Numbers

Implicit This

Static Methods and Fields

Packages

Access Modifiers

Main

Overloading

Exceptions

Etc.
Integers

Java's number system:

• **byte** — an integer between $-128$ and $127$

• **short** — an integer between $-32768$ and $32767$

• **int** — an integer between $-2147483648$ and $2147483647$

• **long** — an integer between $-9223372036854775808$ and $9223372036854775807$

Each set of numbers forms a ring with its operations:

```java
Byte b = 127;
b + 1 → -128
b + 2 → -127
b + 2 - 3 → 126
```
Floating-Point Numbers

- **float** — an inexact real between $-3.4 \times 10^{38}$ and $3.4 \times 10^{38}$
  - Smallest non-zero number: $-1.4 \times 10^{-45}$
  - Digits of precision: about 6

- **double** — an inexact real between $-1.79 \times 10^{308}$ and $1.79 \times 10^{308}$
  - Smallest non-zero number: $4.94 \times 10^{-324}$
  - Digits of precision: about 15

Addition overflows to a special "infinity" value, and sometimes loses precision:

```c
double d = 5;
d + 0.000000000000000000000001 \rightarrow 5.0

d + 1e100 \rightarrow 1e100
```
Numbers as Objects

An `int` cannot be used as an `Object`

But Java provides pre-defined classes `Integer`, `Double`, etc.

```java
Object o = new Integer(5);
o.intValue() → 5
```

```java
List l = new Cons(new Integer(7), new Empty());
((Integer)l.nth(0)).intValue() → 7
```
Numbers

Implicit This

Static Methods and Fields

Packages

Access Modifiers

Main

Overloading

Exceptions

Etc.
Implicit This

In a method, `this` is added implicitly to the front of field uses and method calls that would be undefined otherwise.

```java
class Car {
    String make;
    Car(String make) {
        this.make = make;
    }
    boolean isSame(Car c) {
        return make.equals(c.make);
    }
    boolean isFord() {
        isSame(new Car("Ford"));
    }
}
```
Implicit This

In a method, \texttt{this}. is added implicitly to the front of field uses and method calls that would be undefined otherwise.

```java
class Car {
    String make;
    Car(String m) {
        make = m;
    }
    boolean isSame(Car c) {
        return make.equals(c.make);
    }
    boolean isFord() {
        isSame(new Car("Ford"));
    }
}
```
Numbers

Implicit This

Static Methods and Fields

Packages

Access Modifiers

Main

Overloading

Exceptions

Etc.
Static Methods

It's possible to define a function (as opposed to a method) by using the keyword `static`

```java
class Anything {
    static boolean biggerThanFive(int n) {
        return n > 5;
    }
}

Anything.biggerThanFive(10) → true
```

- Statics have to be in a class, but you don't have to use `new`
- The function name is prefixed by the class where it's declared
- You can't use `this` in a static method — there's no implicit argument
Static Fields

A static field is like a top-level definition

class Anything {
    static int n = 12;
}

17 + Anything.n \rightarrow 29
Anything.n = 15;
Final Fields

A `final` field is like a constant definition

```java
class Anything {
    final int n = 12;
}
```

$17 + Anything.n \rightarrow 29$

`Anything.n = 15;` not allowed
- Numbers
- Implicit This
- Static Methods and Fields
- Packages
- Access Modifiers
- Main
- Overloading
- Exceptions
- Etc.
Packages

Every class declaration resides in a `package`

- Roughly, the source files in the same directory are all in a package by default
- The `package` keyword declares the following class files to be in a particular package

```java
package org.plt.lists;

abstract class List {
    ...
}
```

...
Using Packages

To use classes/function from another package, you can prefix the name with the package name

```java
class Anything {
    static boolean biggerThanFive(int n) {
        return n > java.lang.Math.sqrt(25);
    }
}
```
Importing Packages

You can use `import` to have a prefix applied to any name that would otherwise be undefined.

```java
import java.lang;

class Anything {
    static boolean biggerThanFive(int n) {
        return n > Math.sqrt(25);
    }
}
```
Importing Packages

Actually, `java.lang` is always imported automatically

```java
class Anything {
    static boolean biggerThanFive(int n) {
        return n > Math.sqrt(25);
    }
}
```

`java.lang` is where `Object`, `Integer`, etc. come from
Numbers
Implicit This
Static Methods and Fields
Packages

Access Modifiers
Main
Overloading
Exceptions
Etc.
Access Control

By default, anything you declare can be seen by anything in the same package, and only by things in the same page.

For example, this library is probably useless, since no one outside the package can see `List`:

```java
package org.plt.lists;

abstract class List {
    ...
    ...
}
```

Public Access

The `public` modifier makes a declaration visible to everyone

```java
package org.plt.lists;

public abstract class List {
    ...
}

...
```
Modifiers like `public` must be used on methods and constructors, too, to make them visible.

```java
package org.plt.lists;

public abstract class List {
    public abstract int length();
    abstract boolean myHelper();
}

public class Empty extends List {
    public Empty() { }
    public int length() { return 0; }
    boolean myHelper() { ... }
}

...
Private Access

The `private` modifier restricts access to the current class.

