Object Oriented Programming

1. Course Overview
2. Concepts and paradigms in OOP

Who, what

- Who, what
  - Marius.Joldos@cs.utcluj.ro – lectures, laboratory
  - Alex Bondor – laboratory
  - Ioana Jimborean – laboratory
  - Titus Giuroiu - laboratory
- Web pages
  - Moodle CMS (preferred): https://labacal.utcluj.ro
  - Old site: http://users.utcluj.ro/~jim/OOPE

Course aims

- Teach you a working knowledge of object-oriented software development, focussing on concepts, paradigms and attitudes for writing good object-oriented code
- Allow you to build on your basic programming and software engineering skills to learn to make effective use of an industrially relevant object-oriented programming language

Course Outcomes

- Knowledge / understanding
  - Foundations of OOP
  - Main elements in designing, programming, testing and documenting OO solutions
  - Design methods for relatively low complexity Java programs
  - Basic UML

Course Outcomes (contd)

- Intellectual skills
  - To understand a specification of a problem and realize this in terms of a computer program in the OOP paradigm.
  - To take a partial specification and make appropriate judgements on the functionality of the proposed system.
  - To develop a specification in UML from a natural language description

Course Outcomes (contd)

- Practical skills
  - Effectively use Java programming constructs
  - Develop programs of relatively low complexity in Java
  - Debug, test and document solutions using object orientation
  - Develop applets and relatively simple GUI components

Course Topics

- Concepts and paradigms in OOP. On to Java
- Control structures in Java.
- Classes and Objects. Arrays
- Packages. Inheritance and polymorphism.
- Java Interfaces. OO Application Development
- UML Object and Class Diagrams. Assertions.
- Testing. Debugging. Java Errors and Exceptions
- Java Collections. Generic Programming.
- Introduction to Java I/O
- Event handling in Java. Introduction to Java Graphics
- Graphical User Interfaces (I)
- Introduction to Threads
- Graphical User Interfaces (II)
- Review

Assessment. References

- Assessment
  - Written exam – in-class tests (Ic=0.1) and final (E=0.6), laboratory assignments evaluation and tests (Lt=0.4).
  - Final grade = 0.1Ic + 0.5E + 0.4Lt
- References
  - Bruce Eckel, *Thinking in Java*, 4th edition, Prentice Hall, 2006
  - David J. Barnes & Michael Kölling, *Objects First with Java. A Practical Introduction using BlueJ, 5*th Edition, Prentice Hall / Pearson Education, 2012
  - Paul & Harvey Deitel, *Java. How to Program,* 10th edition, Prentice Hall, 2012
  - Oracle Java Documentation
  - OracleJava Tutorials
  - UML introductory tutorials
  - BlueJ and Netbeans/Eclipse documentation

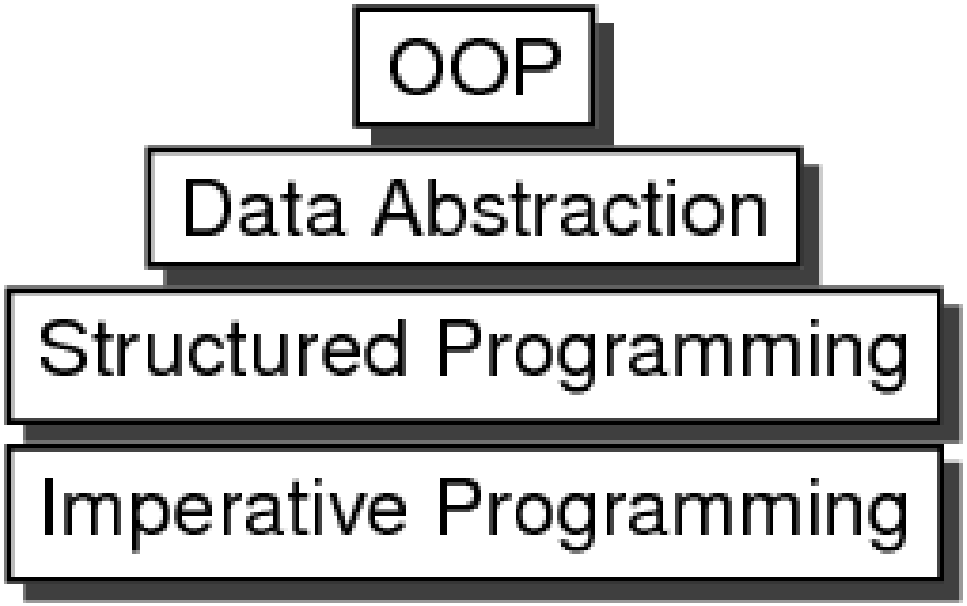
Concepts and paradigms in OOP

- Today's topics
  - Programming paradigms (imperative and structured programming)
  - Data abstraction
    - Abstract data types
  - The OOP paradigm
  - OOP concepts
    - Objects and classes
    - Encapsulation and inheritance

Programming Paradigms

- **Paradigm** (dict. def.): A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline.
- **Programming paradigm**
  - A model that describes the essence and structure of computation
  - Provides (and determines) the view that the programmer has of the execution of the program. Examples:
    - in object oriented programming, programmers can think of a program as a collection of interacting objects
    - in functional programming a program can be thought of as a sequence of stateless function evaluations.

