Classes and objects

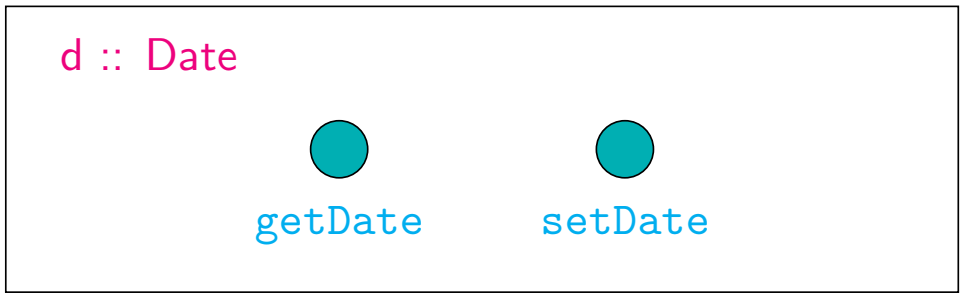
- A class is a template for a datatype
  - Instance variables or fields
  - Functions, or methods, to operate on data
- An object is an instance of a class
  - Private copy of instance variables
  - Methods implicitly attached to objects—e.g., `s.pop()`

Classes and objects

- A class is a template for a datatype
  - Instance variables or fields
  - Functions, or methods, to operate on data
- An object is an instance of a class
  - Private copy of instance variables
  - Methods implicitly attached to objects—e.g., `s.pop()`
- Ideally an object is a black box whose internals are manipulated through a well defined interface

Classes and objects

- A class is a template for a datatype
  - Instance variables or fields
  - Functions, or methods, to operate on data
- An object is an instance of a class
  - Private copy of instance variables
  - Methods implicitly attached to objects—e.g., `s.pop()`
- Ideally an object is a black box whose internals are manipulated through a well defined interface



Classes and objects . . .

- However, most OO languages do give direct access to internal data
  - Fields and methods can be `private` or `public`
- `static` fields and methods can be used without creating objects
- `final` means a value that cannot be modified

Constructors

- Can we initialize an object?
  Analogue of

```
int i = 10;
```

Constructors

- Can we initialize an object?
  Analogue of

```
int i = 10;
```

- Special methods called constructors
  - Invoked once, when an object is created
  - Usually have the same name as the class

```
class Date{
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;  month = m;  year = y;
  }
}
```

Constructors

- Can we initialize an object?
  Analogue of

```
int i = 10;
```

- Special methods called constructors
  - Invoked once, when an object is created
  - Usually have the same name as the class

```
class Date{
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;  month = m;  year = y;
  }
}
```

- `Date d = new Date(27,1,2009);`

Constructors . . .

- Can have more than one constructor

```
public Date(int d, int m){
  day = d;   month = m;  year = 2009;
}
```

Constructors . . .

- Can have more than one constructor

```
public Date(int d, int m){
  day = d;   month = m;  year = 2009;
}
```

- Invoke appropriate constructor by context
  - `Date d1 = new Date(27,1,2008);`
  - `Date d2 = new Date(27,1);`

Constructors . . .

- Can have more than one constructor

```
public Date(int d, int m){
  day = d;   month = m;  year = 2009;
}
```

- Invoke appropriate constructor by context
  - `Date d1 = new Date(27,1,2008);`
  - `Date d2 = new Date(27,1);`
- Two functions can have the same name, different signatures
- Overloading

Constructors . . .

- A later constructor can call an earlier one

```
class Date{
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;    month = m;    year = y;
  }

  public Date(int d, int m){
    this(d,m,2009);
  }
}
```

- `this` refers to the object to which method is associated
  - Objects have a notion of "self"!

Constructors . . .

- Can reverse the order

```
class Date{
  private int day, month, year;

  public Date(int d, int m){
    day = d;    month = m;    year = 2009;
  }

  public Date(int d, int m, int y){
    this(d,m);
    year = y;
  }
}
```

- Call to other constructor must be first instruction

Constructors . . .

