In Java, all types are a subset of the general Object.
Thus, we may write the following code:

```
List<String> ls;
List<Object> lo = ls;
lo.add(new Thread());
String s = ls.get();
// BOOM ~ type error; we assigned a thread to a string
```

The Java compiler can catch this, but when should it send the message?
The error is at line (2); implicit type conversion, which subverts the type mechanism.
Therefore, Java doesn't allow this, since List<String> !<= List<Object>.

This is counterintuitive; it feels like List<String> should be a subclass of List<Object>.
The problem is that lists are mutable; => our intuitions on subtypes don't extend to subclasses.

This introduces another issue: what if we want to print an Object List?

```
void printList(List<Object> lo)
{
  for (Object o: lo)
    System.out.println(o);
}
printList(ls); // ERROR ~ List<String> is missing this method
```

We could write a polymorphic function, but this would require us to write duplicate code.
Thus we use a **wildcard**.

```
void printList(List<?> l)
{
  for (Object o: lo)
    System.out.println(o);
}
printList(lo); // OK
printList(ls); // OK
```

Now suppose the method to be called is restricted in type:

```
void displayShapes(List<?> l)
{
  for (Shape s: l)
    displayShape(s);
}
displayShapes(List<String>);
// allowed by ? but String has no displayShape()
```

Thus we must use a **restricted wildcard**.

```
void displayShapes(List<? extends Shape> l)
{
  for (Shape s: l)
    displayShape(s);
}
/* ... */;
l.add(new Shape());
// may be disallowed since ? doesn't necessarily = Shape
```

Now suppose we want a function with two parameters

```
void convert(Object[] ao, List<Object> lo)
{
  for (Object o: ao)
    lo.add(o);
} // adds all of ao to lo
convert(String[], List<String>);
// ERROR ~ List<Object> != List<String>
```

Again, we could just duplicate code, but that is inefficient.
We also cannot just use a wildcard

```
void convert(Object[] ao, List<?> lo)
{
  for (Object o: ao)
    lo.add(o);
} // ERROR ~ casting ? to Object
```

We instead use a type parameter

```
<T> void convert(t[] ao, List<T> l)
{
  for (T o: ao)
    l.add(o);
}
String[] as = /* ... */;
List<Object> lo - /* ... */;
convert(as, lo);
// ERROR ~ List<Object> is not a supertype of List<string>
```

We can enforce this type rule with another bounded wildcard

```
<T> void convert(T[] a, List<? super T> l)
{
  for (T o: ao)
    l.add(o);
}
```

This complicated type algebra came naturally to meet demand;
   How is it implemented?
The Java language does not specify an implementation, but a common approach is:

- Every Object is a piece of storage with an address.
- The storage begins with a type 'tag'.
- This 'tag' points to a runtime representation of the type.

This, however, does not capture the notion of generic types;
   We don't have a separate representation for each implementation of the generic.
   Instead we point 'tag' to List<?> and trust the compiler to type check; this is called **erasure**.
This simplifies the system and lets us implement Java on simpler JVM's without generics.
This also lets us have only one copy of function code and static variables.

This is not perfect, however; consider the following usage

```
List<Shape> ls = /* ... */;
List<Rectangle> lr = (List<Rectangle>) ls;
// the standard compiler cannot do this check with the above information
```

As a result, we do not cast to generic types;
   instead we utilize Duck Typing

## Duck Typing

"If it waddles like a duck and quacks like a duck, it is a duck." Objects don't have types, only methods.
Instead of worrying about what an object 'is', we worry about what it 'does'.

For example, instead of this standard approach:

```
if (isaDuck(o))
{
  print "is a duck; it will waddle";
  o.waddle();
  o.quack();
}
```

We simply attempt the behavior with a contingency (Pseudocode):

```
try {
  o.waddle();
  o.quick();
} except {
  //deal with non-ducks
}
```

This approach is common in languages like Python, Ruby, Smalltalk, etc..

Is it possible to implement duck typing at compile time?
Yes! it was even one of the major motivations for Java:

### History of Java

In 1994, Sun Microsystems was working on Solaris

- kernel in machine code/apps in C/C++
- ran on SPARC workstations and multiprocessor servers

Sun knew C worked well on their complicated machinery, but what about IOT? They built a lab to address the following issues:

1. C/C++ crash easily, which is unacceptable.
2. SPARC is expensive, so code needs to be made for all machines.
3. Networking is VERY slow, so C/C++ updates would be long.
4. C++ is too complicated, you can't know all of the rules.