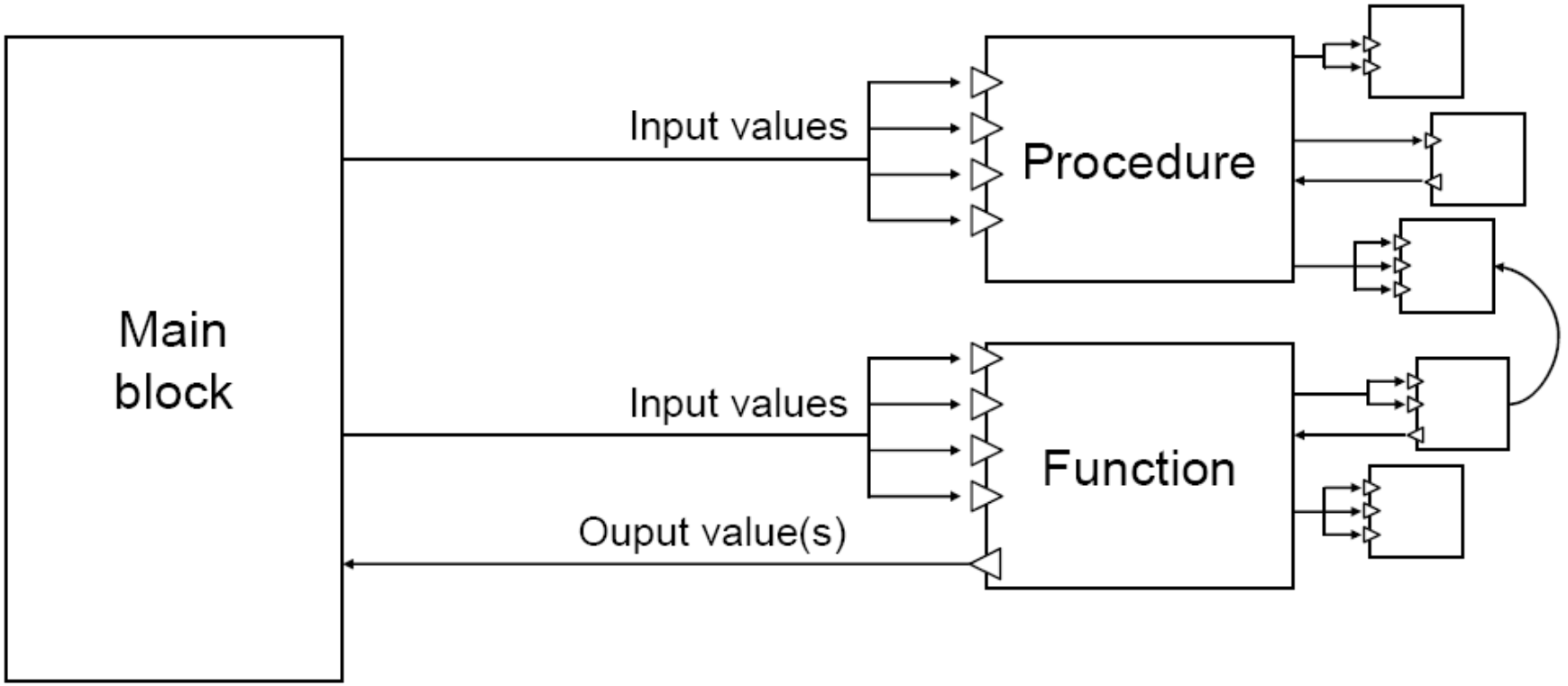
Programming Paradigms



Imperative Programming

- In the traditional von Neumann model, a computer consists of a central processing unit and storage, and performs a sequence of atomic instructions that access, operate on, and modify values stored in individually adressable storage locations. Here, **a computation is a series of arithmetic operations and side effects, such as assignments or data transfers that modify the state of the storage unit, an input stream, or an output stream.**
- We refer to this model as the ***imperative or procedural paradigm.***
- The importance of assignments, statements and variables as containers to the imperative paradigm is emphasized.
- Examples of programming languages: Fortran, Pascal, C, Ada.

Structured Programming



Structured Programming

- Operation abstraction
  - Structure of a module
    - Interface
      - Input data
      - Output data
      - Functionality description
    - Implementation
      - Local data
      - Sequence of instructions
  - Language syntax
    - Organization of code in blocks of instructions
      Definitions of functions and procedures
    - Extension of the language with new operations
      Calls to new functions and procedures

Structured Programming Benefits

- Eases software development
  - Avoids repetition of work
  - Programming work decomposed in independent modules
  - Top-down design: decomposition into subproblems
- Facilitates software maintenance
  - Easier to read code
  - Independence of modules
- Favors software reuse

Structured Programming. Example

```
int main()
{
   double u1, u2, m;
   u1 = 4;
   u2 = -2;
   m =sqrt (u1*u1 + u2*u2);
   printf("%lf\n", m);
   return 0;
}
```

```
double module(double u1, double u2)
{
   double m;
   m =sqrt (u1*u1 + u2*u2);
   return m;
}
int main()
{
   printf("%lf\n", module(4, -2));
   return 0;
}
```

Data Abstraction

- Data abstraction: enforcement of a clear separation between the *abstract properties* of a *data type* and the *concrete details* of its *implementation*
  - Abstract properties: those that are visible to client code that makes use of the data type - the *interface* to the data type
  - Concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time.

Abstract Data Types

- Data abstraction + operation abstraction
  - An Abstract Data Type consists of:
    - **Data structure** which stores information to represent a certain concept
    - **Functionality:** set of operations that can be applied to the data type
  - Language syntax
    - Modules are associated to data types
    - Not necessarily new syntax w.r.t. modular programming

Abstract Data Type Example in C

```
struct vector {
   double x;
   double y;
}
void construct (vector *v, double v1, double v2)
{
   v->x = v1;
   v->y = v2;
}
double module(vector v)
{
   double m;
   m = sqrt (v.x*v.x + v.y*v.y);
   return m;
}
```

```
int main()
{
   vector v;
   construct(&v, 4, 2);
   printf("%lf\n", module(v));
   return 0;
}
```

Abstract Data Type Extensibility

```
...

double product(vector v, vector w) {
   return v.x*w.x + v.y*w.y;
}
int main() {
   vector v;
   construct(&v, 4, 2);
   construct(&w, -1, 7);
   printf("%lf\n", product(v, w));
   return 0;
}
```

Abstract Data Type Benefits

- Domain concepts are reflected in the code
- *Encapsulation:* internal complexity, data and operation details are hidden
- Usage of data type is independent from its internal implementation
- Provides higher modularity
- Increases ease of maintenance and reuse of code

The Object-Oriented Paradigm

- The structured paradigm was successful initially (1975-85)
  - It started to fail with larger products (> 50,000 LOC)
- SP had postdelivery maintenance problems (today, 70 to 80 percent of total effort)
- Reason: Structured methods are
  - operation oriented (data flow analysis) or
  - attribute oriented (Jackson system development)
  - But not both

The Object-Oriented Paradigm

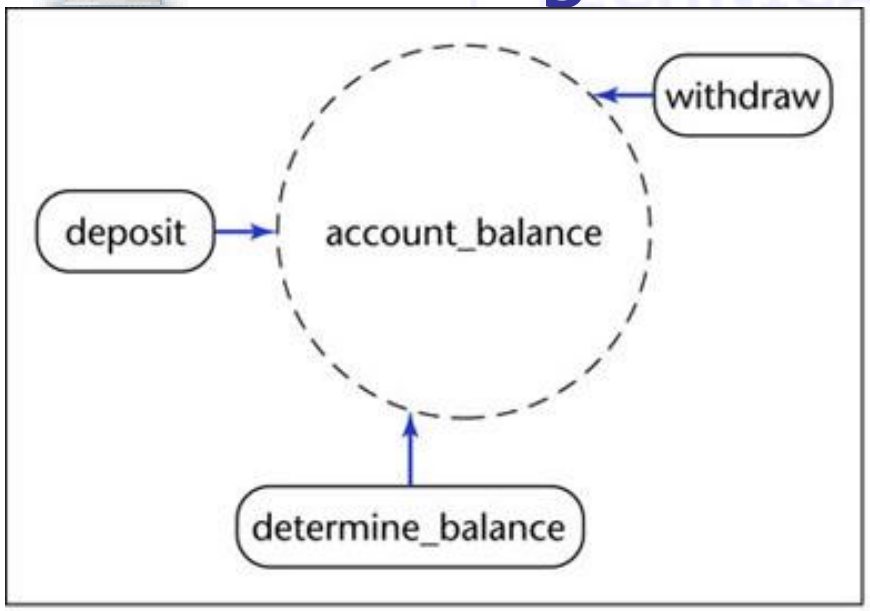
- The intended paradigm for the use of the language was a simulation of a problem domain system by abstracting behavior and state information from objects in the real world.

- The concepts of objects, classes, message passing, and inheritance is known as the object-oriented paradigm.