- If no constructors are defined, default constructor initializes methods to default values
  - `Date d = new Date();`
  - Note the brackets after `Date`
- Default constructor is available only if no constructors are defined

Back to Java

- Java program : collection of classes
- Each class `xyz` in a separate file `xyz.java`
- To start the computation: one class must have a static method

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

  - `void` is the return type
  - `String[] args` refers to command line arguments

Back to Java

- Java program : collection of classes
- Each class `xyz` in a separate file `xyz.java`
- To start the computation: one class must have a static method

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

  - `void` is the return type
  - `String[] args` refers to command line arguments
- Java programs are usually interpreted on Java Virtual Machine

Back to Java

- Java program : collection of classes
- Each class `xyz` in a separate file `xyz.java`
- To start the computation: one class must have a static method

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

  - `void` is the return type
  - `String[] args` refers to command line arguments
- Java programs are usually interpreted on Java Virtual Machine
- `javac` compiles Java into bytecode for JVM
  - `javac xyz.java` creates "class" file `xyz.class`

Back to Java

- Java program : collection of classes
- Each class `xyz` in a separate file `xyz.java`
- To start the computation: one class must have a static method

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

  - `void` is the return type
  - `String[] args` refers to command line arguments
- Java programs are usually interpreted on Java Virtual Machine
- `javac` compiles Java into bytecode for JVM
  - `javac xyz.java` creates "class" file `xyz.class`
- `java xyz` interprets and runs bytecode in class file

Java syntax

Basic datatypes are similar to other programming languages

- `int` (4 bytes), `long` (8 bytes), `short` (2 bytes)
- `float` (4 bytes), `double` (8 bytes)
- `char` (2 bytes)
- `boolean`

Java syntax

Basic datatypes are similar to other programming languages

`int` (4 bytes), `long` (8 bytes), `short` (2 bytes)
`float` (4 bytes), `double` (8 bytes)
`char` (2 bytes)
`boolean`

- Size of `int` etc are fixed, operational semantics is wrt JVM

Java syntax

Basic datatypes are similar to other programming languages

`int` (4 bytes), `long` (8 bytes), `short` (2 bytes)
`float` (4 bytes), `double` (8 bytes)
`char` (2 bytes)
`boolean`

- Size of `int` etc are fixed, operational semantics is wrt JVM
- `char` is 2 bytes, Unicode, but otherwise behaves as usual

Java syntax

Basic datatypes are similar to other programming languages

`int` (4 bytes), `long` (8 bytes), `short` (2 bytes)
`float` (4 bytes), `double` (8 bytes)
`char` (2 bytes)
`boolean`

- Size of `int` etc are fixed, operational semantics is wrt JVM
- `char` is 2 bytes, Unicode, but otherwise behaves as usual

```
char c = 'a';
c = 'X';
if (c != '}') {...}
```

Java syntax

Basic datatypes are similar to other programming languages

`int` (4 bytes), `long` (8 bytes), `short` (2 bytes)
`float` (4 bytes), `double` (8 bytes)
`char` (2 bytes)
`boolean`

- Size of `int` etc are fixed, operational semantics is wrt JVM
- `char` is 2 bytes, Unicode, but otherwise behaves as usual

```
char c = 'a';
c = 'X';
if (c != '}') {...}
```

- Explicit `boolean` type

```
boolean b, c = false;
b = true;
b = (i == 7);
```

Java syntax . . .

- Expressions are similar to C
  - `x = 7` returns value `7`
  - `flag = (x = 0)` is caught as a syntax error
- Compound statements are familiar
  - `if (condition) ... else ...`
  - `while (condition) ...`
  - `do ... while (condition)`
  - `for (i = 0; i < n; i++) ...`
- No `goto`, but labelled `break` and `continue`

```
outer_loop:                    // this is a loop label
 for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
     if (a[i][j] == k){
        break outer_loop;  // exits the outer for loop
     }
   }
 }
```

Java syntax, strings

- `String` is a built in class
  - `String s,t;`
- String constants enclosed in double quotes
  - `String s = "Hello", t = "world";`
- Strings are not arrays of characters
  - Cannot write `s[3] = 'p'; s[4] = '!';`
- Instead, invoke method `substring` in class `String`
  - `s = s.substring(0,3) + "p!";`
- `+` is overloaded for string concatenation
- If we change a `String`, we get a new object
  - After the update, `s` points to a new `String`
- Java does automatic garbage collection

Java syntax, arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

- Can write `int a[]` instead of `int[] a`
- Can combine as `int[] a = new int[100];`
  - Aside: Why the seemingly redundant reference to `int` in `new`?
- Can create new arrays at run time
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!

