# Review
# Introduction to Computer Science I
# CSE 1020

moodle.yorku.ca

## Early and late binding

| | | |
|---|---|---|
| early binding | compiler | javac |
| late binding | virtual machine | java |

## Early binding

When the compiler encounters the invocation

`r.m(`$a_1, \ldots, a_n$`)`

it performs early binding. It consists of the following three steps.

1. Determine the declared type of the object reference `r`: class `C`.
2. Find compatible methods `m` in class `C`.
3. Select the most specific compatible method `m(`$t_1, \ldots, t_n$`)` in class `C`.

The invocation `r.m(`$a_1, \ldots, a_n$`)` is bound to method `m(`$t_1, \ldots, t_n$`)` of class `C`.

## Late binding

Assume that the compiler binds the invocation

`r.m(`$a_1, \ldots, a_n$`)`

to method `m(`$t_1, \ldots, t_n$`)` of class `C`.

When the virtual machine encounters the invocation

`r.m(`$a_1, \ldots, a_n$`)`

it performs late binding. It consists of the following step.

- Determine the actual type of the object reference `r`: class `C'`.

The invocation `r.m(`$a_1, \ldots, a_n$`)` is bound to method `m(`$t_1, \ldots, t_n$`)` of class `C'`.

Note that the signature does not change.

```
CreditCard card;
Random random = new Random();
if (random.nextBoolean()) {
  card = new CreditCard(...);
} else {
  card = new RewardCard(...);
}
output.println(card.toString());
```

**Question**

What are the early and late bindings for the `toString` invocation?

## Big-O notation

Let $f : \mathbb{N} \to \mathbb{N}$.
Then $f \in O(1)$ if

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \geq M : f(n) \leq F \times 1.$$

**Question**

Let $f(n) = 7$ for all $n \in \mathbb{N}$. Prove that $f \in O(1)$.

## Big-O notation

Let $f : \mathbb{N} \to \mathbb{N}$.
Then $f \in O(n)$ if

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \geq M : f(n) \leq F \times n.$$

**Question**

Let $f(n) = 7n + 9$ for all $n \in \mathbb{N}$. Prove that $f \in O(n)$.

## Interface

- API: `public interface HasVolume`
- UML: interface name preceded by ≪interface≫

## Interface

An interface specifies methods, it does not provide an implementation for them.

A class C implements an interface I if C contains an implementation of each method specified in I.

A class can implement multiple interfaces.

## Class implements interface

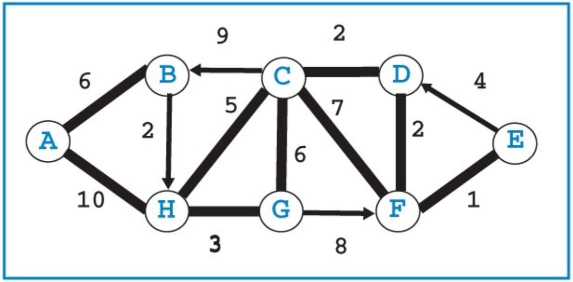
- API: `public class Cube implements HasVolume`
- UML: dashed arrow

## Check10D

A graph is a set of *vertices* (circles) and a set of *edges* (lines) connecting them. You can think of vertices as locations in a city and of edges as streets. An edge in a graph is either bidirectional (represented by a double line) or unidirectional (an arrow). In the figure, and using the city interpretation, we can go from vertex C to B but not from B to C because what connects them is a one-way street.

We associate a label (String) with each vertex and a weight (double) with each edge. You can think of the label as the name of the location and of the weight as the cost of traversing the edge. The cost could be fuel cost or time needed for the journey, given the road and traffic conditions. In this app we are going to assume that the label is made up of one character and that all weights are non-negative.

# Check10D



## Check10D

One way of representing a graph in memory is through a "map of maps" (also called a two-dimensional map). We imagine a map whose keys are the vertices of the graph. For the figure, this map would have eight elements. The value of the element with key $X$ is a map whose keys are the vertices that can be reached from $X$ by traversing a single edge and whose values are the weights of these edges.

## Check10D

There is the map of the graph in the figure.

```
{A={B=6,H=10},
 B={A=6,H=2},
 C={B=9,D=2,F=7,G=6,H=5},
 D={C=2,F=2},
 E={D=4,F=1},
 F={C=7,D=2,E=1},
 G={C=6,F=8,H=3},
 H={A=10,C=5,G=3}}
```

## Check10D

Given this map of maps, we can answer any question related to the graph; for example, how many vertices there are, how many (bi/unidirectional) edges there are, and whether there is a path from a given vertex to another. You can assume that the graph has already been represented and that the resulting graph has been serialized to a disk file.

A *path* in a graph is defined as any nonempty sequence of edges. In our case (where each vertex label is made up of one letter), we can represent the path by concatenating the labels of the vertices it traverses, that is, by a string. The *cost of the path* is defined as the sum of the costs of its edges. For example, ABHC is a path that takes us from A to C at a cost of $6 + 2 + 5 = 13$.

## Check10D

Note that not any string of letters constitutes a path. First of all, it must have at least two characters. Second, any consecutive vertices in it must correspond to an edge in the graph connecting the first vertex to the second. Based on this, we see that H is not a valid path, and neither are AZ, ABC, and ABF.

The app `Check10D` starts by prompting for and reading the name of the file in which graph map is stored. It then prompts for and reads a path in the graph. If the entered path is valid, the app must output its cost; otherwise it outputs `"Invalid path"` followed by a message indicating why, either `"(The path must have more than one vertex)"` or `"(Edge xx does not exist)"`, where `xx` is the first offending vertex pair.

## Check10D

```
Filename of the graph... L10D.ec
Enter the path... ABHCDFE
The cost of this path is: 18.0
```

## Check10D

```
Filename of the graph... L10D.ec
Enter the path... AHCBHG
The cost of this path is: 29.0
```

## Check10D

```
Filename of the graph... L10D.ec
Enter the path... ABCDF
Invalid path (Edge BC does not exist)
```

## Check10D

```
Filename of the graph... L10D.ec
Enter the path... AHD
Invalid path (Edge HD does not exist)
```

## Check10D

```
Filename of the graph... L10D.ec
Enter the path... G
Invalid path (The path must have more than one vertex)
```

## Check10D

```
Filename of the graph... L10D.ec
Enter the path... AHRF
Invalid path (Edge HR does not exist)
```

## Object Serialization

```
ObjectInputStream objectInput =
   new ObjectInputStream(
      new FileInputStream(name));
Map<String, URL> map =
   (Map) objectInput.readObject();
objectInput.close();
```