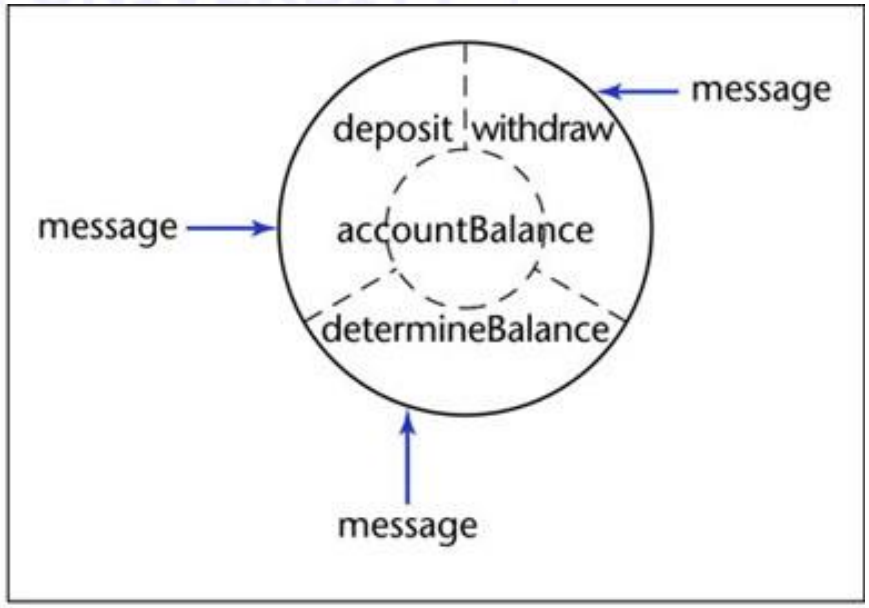
The Object-Oriented Paradigm

- OOP considers both the attributes and the operations to be equally important.
- A simplistic view of an object is:
  - A software component that incorporates both the attributes and the operations performed on the attributes that supports inheritance.
- Example:
  - Bank account
    - Data: account balance
    - Actions: deposit, withdraw, determine balance

Structured versus Object-Oriented Paradigm



(a)



(b)

- Information hiding
- Responsibility-driven design
- Impact on maintenance, development

Information Hiding

- In the object-oriented version
  - The solid line around `accountBalance` denotes that outside the object there is no knowledge of how `accountBalance` is implemented
- In the classical version
  - All the modules have details of the implementation of `account_balance`

Strengths of the Object-Oriented Paradigm

- With information hiding, postdelivery maintenance is safer
  - The chances of a regression fault are reduced (software does not repeat known mistakes)
- Development is easier
  - Objects generally have physical counterparts
    - This simplifies modeling (a key aspect of the object-oriented paradigm)

Strengths of the Object-Oriented Paradigm (contd)

- Well-designed objects are independent units
  - Everything that relates to the real-world object being modeled is in the object — *encapsulation*
  - Communication is by sending *messages*
  - This independence is enhanced by *responsibility-driven design*

Strengths of the Object-Oriented Paradigm (contd)

- A classical product conceptually consists of a single unit (although it is implemented as a set of modules)
  - The object-oriented paradigm reduces complexity because the product generally consists of independent units
- The object-oriented paradigm promotes reuse
  - Objects are independent entities

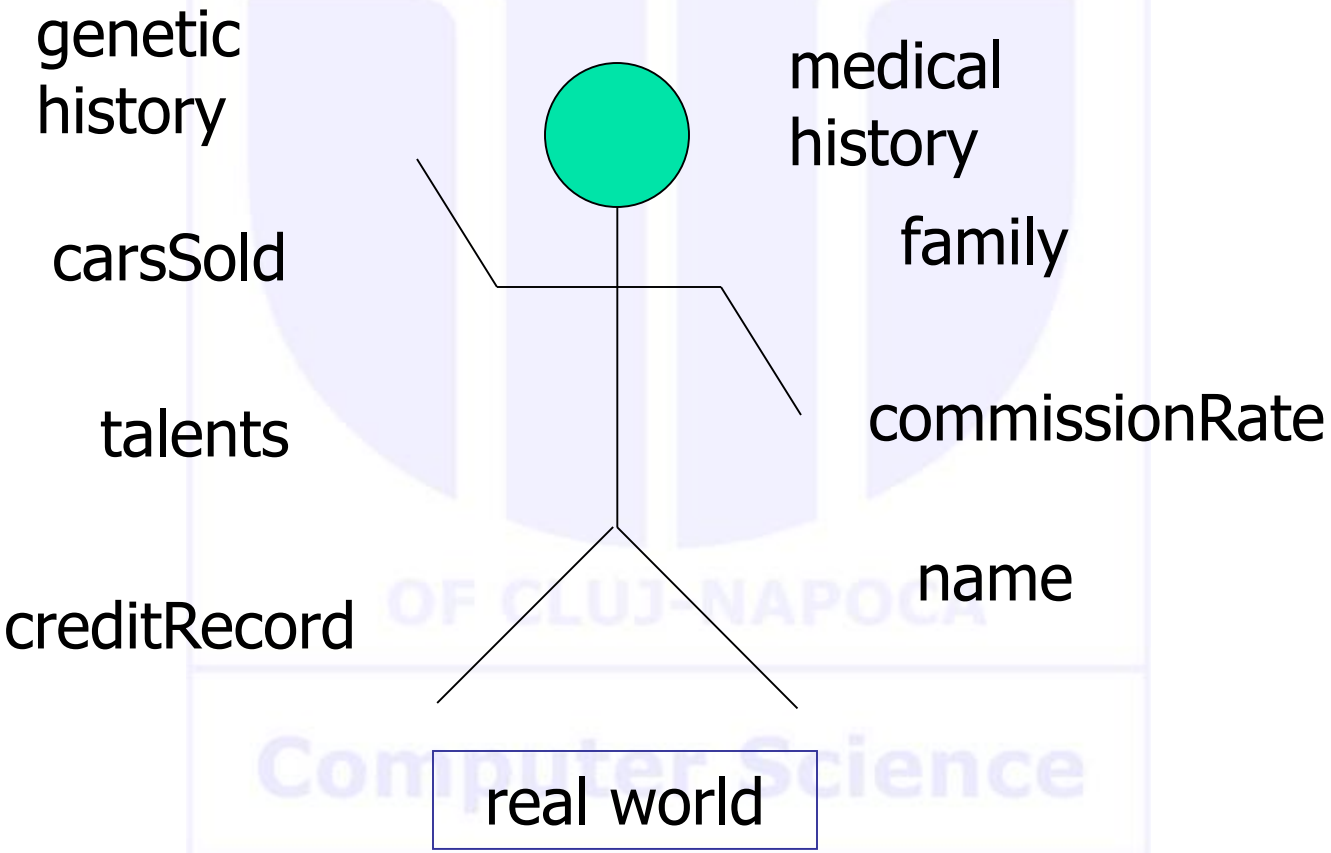
Object Oriented Programming

- Provides syntactic support for abstract data types
- Provides facilities associated to class hierarchies
- Change of point of view: programs are appendices of data
- A new concept appears: *object* = abstract data type with *state* (attributes) and *behaviour* (operations)

