Lecture 2, 8 January 2026
Classes, objects, Java

Madhavan Mukund, S P Suresh

Programming Language Concepts
January–April 2026

Abstract datatypes

- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`

Abstract datatypes

- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queue using sorted or unsorted lists, or using a heap

Abstract datatypes

- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queue using sorted or unsorted lists, or using a heap
  - Define ADTs without reference to implemention?

$$pop(push(s, v)) = v$$

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template
  - Each object maintains a separate copy of local data
  - Invoke methods on objects — send a message to the object

push (s, v)

s.push(v)

Example: 2D points, in Python

- A point has coordinates (x, y)
  - Each point object stores its own internal values `x` and `y` — instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x`, `self.y`

Example: 2D points, in Python

- A point has coordinates (x, y)
  - Each point object stores its own internal values `x` and `y` — instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x`, `self.y`
- When we create an object, we need to set it up
  - Implicitly call a constructor function with a fixed name
  - In Python, constructor is called `__init__()`
  - Parameters are used to set up internal values
  - In Python, the first parameter is always `self`

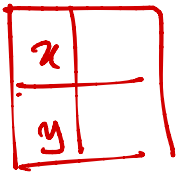
```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

int x ;

x = 5 ;

int x = 5;

p = new Point (4, 7);




```
class Point {
    float x, y;
    -- construct
```

Lists

```
l = []        l = new List()

l.append(5)

List.append(l, 5)
```

self

Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,dx,dy):
    self.x += dx
    self.y += dy
```

x = a
y = b

Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables
- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$
  - Does not update instance variables
  - state of object is unchanged

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,dx,dy):
    self.x += dx
    self.y += dy

  def odistance(self):
    import math
    d = math.sqrt(self.x*self.x +
                  self.y*self.y)
    return(d)
```

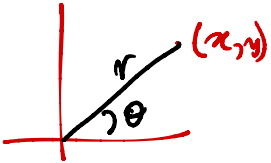
me

me, x

me y

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$




```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)


  def odistance(self):
    return(self.r)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r\cos\theta$, $y = r\sin\theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$

```
def translate(self,dx,dy):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += dx
  y += dy
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r \cos \theta$, $y = r \sin \theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not be aware whether representation is (x, y) or (r, θ)

```
def translate(self,dx,dy):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += dx
  y += dy
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

p.r
p.theta
p.x

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

  - Breaks the abstraction
  - Changing the internal implementation of `Point` can have impact on other code

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

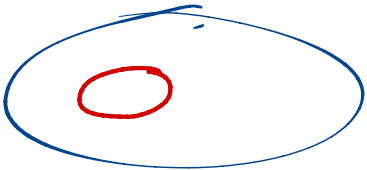
  - Breaks the abstraction
  - Changing the internal implementation of `Point` can have impact on other code
- Rely on programmer discipline

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables



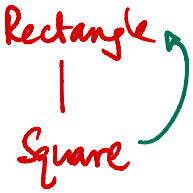
```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables
- The following is legal

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`




```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Can change the instance variable in `Square`
  - `self.side`

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping and inheritance . . .

- Can change the instance variable in `Square`
  - `self.side`
- The following gives a run-time error

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`
  - But `s.width` and `s.height` have not been defined!
  - Subtype is not forced to be an extension of the parent type

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables
- The following gives a run-time error

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` constructor sets `s.width` and `s.height`
  - But the instance variable names have changed!
  - Why should `Square` be affected by this?

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

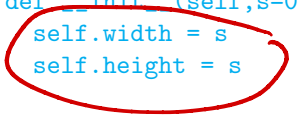
- Need a mechanism to hide private implementation details
  - Declare component private or public

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```



Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables
- Static type checking catches errors early

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Getting started with Java

The C Programming Language,
Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

Getting started with Java

The C Programming Language,
Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

Getting started with Java

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

Getting started with Java

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

- . . . and Java

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

Why so complicated?

