ***descendant***) of `A` and `B`

Inheritance relation

- chief goals of inheritance
  - ◇ to reuse the code of the base classes, therefore
  - ◇ to decrease the redundance of the code
- inheritance relation: a partial ordering between classes
  - ◇ it forms a directed graph
    - no class can be based upon itself, so the graph does not contain directed cycles
    - there is no multiple inheritance between classes in Java
    - all classes are derived from the class `java.lang.Object`
  - ◇ summed up: the inheritance between the classes is a ***directed tree***

Why only single inheritance between classes?

- multiple inheritance between classes would allow the following:

```
class A                  { void f() { /* body in  A */ } }
class B1 extends A       { void f() { /* body in B1 */ } }
class B2 extends A       { void f() { /* body in B2 */ } }
class C  extends B1, B2 { void f() { /* what now?  */ } }
```

- B1 and B2 both replace the body of f
- ***diamond inheritance problem***: what should the body in C be like?
  - ◇ it has to conform to both B1 and B2 (see: substitution principle)
- the designers of Java decided to keep it simple, and avoid such questions

Inheritance: fields, methods

- what is inherited?
  - ◇ all fields
  - ◇ all methods (both their headers and their bodies)

```
class A /* extends Object */ {       // if left unspecified,
    public  int    val;              // the parent is Object
    private String txt;

    String f() { return txt + val; }
}

class B extends A { }
```

- inherited fields/methods are not necessarily accessible

```
B b = new B();
System.out.println(b.val);  // OK
System.out.println(b.txt);  // txt is not accessible
System.out.println(b.f());  // OK
```

Inheritance: constructors

- the constructor of the child class must call the constuctor of the parent class

```
class A {
    public A(int val, String txt) { ... }
}

class B extends A {
    public B(String txt) {
        super(8, txt); // can only be the first statement
        ...            // in the constructor body
    }

    public B() {
        this("abc");   // calls the parent's constructor
    }                  // through the other constructor
}
```

Inheritance: constructors

- constructors are not inherited
  - ◇ if we need a constructor similar to that of the parent, we have to explicitly put it here, too

```
class A {
    public A(char c) { ... }
}

class B1 extends A { }
class B2 extends A {
    public B2(char c) { super(c); ... }
}


new B1('x');  // B1 does not have such a constructor
new B2('o');  // OK
```

Inheritance: constructor: super

- if no this or super starts the body, a super() call is autogenerated
  - ◇ induces a compilation error if the parent class has no 0-arg constructor

```
class A {
    // suppose we have no zero-arg constructor in A
    // no default constructor is generated either
    public A(int val, String txt) { ... }  // b/c of this one
}

class B extends A {
    public B(String txt) {
     // super();     // implicitly generated
        ...          // would call 0-arg (not present in A)
    }                // compilation error
}

class B2 extends A { /* B2() { super(); } */ }
                     // ----------------- not good either
```

Inheritance: incorrect usage

- inheritance can technically be used to simply extend the class with fields and methods

```
class BadSquare {        class BadRect extends BadSquare {
    int aSideLen;            int bSideLen;
}                        }
```

- however, inheritance also induces ***subtyping***: objects from the derived class can be used in all places where the superclass is expected
  - ◇ if `class B extends A`, then we regard them as $B \subseteq A$

```
BadSquare n = new BadRect(); // a rectangle should not
                             // identify as a square
```

```
class GoodRect {         class GoodRect extends GoodRect {
    int aSideLen;            // downside: the correct solution
    int bSideLen;            // uses more space in this class
}                        }
```

Substitution principle

- ***Liskov substitution principle, 1987***: all properties of the supertype must hold for the subtype
  - ◇ the subtype must be able to stand in for the supertype
  - ◇ inheritance (if the language has it) is a natural tool to support this
- ***aggregation***: adding the supertype as a field in the subtype
  - ◇ ***composition***: aggregation, where the lifetimes of the supertype and the subtype are linked
    - ▸ in some cases, more appropriate solution than inheritance
  - ◇ to support the principle, the interface of the supertype has to be duplicated in the subtype

```
class Person  {                     public void f() {......} }
class Student { private Person p; public void f() {p.f();} }
```