```java
public class Cons extends List {
    private Object first;
    private List rest;
    public Cons(Object f, List r) {
        first = f; rest = r;
    }
    public Object getFirst() { return first; }
    ...
}

List l = new Cons(new Integer(7), ...);
((Cons)l).first not allowed
((Cons)l).getFirst() → Integer(val = 7)
```
Protected Access

The protected modifier allows access only within the class or subclasses
Numbers
Implicit This
Static Methods and Fields
Packages
Access Modifiers
Main
Overloading
Exceptions
Etc.
As you may have noticed, few people actually run Java programs in ProfessorJ...

A standalone program is started by calling the `Main` function of a designated class

```java
class MyProgram {
    public static void main(String[] cmdLineArgs) {
        ...
    }
}
```
Overloading

When you change the input part of a method's contract, you effectively change the name of the method:

```java
class Anything {
    int x;
    boolean y;
    Anything(int x, boolean y) { this.x = x; this.y = y; }
    boolean isSame(int v) { return v == this.x; }
    boolean isSame(boolean v) { return v == this.y; }
}
```

new Anything(1, false).isSame(1) → true
new Anything(1, false).isSame(false) → true
new Anything(1, false).isSame(17) → false
new Anything(1, false).isSame("hi") no such method
Overloading Can Be Tricky

class Car {
    String make;
    Car(String make) { this.make = make; }
    boolean isSame(Car c) { return make.equals(c.make); }
}
class Ford extends Car {
    String model;
    Ford(String model) { super("Ford"); this.model = model; }
    boolean isSame(Ford c) {
        return (make.equals(c.make) && model.equals(c.model));
    }
}

Car c1 = new Ford("Pinto");
Car c2 = new Ford("Fiesta");

c1.isSame(c2) → true

uses boolean isSame(Car c)
class Car {
    String make;
    Car(String make) { this.make = make; }
    boolean isSame(Car c) { return make.equals(c.make); }
}
class Ford extends Car {
    String model;
    Ford(String model) { super("Ford"); this.model = model; }
    boolean isSame(Ford c) {
        return (make.equals(c.make) && model.equals(c.model));
    }
}

Car c1 = new Ford("Pinto");
Car c2 = new Ford("Fiesta");

((Ford)c1).isSame(c2) \rightarrow true

uses boolean isSame(Car c)
class Car {
    String make;
    Car(String make) { this.make = make; }
    boolean isSame(Car c) { return make.equals(c.make); }
}
class Ford extends Car {
    String model;
    Ford(String model) { super("Ford"); this.model = model; }
    boolean isSame(Ford c) {
        return (make.equals(c.make) && model.equals(c.model));
    }
}

Car c1 = new Ford("Pinto");
Car c2 = new Ford("Fiesta");

c1.isSame((Ford)c2) \rightarrow \text{true} 

\text{uses} boolean isSame(Car c)
Overloading Can Be Tricky

class Car {
    String make;
    Car(String make) { this.make = make; }
    boolean isSame(Car c) { return make.equals(c.make); }
}
class Ford extends Car {
    String model;
    Ford(String model) { super("Ford"); this.model = model; }
    boolean isSame(Ford c) {
        return (make.equals(c.make) && model.equals(c.model));
    }
}

Car c1 = new Ford("Pinto");
Car c2 = new Ford("Fiesta");

((Ford)c1).isSame((Ford)c2) → false

uses boolean isSame(Ford c)
Numbers
Implicit This
Static Methods and Fields
Packages
Access Modifiers
Main
Overloading
Exceptions
Etc.
Exceptions

The **throw** form is similar to Scheme's **error**

When a method can raise an exception, the exception must be declared (usually)

```java
abstract class List {
    abstract Object nth(int n) throws NthException;
}

class Empty extends List {
    Object nth(int n) throws NthException {
        throw NthException("index too big");
    }
}

class Cons extends List {
    Object nth(int n) throws NthException {
        if (n == 0)
            return first;
        else
            return rest.nth(n - 1);
    }
}
```
Catching Exceptions

The `try ... catch` form is used to catch an exception and continue evaluation

```java
class Anything {
    static Object nthOrFalse(List l, int n) {
        try {
            return l.nth(n);
        } catch (NthException e) {
            return new Boolean(false);
        }
    }
}
```
Numbers
Implicit This
Static Methods and Fields
Packages
Access Modifiers
Main
Overloading
Exceptions
Etc.
A Few More Language Constructs

- **switch** — a shortcut for `if . . . else if . . .`
- threads — doing multiple things at a time
- characters — more than 255!
- implicit string conversion — "hello" + 1 → "hello1"
- ...

but not too many other things
Lots More Libraries

- GUIs (Swing)
- Container classes (lists, vectors, tables, etc.)
- Big numbers
- Stream I/O
- Cryptography
- Networking

... ... ... ...

many standard libraries, many more available