They considered two approaches:

1. Use C++, but remove the confusing stuff = C+-
2. Steal from XEROX PARC
    - invented the mouse, internet, IDE, Smalltalk (OO dynamic typing)
    - build workstations for text processing (but charged too much)

Smalltalk solved some problems of C, but introduced some new ones: The good:

- interpreted, so no crashing
- garbage collection rather than new/del, so more reliable
- operated via byte code, so machine independent
- Object-Oriented

The bad:

- weird syntax
- single-threaded
- terrible marketing

Solaris ultimately blended the two and called the result "Oak" The marketing department renamed it "Java" They decided to market it by writing a browser The main browser at the time was Mosaic bu UIUC

- crashed a lot (C++)
- required rewriting & recompiling to extend
- would ultimately diverge:
- netscape → mozilla → firefox
- internet explorer → microsoft edge

So Sun decided to create an extensible browser;

- it would accept Java bytecode and run it in a sandbox
- it was popular for demos but isn't used much anymore
- it convinced Eggert that Java was great for server-side applications

### Java Functionality

- compile-time checking — to avoid crashing like Smalltalk
- variables are always initialized — to avoid inconsistency like SmallTalk
- left-to-right order of evaluation — avoid crashing (*p++ / *p++ is undefined in C)
- well defined primitive types — consistency (a*b+c → fmul+fadd, not fmuladd)
- wrapped pointers — to avoid inconsistencies in behavior (because ptr length)

Let's consider how this last one works with the edge case of Arrays:

- are a reference type

- mandate subscript checking
- are all held on the heap (and thus return/passable)
- have fixed size when allocated
- are typed

Much of the rest of Java is analogous to C++, the exception being inheritance

**Inheritance**

Java has single inheritance, so the class hierarchy forms a tree.
Duck typers may be worried about this rigidity
eg. suppose you have a length() and a width() method;
A hierarchy cannot represent all versions of both!
We address this problem using Interfaces; they declare functions, but leave the class to implement them.

```
interface Lengthable
{
  int length();
}
interface Geometry2D extends Lengthable
{
  int width();
}
class Canvas implements Geometry2D
{
  int length() {return -3;}
  int width() {return 27;}
}
```

There is a hierarchy here, but

- a class can implement multiple interfaces.
- parents pass code, but interfaces pass "obligations"/API.
- interfaces form some type of a static version of duck typing.

Interfaces are often not powerful enough though;
if we use the same one many times, we may end up duplicating code.
We can solve this with abstract classes, which merge interfaces and classes

```
abstract class C
{
  int length() {return 1;}
  abstract int width();
}
new C(); // ERROR ~ width() is undefined!
abstract class D extends C
{
  int height() {return 3;}
}
new D(); // ERROR ~ width() is undefined!
class E extends D
{
  int width() {return 2;}
}
C x = new E();
foo(x);
```

Final classes are in some sense the opposite of abstract. Abstract Classes force you to subclass and define methods.
Final Classes don't allow you to subclass and define sub-methods.

A Final Class thus forms a leaf in the class hierarchy. Both classes and methods can be declared as final & thus can't be overwritten What are they good for?

1. Provides a barrier preventing subclasses from misbehaving
2. Low level reasons we won't discuss

A bioengineer wrote a simulator of the contents of the cell in Java.
It was a mess, unsurprisingly, since it was written by a bioengineer.
Everything was declared 'final'... Why? "because it made things faster!"
Finals make things faster! Why? It lets the compiler 'inline' the body of the method.

## Concurrency

Concurrency is one of the biggest advantages of Java.
To discuss it, consider the question: what is the world's fastest computer?
This will let us think ahead and plan such that our language lasts The world's fastest computer: SUMMIT:
CPU in C/assembly to squeeze out performance.
RedHat operating system (like SEASNet).
Most of the power goes toward cooling but still relatively power efficient.
Power cost of roughly a million laptops!
Speed is determined by LINPAC.
≡ Matrix multiplication floating point operations (FLOP) per second
This is the speed of 149e6 laptops!
Java was designed at the supercomputers of 1990!

The idea was to lash many processors/cores together.
There were two main philosophies for this type of parallelism:
SIMD — multiple instructions on one data set MIMD — multiple instruction pointers for multiple data sets (ex MIPS) The computer they looked at (POWER) does both:
it has Nvidia (GPU-like devices) & array instructions for small arrays (SIMD).
it has simple instructions for complicated structures (MIMD).

Java decided to focus on the latter:
The Java abstraction for MIMD is called the "Thread" Each Thread corresponds to one computation with its own IP The life cycle of the Thread:

1. Thread t = new Thread(); — NEW
2. t.start(); — RUNNABLE
   (a) compute — RUNNABLE
   (b) yield — RUNNABLE
   (c) IO — BLOCKED
   (d) wait — WAITING
   (e) sleep — TIMEDWAITING
   (f) return from run — TERMINATED

This leaves the Thread Object as the Threads "tombstone".

They are used in one of two ways

1. the classic object-oriented method — subclass Thread & override the run() method

```
class Foo extends Thread
{
  void run() {
    /* this code will be run when you start the thread */}
}
Foo f = new Foo;
f.start(); // allocates resources and calls t.run()
```

2. the preferred, duck typing method — define a new class implementing Runnable

```
// the interface is along the lines of:
interface Runnable
{
  void run();
  ....
}
class Bar implements Runnable
{
  void run() {/* ... */}
}
```

```
Bar b = /* ... */;
new Thread(b);
```

Threads have a major problem though, and that is the problem of Thread communication.
Java's solution is shared memory.
A Thread T can communicate with a Thread U by modifying a shared Object o

```
// T:
O.set(x);
// U:
...
u = get()
```

The main problem with this is the problem of race conditions.
get() may be called earlier or even concurrently with setting, causing a garbage get().
Race Conditions are a big enough problem that Java has multiple techniques.