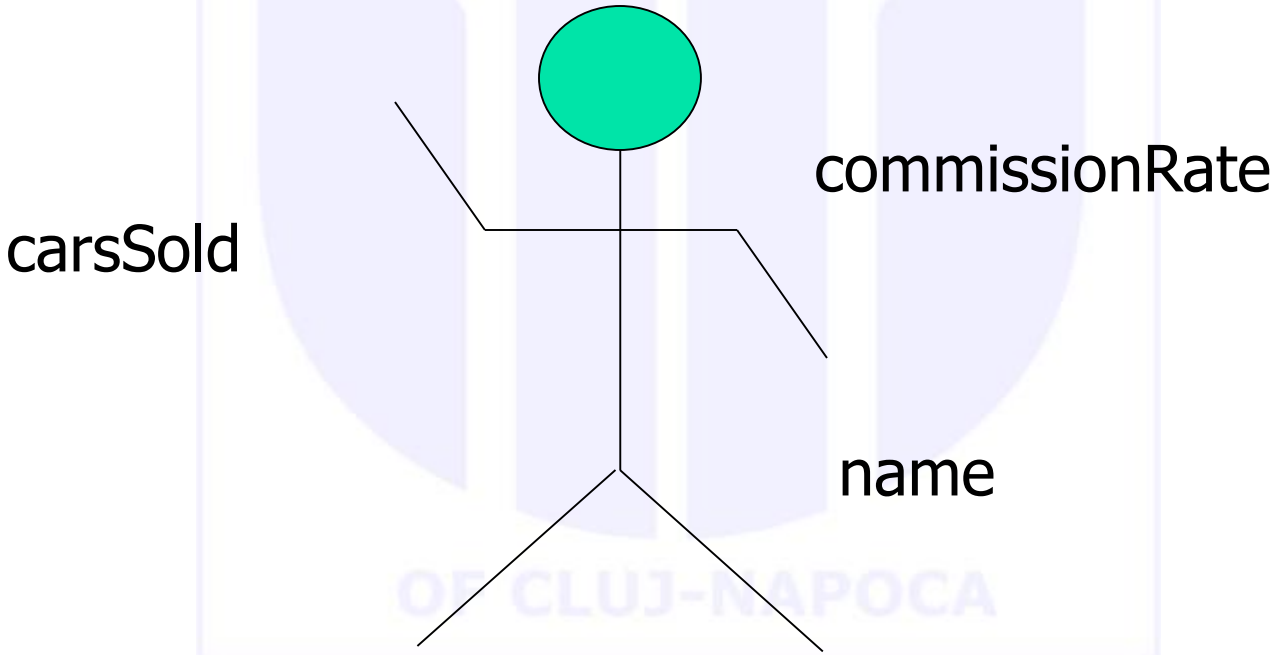
OOP Concepts

- We are only concerned with those data that are of interest to our problem (i.e. program).
- This process of filtering out unimportant details of the object so that only the important characteristics are left is known as *abstraction*.

Abstraction

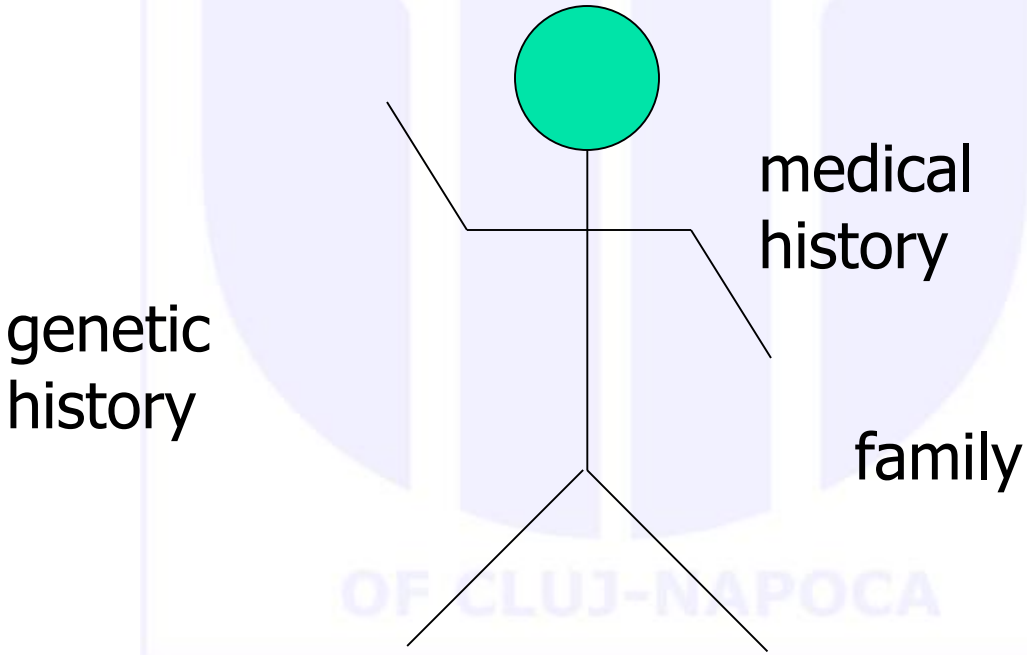


Abstraction



Abstraction of a SalesPerson for Sales Tracking System

Abstraction



Abstraction of a Patient in a Medical Database

What Are Software Objects?

- Building blocks of software systems
  - program is a collection of interacting objects
  - objects cooperate to complete a task
  - to do this, they communicate by sending "messages" to each other
- Objects model *tangible* things
  - person
  - bycicle
  - ...

What Are Software Objects?

- Objects model *conceptual things*
  - meeting
  - date
- Objects model *processes*
  - finding path through a maze
  - sorting a deck of cards
- Objects have
  - capabilities: what they can do, how they behave
  - properties: features that describe the objects

Object Capabilities: Actions

- Objects have *capabilities* (*behaviors*) that allow them to perform specific actions
  - objects are smart—they "know" how to do things
  - an object gets something done only if some other object tells it to use one of its capabilities
- Capabilities can be:
  - *constructors*: establish initial state of object's properties
  - commands: change object's properties
  - queries: provide answers based on object's properties

Object Capabilities: Actions

- Example: *trash cans* are capable of performing specific actions
  - constructor: be created
  - commands: add trash, empty yourself
  - queries: reply whether lid is open or closed, whether can is full or empty

Object properties: State

- *Properties* determine how an object acts
  - some properties may be constant, others variable
  - properties themselves are objects — also can receive messages
  - *trash can*'s lid and trash are objects
- Properties can be:
  - attributes: things that help describe an object
  - components: things that are "part of" an object
  - associations: things an object knows about, but are not parts of that object

Object properties: State

- *State*: collection of all object's properties; changes if any property changes
  - some don't change, e.g., steering wheel of car
  - others do, e.g., car's color
- Example: properties of *trash cans*
  - attributes: color, material, smell
  - components: lid, container, trash bag
  - associations: a *trash can* can be associated with the room it's in

Classes and Instances

- Our current conception: each object corresponds directly to a *particular* real-life object, e.g., a specific atom or automobile
- Disadvantage: much too impractical to work with objects this way
  - may be indefinitely many
  - do not want to describe each individual separately, because they may have much in common
- Classifying objects factors out commonality among sets of similar objects
  - lets us describe what is common once
  - then "stamp out" any number of copies later

Object Classes

- *Object class*
  - category of object
  - defines capabilities and properties common among a set of individual objects
    - all *trash can*s can open, close, empty their trash
  - defines template for making *object instances*
    - particular *trash can*s may have a metal casing, be blue, be a certain size, etc.