- ***duck typing***: “if it walks like a duck and quacks like a duck then it is a duck”
  - ◇ checks the existence of the necessary fields/methods in runtime
  - ◇ possible to do in Java, but inheritance is preferable in general

Polymorphism

- ***polymorphism***: handling different data types in a common way
  - ◇ inheritance is a special case: ***subtype polymorphism***

```java
class A {}
class B extends A {}

class C {
    public static void f(A a){ ... }
    public static A    g()   { return new B(); }
                    // ---                  ------- it's valid
                    // callers will only see it as having type A
}

C.f(new B());  // OK, even though the static type of
               // formal argument "a" is A
A a = C.g();   // OK
B b = C.g();   // ERROR, the expression has static type A
B b = (B)C.g(); // solution: downcast (more on it later)
```

Static and dynamic type

- ***static type***: type of the variable known in compile time
- ***dynamic type***: type of the value bound to the variable
  - only known in runtime
  - can change in time, because the variable might refer to different values
  - always the subtype of the static type

```
List<A> as = new ArrayList<>();
Random rnd = new java.util.Random();
for (int i = 0; i < 100; ++i) {
    A a;

    if (rnd.nextBoolean()) a = new B1(); // B1 and B2 both
    else                   a = new B2(); // extend A

    // the dynamic type of "a" is either B1 or B2 here
    as.add(a);
}
```

Upcast, downcast

- ***upcast***: runtime conversion from a subclass to a superclass (“upwards” in the type hierarchy)
  - ◇ it is always permissible to do so
  - ◇ can only be done between classes derived from one another
  - ◇ does not change the representation of the object
- ***downcast***: runtime conversion from a superclass to a subclass
  - ◇ checks in runtime whether the object’s dynamic type is suitable

```
class A           { int a; }
class B extends A { int b; }

new B().a      = 1;   // what really happens is:
((A)new B()).a = 1;   // upcast, OK

((B)new A()).a = 1;   // downcast, runtime ClassCastException
((Object)new B()).a = 1; // compile error
                         // Object does not have a field "a"
```

The `instanceof` operator

- ***instanceof***: type checking operator
  - all lowercase! (not `instanceOf`)
  - right operand: the name of a type
  - left operand: expression
  - evaluates to `true` iff the dynamic type of the given expression is a subtype of the given type

```
class A           { int a; }
class B extends A { int b; }

null instanceof T    // false for any T
new A() instanceof A // true
new B() instanceof A // true
new A() instanceof B // false
"abc"   instanceof A // compile error: incompatible types:
                       // String cannot be converted to A
A a = new B();
a instanceof B // true (the static type of "a" doesn't matter)
```

The instanceof operator

- we only use instanceof in rare cases

```
class A { void doSomethingA() { ... } }
class B { void doSomethingB() { ... } }

Object o = ...;
if      (o instanceof A)   ((A)o).doSomethingA();
else if (o instanceof B)   ((B)o).doSomethingB();
```

- better suited to object orientation: use a common base class (if possible)

```
class Base           { void do() { ... } }
class A extends Base { void do() { ... } } // method override
class B extends Base { void do() { ... } } // in both classes

Object o = ...;
((Base)o).do();
```

Overriding

```
// we'll use these types in the following examples
class T1 {}
class T2 extends T1 {}

// we'll override f in this class in the examples
class Base { T1 f(T1 t) { return t; } }
```

- **overriding**: we give the method a new body in the subclass
- visibility can change from package private to public

```
class Ext extends Base { public T1 f(T1 t) { return t; } }
                      // ------ package private -> public
```

Overriding

- the parameters may not change (not even to a subtype)
  - if the `@Override` annotation is present, we get a compilation error

```
class Ext extends Base {
    @Override
    public Type1 f(Type2 t) { return t; }}
                // ----- compilation error
```

- the code is valid without the annotation, but its meaning is different
  - the method is overloaded in this case, probably not what we wanted

```
class Ext extends Base {
    public T1 f(T2 t) { return t; }}
            // -- valid without @Override, but...

new Ext().f(new T1()) instanceof T2)  // == false
      // ----------- calls Base.f
new Ext().f(new T2()) instanceof T2)  // == true
      // ----------- calls Ext.f
```