1. Embarrassing Parallelism
   this is the simplest and easiest, but is SLOW

   ```
   Thread t = /* ... */;
   t.join(); // waits for t to finish
   ```

2. Volatile Instance Variables
   The 'volatile' keyword tells the compiler it is intended for communication.
   T he compiler then gets the variable on each usage, so access is not optimized away (some compilers place this constraint on themselves; this is very specialized & slow) Not only does this influence speed, but also correctness:

   ```
   y = f(x) + g(x); // x may change between function calls
   y = x - x; // optimization would make y zero
   while (x == x) ; // optimization would make this an infinite loop
   // volatile variables still leave room for errors, though:
   class C
   {
     volatile int x, y; // invariant: x < y
     int set(int newx) { x = newx; y = newx + 1;}
     int getx() {return x;}
     int gety() {return y;}
   }
   int a = getx();
   int b = gety();
   if (a >= b) disaster_strikes();
   // we may grab x before a change and y after
   ```

3. Synchronized Methods/Monitors
   Only one synchronized method may be used on an object at the same time.
   This slows you down, since Threads much wait to execute.
   Effectively places a lock on the object → methods must be VERY fast.
   This fixes many errors that volatile doesn't.

   ```
   class C
   {
     volatile int x, y;
     synchronized void set(int newx)
     {
       if (newx < newx + 1)
       {
         x = newx; y = newx + 1;
       }
     }
     int getx() {return x;}
     int gety() {return y;}
     synchronized long getBoth () {return ((long) x << 32) + y;}
   }
   long ab = getBoth();
   long a = ab >> 32;
   long b = ab & 0xFFFFFFFF;
   if (a >= b) disaster_strikes();
   ```

4. Waiting for events
   We use this when our method is too slow to synchronize

```
o.wait(); // remove all locks and wait until all are free again
o.notify(); // wake up one of the waiting Threads (FIFO is common)
o.notifyAll(); // wakes ALL the waiting threads
// can be used to allow programmer's choice of next Thread
```

   While Wait() does work in all addressed scenarios, it is too low level for many people.

**Exceptions**

These are error conditions that interrupt the regular flow of a program The most general syntax is much like the syntax of C:

```
System.out.print("1");
try {
  System.out.print("2");
  if (true) throw new Exception();
  // if(true) so no code is unreachable
  System.out.print("3");
} catch (Exception e) {
  System.out.print("4");
} finally {
  System.out.print("5");
}
System.out.print("6");
// 12456 -- finally is run even with an exception!
```

Java implements them in an interesting way:
   There is a class called Throwable which contains two subclasses:

- error — exceptions that cannot be caught
- exception — exceptions that can be caught

Exceptions derived from error or exception.runtimeException are unchecked.
Checked exceptions cannot be ignored at ANY LEVEL.
We can use this to implement our own exceptions:

```
public class OutOfGas extends Exception
{
  private int miles;
  public OutOfGas (String details, int m)
  {
    super (details); miles = m;
  }
  public int getMiles()
  {
    return miles;
  }
}
try {
  throw new OutOfGas ("You have run out of gas");
} catch (OutOfGas e)
{
  System.out.print(e.getMessage());
  System.out.printIn("Odometer: " + e.getMiles());
}
```

This can be done in two major ways, which are built into Java

1. <u>**Cyclic Barrier**</u>
   Threads run independently, but wait at a barrier point.
   Runnable object run() may be called every time the last Thread reaches the barrier.

```
// Example: Matrix Decomposition
class Solver
{
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable
  {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run()
    {
      while (!done())
      {
        processRow(myRow);
        try {
          barrier.await();
        } catch (InterruptedException ex) {
          return;
        } catch (BrokenBarrierException ex) {
          return;
        }
      }
    }
  }

  public Solver(float[][] matrix)
  {
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N, new Runnable() {
      public void run() {
        mergeRows(...);
      }
    });
    for (int i = 0; i < N; ++i)
      new Thread(new Worker(i)).start();

    waitUntilDone();
  }
}
```

2. **Exchanger**
   Works similarly to a "take a variable, leave a variable" drop box

```
// swap buffers between threads such that
// filling thread gets an empty one when full
class FillAndEmpty {
  Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
  DataBuffer initialEmptyBuffer = ... a made-up type
  DataBuffer initialFullBuffer = ...

  class FillingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialEmptyBuffer;
      try {
        while (currentBuffer != null) {
          addToBuffer(currentBuffer);
          if (currentBuffer.isFull())
          currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle ... }
    }
  }

  class EmptyingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialFullBuffer;
      try {
        while (currentBuffer != null) {
          takeFromBuffer(currentBuffer);
          if (currentBuffer.isEmpty())
          currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle ...}
    }
  }

  void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
  }
}
```