Object Classes

- Classes implement capabilities as *methods*
  - sequence of statements in Java
  - objects cooperate by sending messages to others
  - each message "invokes a method"
- Classes implement properties as *instance variables*
  - slot of memory allocated to the object that can hold potentially changeable value

Object Instances

- *Object instances* are individual objects
  - made from class template
  - one class may represent an indefinite number of object instances
  - making an object instance is called *instantiating* that object
- Shorthand:
  - class: object class
  - instance: object instance (not to be confused with instance variable)

Object Instances

- Different instances of, say, `TrashCan` class may have:
  - different color and position
  - different types of trash inside
- So their *instance variables* have different values
  - note: object instances contain instance variables — two different but related uses of the word *instance*
- Individual instances have individual identities
  - allows other objects to send messages to given object
  - each is unique, even though it has same capabilities
  - think of class of students in this course

Messages for Object Communication

- No instance is an island — must communicate with others to accomplish task
  - properties allow them to know about other objects
- Instances send messages to one another to invoke a capability (i.e., to execute a task)
  - method is code that implements message
  - we say "call a method" instead of "invoke capability"

Messages for Object Communication

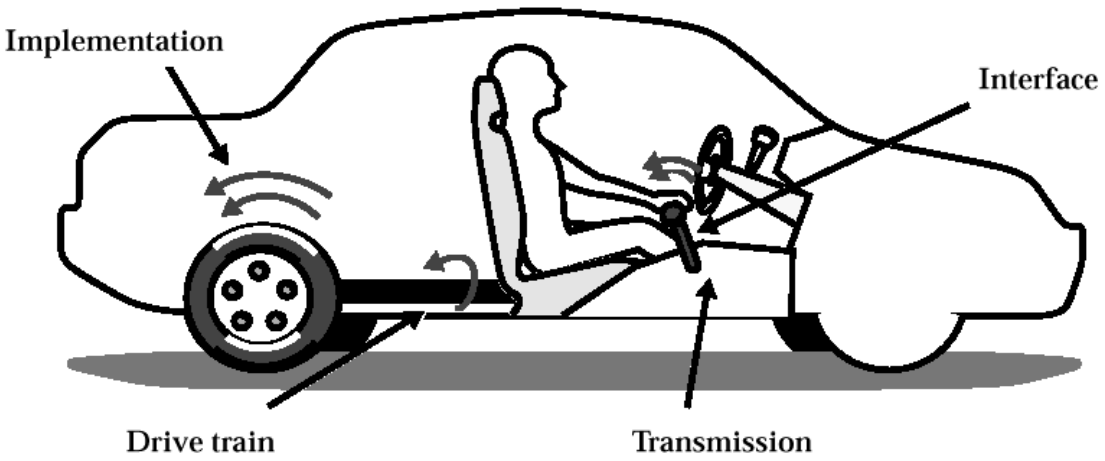
- Each message requires:
  - *sender*: object initiating action
  - *receiver*: instance whose method is being called
  - *message name*: name of method being called
  - optionally *parameters*: extra info needed by method to operate
    - more on this later
- Receiver can (but does not need to) send reply
  - we'll discuss *return types* in detail later

Encapsulation

- Car *encapsulates* lots of information
  - quite literally, under hood and behind dashboard
- So, you do not need to know how a car works just to use it
  - steering wheel and gear shift are the interface
  - engine, transmission, drive train, wheels, . . . , are the (hidden) implementation

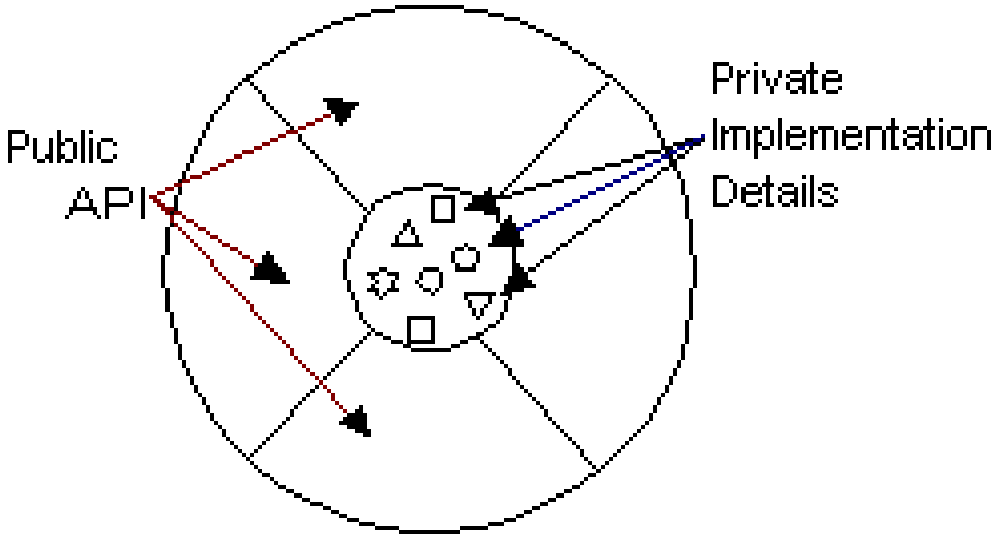
Encapsulation

- Likewise, you do not need to know how an object works to send messages to it
- But, you do need to know what messages it understands (i.e., what its capabilities are)
  - class of instance determines what messages can be sent to it



Encapsulation

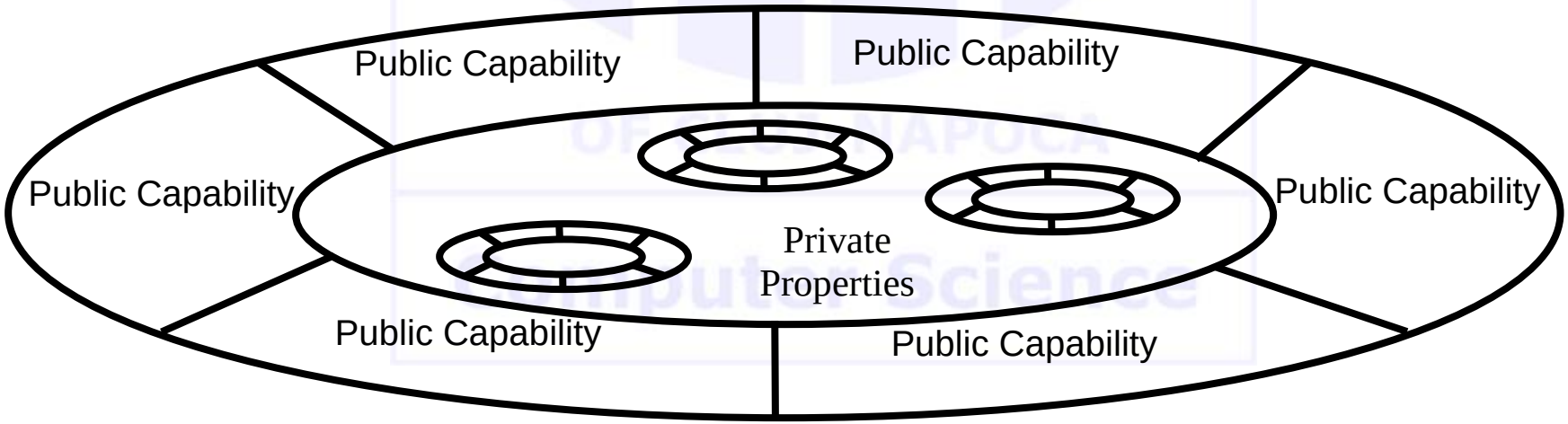
- Enclose data inside an object
  - Data can not be accessed directly from outside
- Provides data security



Views of a Class

- Objects separate *interface* from *implementation*
  - object is "black box"; hiding internal workings and parts
  - interface protects implementation from misuse

Note: private properties shown schematically as literally contained in an object. In reality, they are actually stored elsewhere in memory and referenced by their addresses