Overriding

- the return type can change to a subtype
  - this is **covariance**: as we go to a more specific type (Base→Ext), the return type changes in the "same direction"

```
class Ext extends Base {
    @Override
    public T2 f(T1 t) { return new T2(); }
        // -- this is OK       -------- and so is this
}
```

- we have already seen that the types of the parameters are **invariant**
  - other languages also feature **contravariance**

Overriding

- using the `super` keyword, we can access a method/variable/constructor of our ancestor
  - ◇ `((Base)this)` cannot stand in as a substitute

```
class Base { T1 f(T1 t) { return t; } }
class Ext extends Base {
    T1 f(T1 t) { return t == null ? new T2() : super.f(t); }}

Base b  = new Base();
Ext  e  = new Ext();
Base b2 = new Ext();
T1   t1 = new T1();
b.f(t1);     // == t1
b.f(null);   // == null
e.f(t1);     // == t1
e.f(null);   // a new T2 instance
b2.f(t1);    // == t1
b2.f(null);  // a new T2 instance; see next page
```

Dynamic binding

- ***dynamic binding***, ***late binding***: upon instance method call, the dynamic type (the type of the instance) determines which body will run
  - ◇ the runtime system does a lookup along the chain of base classes
    - ▸ it chooses the body defined closest to the dynamic type
  - ◇ takes a little more time in runtime
  - ◇ the compiler checks that the method is callable based on static types
    $\Rightarrow$ it is guaranteed that the runtime system will find a body to call

```
class A           { void f() { ... } } // (1)
class B extends A {                  }
class C extends B { void f() { ... } } // (2)
class D extends C {                  }

new A().f();    // lookup order: A    -> calls (1)
new B().f();    // lookup order: B, A -> calls (1)
new C().f();    // lookup order: C    -> calls (2)
new D().f();    // lookup order: D, C -> calls (2)
```

New body: static methods

- static methods cannot be overriden
  - the static methods of subclasses ***hide*** similar methods from the superclass
  - no dynamic binding between them: calls are decided in compile time
  - in general, much better to avoid this altogether, and give the method a different name

```
class A           {static void f(){System.out.println("A");}}
class B extends A {}
class C extends B {static void f(){System.out.println("C");}}

A.f(); // -> A
B.f(); // -> A, because it is closest in the
C.f(); // -> C                (static) hierarchy upwards

A a = new C();
a.f();      // -> A, decided on the static type of "a"
((C)a).f(); // -> C, as the expr's static type is C now
```

New body: fields

- variables (both instance ∼s and static ∼s) can only be ***hidden***
  - ◇ both the old and new variable are present in the instance/class
  - ◇ which one is accessed is determined using static binding
  - ◇ in general, much better to avoid this altogether, and give the field a different name

```
class A           { char x='A'; char a()  {return x;} }
class B extends A { char x='B'; char b()  {return x;}
                                char bS() {return super.x;}}

A a = new A();
A ab = new B();
B b = new B();
System.out.printf("%c %c %c %c %c %c %c %c%n",
   a.x,ab.x,b.x,((A)b).x,a.a(),ab.a(),b.a(),b.b(),b.bS());
//   A    A   B        A     A       A     A     B      A
```

The `final` modifier on classes and methods

- `final` classes cannot have descendants

```
final class FinalClass { ... }

class C2 extends FinalClass { ... } // compile error
```

- `final` methods cannot be overridden

```
class BaseClass {
    final void f() { ... }
}

class C2 extends BaseClass {
    final void f() { /* new body */ } // compile error
}
```

Interfaces

- ***interface***: has at least two relevant meanings

1. interface of a class: the collection of services that it provides
   - in general, comprises its public methods
2. as a language construct: a reference type similar to classes
   - it can only contain method headers with public accessibility
     - ◇ these are only "promises", and do not contain bodies
   - … and more rarely, fields with `public static final` modifiers
   - … and some more possibilities (static methods, default methods)

```
interface IFace {
    public int f(int n);
           int f2(int n); // public even if not stated

    static final int DATE = 19700101;
}
```