Java arrays, . . .

```
public class arraycheck{
  public static void main(String[] argv){
    int[] a;
    int i, n;

    n = 10;
    a = new int[n];
    for (i = 0; i < n; i++){
        a[i] = i;
    }

    n = 20;
    a = new int[n];
    for (i = 0; i < n; i++){
        a[i] = -i;
    }
  }
}
```

Java syntax

```
class helloworld{
  public static void main(String[] args){
    System.out.println("Hello world!");
  }
}
```

- `args` is an array of `String`
  - `argv` in C
- Don't explicitly need the number of arguments (`argc` in C)
  - Use `args.length` to get this

More about `private` fields

- Should `private` fields be visible to other objects of the same type?
- How do we check if two objects are equal?

```
Date s,t;
..
if (s == t) { ... };
```

More about `private` fields

- Should `private` fields be visible to other objects of the same type?
- How do we check if two objects are equal?

```
Date s,t;
..
if (s == t) { ... };
```

- `==` checks whether `s` and `t` are the same object
  - `Date s = new Date(27,1,2009);` `Date t = s;` — `s == t` ✓
  - `Date s = new Date(27,1,2009);` `Date t = new Date(27,1,2009);` — `s == t` ×

More about `private` fields

- Should `private` fields be visible to other objects of the same type?
- How do we check if two objects are equal?

```
Date s,t;
..
if (s == t) { ... };
```

- `==` checks whether `s` and `t` are the same object
  - `Date s = new Date(27,1,2009);` `Date t = s;` — `s == t` ✓
  - `Date s = new Date(27,1,2009);` `Date t = new Date(27,1,2009);` — `s == t` ×
- We want to check if the contents of `s` and `t` are the same

More about `private` fields

- Add a function `isequal` to the class `Date`

```
class Date {
    private int day, month, year;

    public boolean isequal(Date d){
      return (this.day == d.day) &&
             (this.month == d.month) &&
             (this.year == d.year)
 }
```

More about `private` fields

- Add a function `isequal` to the class `Date`

```
class Date {
    private int day, month, year;

    public boolean isequal(Date d){
      return (this.day == d.day) &&
             (this.month == d.month) &&
             (this.year == d.year)
 }
```

- Invoke as `s.isequal(t)` (or `t.isequal(s)`)

More about `private` fields

- Add a function `isequal` to the class `Date`

```
class Date {
    private int day, month, year;

    public boolean isequal(Date d){
      return (this.day == d.day) &&
             (this.month == d.month) &&
             (this.year == d.year)
 }
```

- Invoke as `s.isequal(t)` (or `t.isequal(s)`)
- The object that executes `isequal` needs access to private information of the other object . . .

More about `private` fields

- Add a function `isequal` to the class `Date`

```
class Date {
    private int day, month, year;

    public boolean isequal(Date d){
      return (this.day == d.day) &&
             (this.month == d.month) &&
             (this.year == d.year)
 }
```

- Invoke as `s.isequal(t)` (or `t.isequal(s)`)
- The object that executes `isequal` needs access to private information of the other object . . .
- . . . but both are objects of the same type, so internal structure is not a secret!

More about `private` fields

- Add a function `isequal` to the class `Date`

```
class Date {
    private int day, month, year;

    public boolean isequal(Date d){
      return (this.day == d.day) &&
             (this.month == d.month) &&
             (this.year == d.year)
 }
```

- Invoke as `s.isequal(t)` (or `t.isequal(s)`)
- The object that executes `isequal` needs access to private information of the other object . . .
- . . . but both are objects of the same type, so internal structure is not a secret!
- Note the use of `this` to refer to the parent object
  - `this` can be omitted if context is clear

Subclasses

- A class `Employee` for employee data

```
class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  double bonus(float percent){
     return (percent/100.0)*salary;
  }
```

Subclasses

- Managers are special types of employees with extra features

```
class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.

Subclasses

- Managers are special types of employees with extra features

```
class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.
- `Manager` is a subclass of `Employee`
  - Think of subset

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
- Common to extend a parent class written by someone else

Subclasses

- Can use parent class's constructor using `super`

```
class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

Subclasses

- Can use parent class's constructor using `super`

```
class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

- In `Manager`

```
public Manager(String n, double s, String sn){
   super(n,s);   /* super calls
                    Employee constructor */
   secretary = sn;
}
```

Subclasses

- Subclass can override methods of super class

Subclasses

- Subclass can override methods of super class

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

Subclasses

- Subclass can override methods of super class

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

- In general, subclass has more features than parent class

Subclasses

- Subclass can override methods of super class

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

- In general, subclass has more features than parent class
- Can use a subclass in place of a superclass

```
Employee e = new Manager(...)
```

- Every `Manager` is an `Employee`, but not vice versa!
- Recall
  - `int[] a = new int[100];`
  - Aside: Why the seemingly redundant reference to `int` in `new`?
- One can now presumably write

```
Employee[] e = new Manager(...)[100]
```