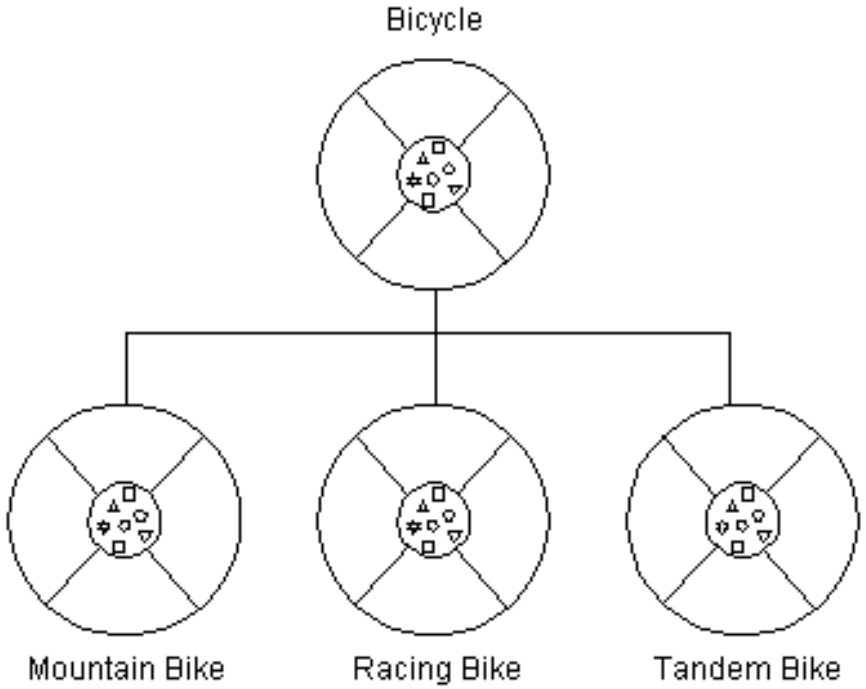


Views of a Class

- Interface: *public* view
  - allows instances to cooperate with one another without knowing too many details
  - like a contract: consists of list of capabilities and documentation for how they would be used
- Implementation: *private* view
  - properties that help capabilities complete their tasks

Inheritance

- A class *(SubClass)* can *Inherit* attributes and methods from another class (SuperClass)
- Subclasses provide specialized behavior
- Provides *Code Reuse*
- Avoids *Duplication* of Data



Polymorphism

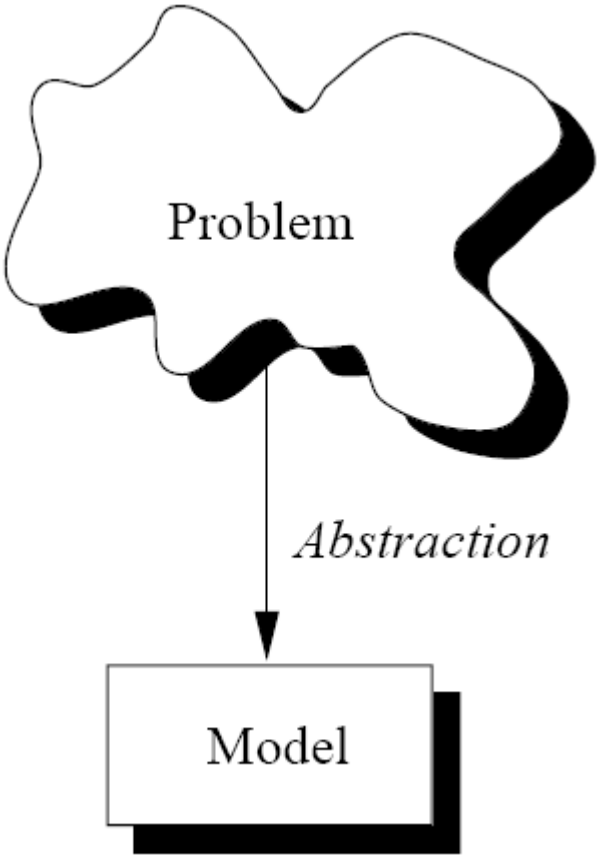
- The ability to take many forms
- The *same method* that is used in a superclass can be *overridden* in subclasses to give a *different functionality*
- E.g. Superclass 'Polygon' has a method, getArea

  getArea in Subclass 'Triangle' → a=x*y/2

  getArea in Subclass 'Rectangle' → a=x*y

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, or structure.

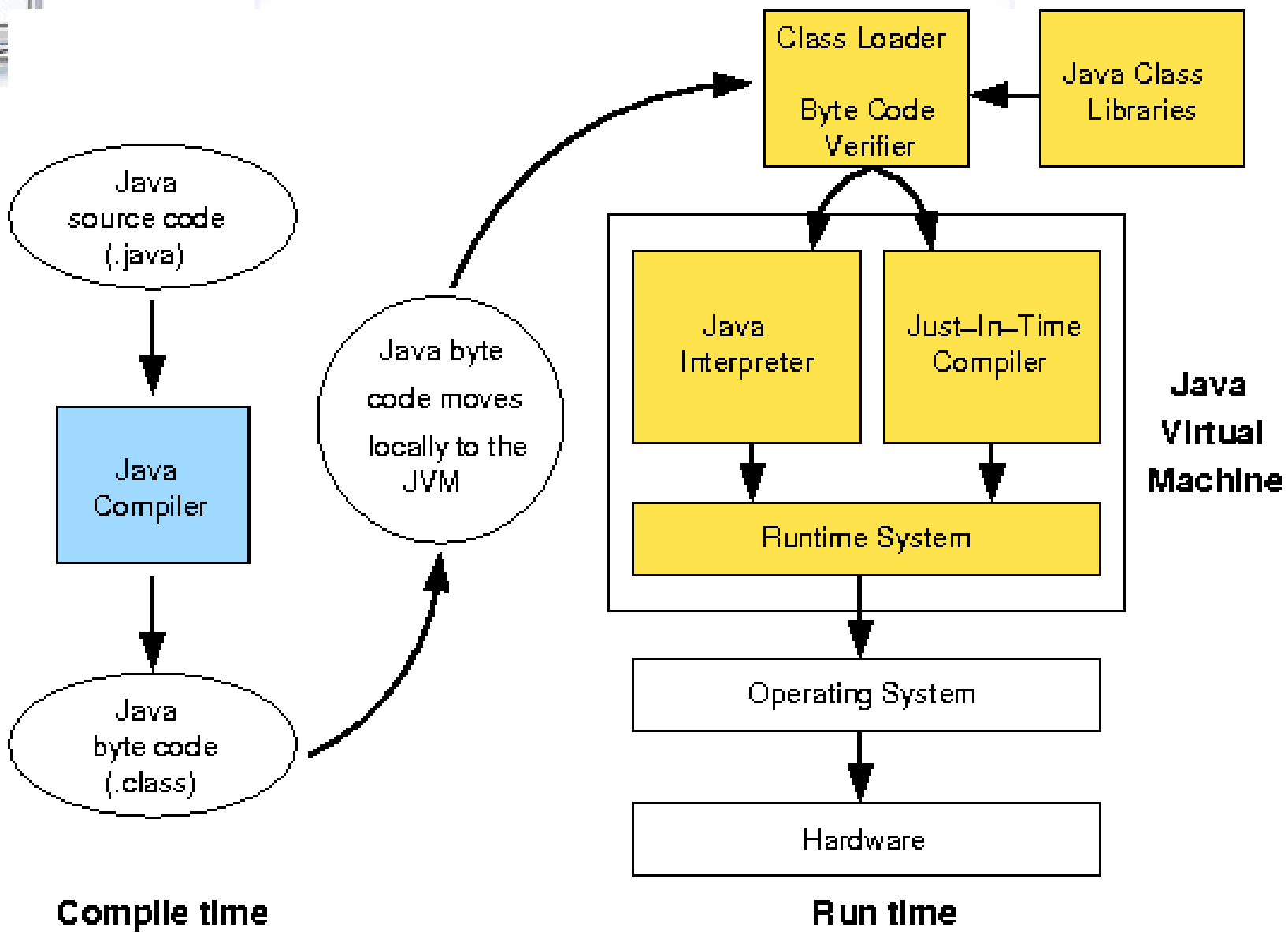


On to Java. Java Features

- James Gosling and Patrick Naughton (team leaders who led Java development), best described these unique features in their definition of the Java language as,

"A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture-neutral, portable, high performance, multithreaded, dynamic language".