Interfaces: implementation

- classes can ***implement*** one or more interfaces
  - ◇ they have to give a body to each promised method
  - ◇ multiple inheritance between interfaces and classes is allowed: there are no (possibly clashing) bodies

```
interface F1 { public double f(int n); }
interface F2 { public double f(int k); }
                                    // ↓↓↓

class Impl implements F1, F2 {
    public double f(int z) { return 2 * z + 1; }
                  // --- the parameter names may be changed
}

F1 f1  = new F1();   // error: interfaces cannot be instantiat
F1 f1b = new Impl(); // only the implementing class
System.out.println(f1b.f(9)); // methods in F1 are invokable
```

Interfaces: usage

- the below Impl2 ***implements*** the interface F

```
interface F             { public double f(int n); }
interface G extends F { public void   g(F f);   }

class Impl implements G {
    public double f(int z) { ... }
    public void   g(F f)   { System.out.println(f); }
}

class Impl2 extends Impl { }


// ---------- Impl -> G, so it has a method g()
   new Impl().g(new Impl2());
             // ----------- Impl2 -> Impl -> G -> F
             //             so it's fit as argument for g()
```

Interfaces: inheritance

- multiple inheritance is allowed between interfaces
  - ◇ no clash here either, usually (as there are no method bodies)

```
interface F1 { double f(int n); void g(); }
interface F2 { int    f();      void g(); }

interface F3 extends F1, F2 { public String f(String s); }

class Impl implements F3 {
    public double f(int n)    { ... }
    public int    f()         { ... }
    public String f(String s) { ... }
                  // ----------- the name "f" is overloaded,
                  //             all of them will have to be impl

    public void g() { ... } // only one implementing function
 // ------ the modifier must explicitly be stated here
}
```

Interfaces: default body

- interfaces usually don't change
  - ... but if they would, it would require derived classes to change
  - ... even if we only add methods to the interfaces
    - all implementing classes would have to give a body to the method
- solution in Java 8: the interface can define a default body

```
interface F {
    default double f(int n) { return 2 * n + 1; }
    void g();
}

class Impl implements F {
    // f does not have to be implemented here

    public void g() { ... }   // ... however, g must be
}
```

Interfaces: passing functions as arguments

- already mentioned: “functions” can be used as function arguments

```
IntUnaryOperator f = x -> c*x;
f.applyAsInt(123);
```

- `IntUnaryOperator` is an interface in the package `java.util.function`
- Java 8: ***functional interface***: an interface that has exactly one method
  - ◇ the method can have any number of arguments (zero, too)
  - ◇ the above syntax sugar can be used with such interfaces
- the reality: `x -> c*x` stands for an instance of a(n anonymous) subclass of `IntUnaryOperator`

```
class MyFun implements IntUnaryOperator {
    public int applyAsInt(int x) { return c*x; }
}

IntUnaryOperator f = new MyFun();
```

Interfaces: passing functions as arguments

- ***anonymous class***: creating a new class and immediately instantiating it
  - the new class can be derived of another class or an interface
  - the compiler generates names for the "anonymous" classes (`MyClass$1`) and functions (`lambda$methodName$0`)

```
class MyClass {
    int f(int par1, IntUnaryOperator parFun) {
        int c = 36;
        IntUnaryOperator op = new IntUnaryOperator() {
// indicates instantiation       ---                  --
// name of the base class/interface  ----------------
            public int applyAsInt(int x) {
                return c*x;
            }
        };
        return op.applyAsInt(parFun.applyAsInt(par1));
    }
}
```

Interfaces: passing functions as arguments

- problem: the enveloping function might have finished running when the anonymous function would run; how can its local variable (whose lifetime is over) be accessed?
  - ◇ Java 8: ***effectively final*** variable: a local variable whose value is not changed after initialisation
    - it could get the `final` modifier
    - anonymous functions may only use the (effectively) final variables of their environments
    - as the variable is `final`, its value can be conserved

```
int c = 36;
IntUnaryOperator f = x -> c*x;
// ++c;  // this would make "c" not "effectively final"
```

- trick: should we need a changeable variable, we can put it in a (single element) array

```
int[] c = { 36 };
IntUnaryOperator f = x -> { ++c[0]; return c[0]*x; };
```