- Let's unpack the syntax

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

Why so complicated?

- Let's unpack the syntax
- All code in Java lives within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility

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

Why so complicated?

- Let's unpack the syntax
- All code in Java lives within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility
- How does the program start?
  - Fix a function name that will be called by default
  - From C, the convention is to call this function `main()`

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

Why so complicated . . .

- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is `void`

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

Why so complicated . . .

- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is `void`
- Visibility
  - Function has be available to run from outside the class
  - Modifier `public`

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

Why so complicated . . .

- Availability
  - Functions defined inside classes are attached to objects
  - How can we create an object before starting?
  - Modifier `static` — function that exists independent of dynamic creation of objects

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

Why so complicated . . .

- The actual operation
  - `System` is a public class

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

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`

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

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`
  - `println()` is a method associated with streams
    - Prints argument with a newline, like Python `print()`

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

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`
  - `println()` is a method associated with streams
    - Prints argument with a newline, like Python `print()`

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

- Punctuation {, }, ; to delimit blocks, statements
  - Unlike layout and indentation in Python

Compiling and running Java code

- A Java program is a collection of classes

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

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`

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

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`

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

- Java programs are usually interpreted on Java Virtual Machine (JVM)
  - JVM provides a uniform execution environment across operating systems
  - Semantics of Java is defined in terms of JVM, OS-independent
  - "Write once, run anywhere"

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`

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

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`

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

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`

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

- Note:
  - `javac` requires file extension `.java`
  - `java` should not be provided file extension `.class`
  - `javac` automatically follows dependencies and compiles all classes required
    - Sufficient to trigger compilation for class containing `main()`

Scalar types

- In an object-oriented language, all data should be encapsulated as objects

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
`int`	4
`long`	8
`short`	2
`byte`	1
`float`	4
`double`	8
`char`	2
`boolean`	1

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
`int`	4
`long`	8
`short`	2
`byte`	1
`float`	4
`double`	8
`char`	2
`boolean`	1

- 2-byte `char` for Unicode

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

  - Double quotes denote strings

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

  - Double quotes denote strings

- Boolean constants are `true`, `false`

```
boolean b1, b2;

b1 = false;
b2 = true;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

- Can we declare a value to be a constant?

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow?
```

  - Note: Append `f` after number for `float`, else interpreted as `double`

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

- Can we declare a value to be a constant?

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow?
```

  - Note: Append `f` after number for `float`, else interpreted as `double`

- Modifier `final` indicates a constant

```
final float pi = 3.1415927f;

pi = 22/7;  // Flagged as error;
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;   // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n
- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++;   // Same as a = a+1
b--;   // Same as b = b-1
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;   // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n
- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++;   // Same as a = a+1
b--;   // Same as b = b-1
```

- Shortcut for updating a variable

```
int a = 0, b = 10;
a += 7;    // Same as a = a+7
b *= 12;   // Same as b = b*12
```

Strings

- `String` is a built in class

```
String s,t;
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

```
s[3] = 'p';
s[4] = '!';
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello 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!";`

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- + is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello 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!";`
- If we change a `String`, we get a new object
  - After the update, `s` points to a new `String`
  - Java does automatic garbage collection

Arrays

- Arrays are also objects

Arrays

- Arrays are also objects
- Typical declaration

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

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`

Arrays

- Arrays are also objects
- Typical declaration

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

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!

Arrays

- Arrays are also objects
- Typical declaration

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

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`

Arrays

- Arrays are also objects
- Typical declaration

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

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary

Arrays

- Arrays are also objects
- Typical declaration

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

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary
- Array constants: `{v1, v2, v3}`

Arrays

- Arrays are also objects
- Typical declaration

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

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary
- Array constants: `{v1, v2, v3}`
- For example

```
int[] a;
int n;

n = 10;
a = new int[n];

n = 20;
a = new int[n];

a = {2, 3, 5, 7, 11};
```