The Java Environment

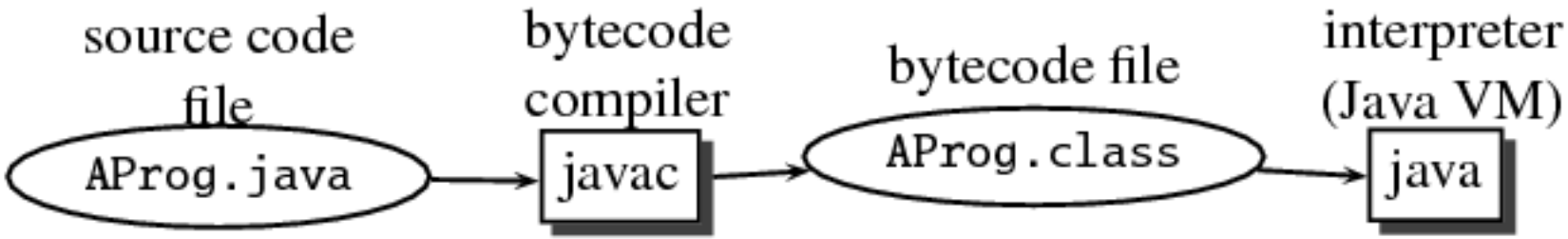


Running Java programs

- Java uses a two-step process
- Compiles program into *bytecodes*
  - bytecode is close to machine language instructions, but not quite — it is a generic "machine language"
  - does not correspond to any particular machine
- *Virtual Machine* (VM) interprets bytecodes into native machine language and runs it
  - different VM exists for different computers, since bytecode does not correspond to a real machine

Running Java programs

- *Same* Java bytecodes can be used on *different computers* without re-compiling source code
  - each VM interprets same bytecodes
  - allows you to run Java programs by getting just bytecodes from Web page
- This makes Java code run **cross-platform**
  - marketing says, "Write once, run anywhere!"
  - true for "pure Java", not for variants



Java Application Programs

- Types of Java programs:
  - *stand-alone applications* and *applets/servlets*
- A Java *application program* or "regular" Java program is a class with a method named **`main`**
  - When a Java application program is run, the *run-time system* automatically invokes the method named **`main`**
  - All Java *application* programs start with the **`main`** method
- You have already seen an application program in BlueJ examples

Applets

- A Java *applet* (*little Java application*) is a Java program that is meant to be run from a Web browser
  - Can be run from a location on the Internet
  - Can also be run with an applet viewer program for debugging
  - Applets always use a windowing interface
- In contrast, application programs may use a windowing interface or console (i.e., text) I/O
- You have already seen an applet in BlueJ examples

Class Loader

- Java programs are divided into smaller parts called *classes*
  - Each class definition is normally in a separate file and compiled separately
- *Class Loader*: A program that connects the byte-code of the classes needed to run a Java program
  - In other programming languages, the corresponding program is called a *linker*

Java keywords (aka Reserved words)

- Special words that can't be used for anything else
  - *abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while*
  - *null, true, false* – predefined like literals

Compiling a Java Program or Class

- Each class definition must be in a file whose name is the same as the class name followed by `.java`
  - The class `AProgram` must be in a file named `AProgram.java`
- Each class is compiled with the command `javac` followed by the name of the file in which the class resides

```
javac AProgram.java
```

  - The result is a byte-code program whose filename is the same as the class name followed by `.class`

```
AProgram.class
```

Running a Java Program

- A Java program can be given the *run command* (**`java`**) after all its classes have been compiled
  - Only run the class that contains the **`main`** method (the system will automatically load and run the other classes, if any)
  - The **`main`** method begins with the line:

```
public static void main(String[ ] args)
```

  - Follow the run command by the name of the class only (no **`.java`** or **`.class`** extension)

```
java AProgram
```

- Look at **`Hello.java`** in BlueJ examples for an extremely simple first application

Naming Conventions

- Start the names of variables, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters ("camelcase")

```
topSpeed    bankRate1    timeOfArrival
```

- Start the names of classes with an uppercase letter and, otherwise, adhere to the rules above

```
AProgram    MyClass    String
```

Variable Declarations

- Every variable in a Java program must be *declared* before it is used
  - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
  - The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
  - Variables are typically declared just before they are used or at the start of a block (indicated by an opening brace {)
  - Basic types in Java are called *primitive types*

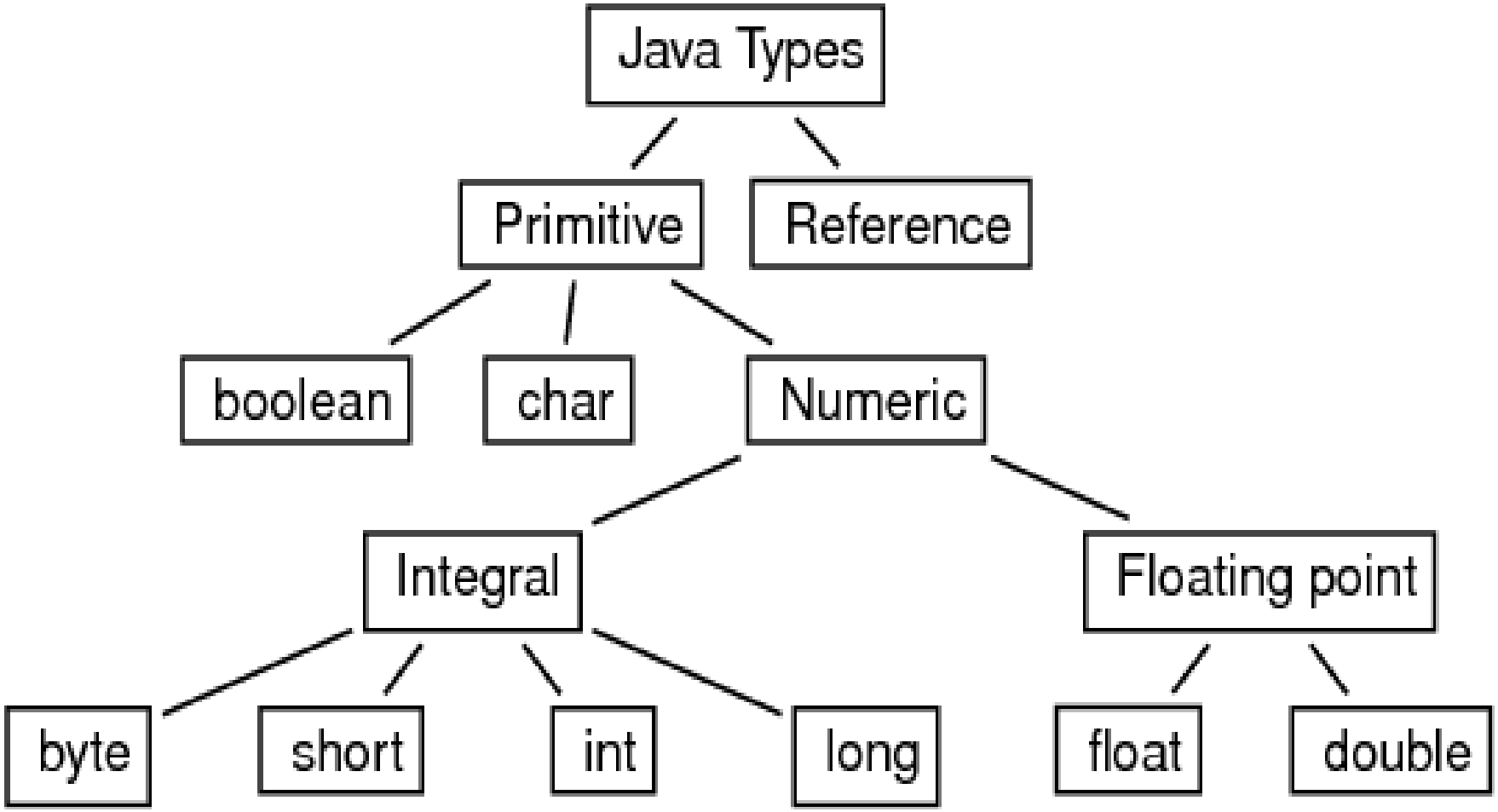
```
int numberOfBeans;
double oneWeight, totalWeight;
```