Abstract classes

- a class whose methods are allowed not to have bodies
  - most of the time, interfaces are a better solution
  - as it is a class, it is single inheritance
  - as it is a class, it can have (non-static) fields and methods
  - non-abstract classes can have abstract subclasses

```
abstract class AbsCl {
    int data;

    abstract String f();  // error: abstract can't have a body

    void g(int n) { /* non-abstract methods are OK, too */ }
}

class MyCl extends AbsCl { ... }

AbsCl ac  = new AbsCl(); // compile error: can't instantiate
AbsCl ac2 = new MyCl();  // OK
```

Arrays

- arrays are objects in Java (reference types)
- the length of array objects cannot be changed
  - ◇ `ArrayLists` can be used if such changes are necessary
- arrays of primitive elements contain the elements directly

```
int[] ints1 = new int[]{ 1, 2, 3 };   // full array literal
int[] ints2 = { 1, 2, 3 };  // can be abbreviated sometimes

int[1] wrong1 = {1};     // [] indicates the array type here
int[]  wrong2 = new int[3]{ 1, 2, 3 };      // here as well
int[]  wrong3 = new int[];   // the instance needs the size

int t1[], t2[]; // several arrays can be declared together
                // separate declarations are easier to read
```

- if no initial values are given, the array will be filled with `0`, `0.0` or `false`

```
int[] ints1 = new int[5];
int[] ints1 = { 0, 0, 0, 0, 0 };  // fully expanded
```

Arrays

- object arrays contain references

```
String[] strs1 = new String[]{ "A", "B" };
String[] strs2 = { "A", "B" };
```

- ***aliasing*** or ***sharing***: an entity can be accessed through separate names

```
class X { public int x = 1; }

X[] xs = new X[2];
xs[1] = xs[0] = new X();
System.out.println(xs[0] == xs[1]);  // true

xs[0].x = 2;
System.out.println(xs[1].x);         // 2
```

- if not explicitly filled, the array contains `null` values

```
String[] strs1 = new String[3];
String[] strs2 = { null, null, null }; // expanded
```

Arrays of arrays

```
String[][] strs1  = new String[][]{ {"a"}, {"b"} };
String[][] strs1b = { {"a"}, {"b"} };
String[][] strs2  = new String[2][1]; // { {null}, {null} }
String[][] strs3  = new String[1][2]; // { {null, null} }
String[][] strs4  = new String[1][];  // { null }

String[][] wrong = new String[][]; // no dimension is known
```

- the external and internal arrays of `int[][]` are of different nature

```
int[][] ints = { {1,2,3}, {4,5} };
              //  - - -     - -   contains values directly
              // -------  -----  contains references to arrays
```

- jagged arrays: the lengths of the internal arrays can differ

```
String[][] strs5 = new String[2][];
strs5[0] = new String[]{};
strs5[1] = new String[]{"A", "B"};
      // ------------ this bit is compulsory here
```

Wrapper classes

- primitive types are effective, but they might not be usable in some places
  - most data structures only take objects
    - arrays are exceptions
- ***wrapper class***: classes that correspond to primitive types

byte	short	int	long	char	boolean	float	double
Byte	Short	Integer	Long	Character	Boolean	Float	Double

Wrapper classes

- ***boxed*** value: can be accessed through a reference
  - ◇ wrappers are boxed versions of primitive types
- ***autoboxing*** and ***unboxing***: conversion between the primitive type and its wrapper does not have to be explicitly marked in the source code
  - ◇ boxing and using references incur extra costs ⇒ use it only if necessary

```
Integer i1 = 3;
Integer i2 = Integer.valueOf(3); // full meaning of the above

int     i3 = i1;
int     i4 = i1.intValue();      // full meaning of the above
```

- be careful: references can take `null` values, too

```
Integer i1 = null;
int     i2 = i1.intValue();        // NullPointerException
```

Wrapper classes

- operators on wrapper classes have extra costs as well
  - the operations are performed on the primitive types
  - the wrappers will be silently unboxed, then boxed again

```
Integer a = new Integer(3);
Integer b = new Integer(4);


Integer c = a + b;

// fully expanded:
Integer c = Integer.valueOf(a.intValue() + b.intValue());
```