Access Modifiers on Variables/Methods

- **private**
  - Variable/method is private to the class.
  - *"Visible to my self".*
- **protected**
  - Variable/method can be seen by the class, all subclasses, and other classes in the same package.
  - *"Visible to the family"*
- **public**
  - Variable/method can be seen by all classes.
  - *"Visible to all".*
- Default access modifier, has no keyword.
  - **public** to other members of the same package.
  - **private** to anyone outside the package.
  - Also called *package access.*
  - *"Visible in the neighborhood"*


Java

Java Types


Java Types
Primitive
Reference
boolean
char
Numeric
Integral
Floating point
byte
short
int
long
float
double

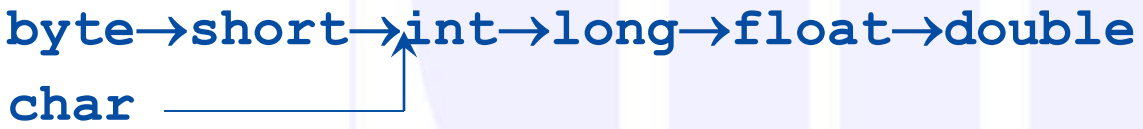
Primitive Types

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode 2^{16} - 1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	—	—	—	Void

All numeric types are signed, so don't go looking for unsigned types.

Assignment Compatibility

- More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it



  - Note that as your move down the list from left to right, the range of allowed values for the types becomes larger
- An explicit *type cast* is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., `double` to `int`)
- Note that in Java an `int` cannot be assigned to a variable of type `boolean`, nor can a `boolean` be assigned to a variable of type `int`

Arithmetic Operators and Expressions

- As in most languages, *expressions* can be formed in Java using variables, constants, and arithmetic operators
  - These operators are + (addition), – (subtraction), * (multiplication), / (division), and % (modulo, remainder)
  - An expression can be used anyplace it is legal to use a value of the type produced by the expression

Arithmetic Operators and Expressions

- If an arithmetic operator is combined with `int` operands, then the resulting type is `int`
- If an arithmetic operator is combined with one or two `double` operands, then the resulting type is `double`
- If different types are combined in an expression, then the resulting type is the right-most type on the following list that is found within the expression

  `byte`→`short`→`int`→`long`→`float`→`double`

  `char` ——↗

  - Exception: If the type produced should be `byte` or `short` (according to the rules above), then the type produced will actually be an `int`

Precedence and Associativity Rules

- When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence

  `base + rate * hours` is evaluated as

  `base + (rate * hours)`

- When two operations have equal precedence, the order of operations is determined by *associativity* rules

Precedence and Associativity Rules

- Unary operators of equal precedence are grouped right-to-left

  `+-+rate` is evaluated as `+(-(+rate))`

- Binary operators of equal precedence are grouped left-to-right

  `base + rate + hours` is evaluated as

  `(base + rate) + hours`

- Exception: A string of assignment operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`

Pitfall: Round-Off Errors in Floating-Point Numbers

- Floating point numbers are only approximate quantities
  - Mathematically, the floating-point number 1.0/3.0 is equal to 0.3333333 . . .
  - A computer has a finite amount of storage space
    - It may store 1.0/3.0 as something like 0.3333333333, which is slightly smaller than one-third
  - Computers actually store numbers in binary notation, but the consequences are the same: floating-point numbers may lose accuracy

Integer and Floating-Point Division

- When one or both operands are a floating-point type, division results in a floating-point type

  `15.0/2` evaluates to `7.5`

- When both operands are integer types, division results in an integer type
  - Any fractional part is discarded
  - The number is not rounded

    `15/2` evaluates to `7`

- Be careful to make at least one of the operands a floating-point type if the fractional portion is needed

Type Casting

- A *type cast* takes a value of one type and produces a value of another type with an "equivalent" value
  - If `n` and `m` are integers to be divided, and the fractional portion of the result must be preserved, at least one of the two must be type cast to a floating-point type **before** the division operation is performed

```
double ans = n / (double)m;
```

  - Note that the desired type is placed inside parentheses immediately in front of the variable to be cast
  - Note also that the type and value of the variable to be cast does not change

Type Casting

- When type casting from a floating-point to an integer type, the number is truncated, not rounded
  - `(int)2.9` evaluates to `2`, not `3`
- When the value of an integer type is assigned to a variable of a floating-point type, Java performs an automatic type cast called a *type coercion*

```
double d = 5;
```

- In contrast, it is illegal to place a `double` value into an `int` variable without an explicit type cast

```
int i = 5.5; // Illegal
int i = (int)5.5 // Correct
```

Increment and Decrement Operators

- When either ++ or – – operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
  - If `n` is equal to `2`, then `2*(++n)` evaluates to `6`
- When either operator follows its variable, and is part of an expression, then the expression is evaluated using the original value of the variable, and only then is the variable value changed
  - If `n` is equal to `2`, then `2*(n++)` evaluates to `4`

Reading

- Deitel – chapters 1, 2
- Eckel – chapter 1

(see slide 8 for full names of books)

Concepts discussed

- Programming paradigms
  - structured
  - object oriented
- Abstraction
  - abstract data types
- Object
  - capabilities(behaviours): actions
    - constructor
    - commmands
    - queries
  - properties: state
    - attributes
    - components
    - associations
  - instances of a class
- Messages: used for object communication
- Encapsulation

Concepts discussed

- Class: template for object creation
  - object capabilities: implemented as methods
  - object properties: implemented as instance variables
  - interface – public view
  - implementation – private view
- Inheritance
  - superclass, subclass
- Polymorphism
- Java
  - environment (compiler, class loader, interpreter, JIT, runtime system)
  - naming conventions
  - declarations and access modifiers