For example
Garbage collection

What is it ?

1. Unreachable objects cannot affect program execution,
2. Therefore, memory occupied by them can be safely reclaimed,
3. Reclamation process is called garbage collection

Why do it?

1. A running program has a finite amount of memory storage it can use
2. When memory is exhausted, program halts with OutOfMemory exception
3. Want to make most efficient use of memory

Reachable Object

An object is reachable if a reference to it is stored in a local or static variable or it is stored in a field or array element of a reachable object.

At a given point in time, the reachable objects:

1. Are those which can potentially be still used
2. Require space allocated in the heap
3. Cannot be deleted from the heap

Outline of algorithm

1. During execution, unreachable objects are mixed up
2. Must first identify unreachable objects, then we can reclaim them
3. Basic algorithm for this is called "mark and sweep"
4. Reachable objects are "marked" by traversing from object "roots"
5. Could use e.g. depth-first search for this
6. Roots are local variables and static variables
7. – Marked objects are "swept" to the left, unmarked objects are "swept" to the right
8. – Then can reclaim the unmarked objects

Java8

Default and static Interface Methods ?

Defa

1. Easy to add new methods to interface without changing other classes
2. Two interfaces can have the same default method, but different implementations
3. The common default method(s) must be overridden

Sta

1. Similar to default methods, except that we cannot override them in the implementation classes

- Interface

fields	constructors	privates	many
✗	✗	✗	✓

- Abstract class

fields	constructors	privates	many
✓	✓	✓	✗

2.

Lambdas ?

4. Anonymous single-method class can be unnecessarily long
5. Use Lambdas to make it more compact
6.

```
Collections.sort(ls, new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
}});
```

```
Collections.sort(ls, (s1,s2) -> s1.compareToIgnoreCase(s2));
```

Parameters **Method body**

7.

- A single expression `s -> s == ""`
- A statement block `s -> {return s == "";}`
- A void method with no brace `s -> System.out.println(s)`

8.

```
(customer, product) -> {
 if (customer.getAge() < 25 && product.hasAlcohol())
   return "Please show your ID!"
 return "Do you need a receipt?"
}
```

9. Anonymous single-method class can be unnecessarily long
10. Use Lambdas to make it more compact
11. – Can omit the data type of the parameters
12. – Can omit the parentheses If only one parameter

Functional Interfaces?

1. Interfaces with one and only one abstract method
2. Decorated with **@FunctionalInterface**
3. Can be represented as a lambda expression

Optional?

1. Optional<Soundcard> sc = Optional.empty();

2. Soundcard c = new SoundCar();
3. Optional<Soundcard> sc = Optional.of(c);

4. Soundcard c = ...;
5. Optional<Soundcard> sc = Optional.ofNullable(c);

6. Optional<Soundcard> sc =..
   Sc.isPresent()
   Sc.get();

1. •T orElse(T default)
   – Return the default value if the Optional is empty
2. Optional<Soundcard> maybeSoundcard = ...;
3. Soundcard soundcard = maybeSoundcard.orElse(new Soundcard("default"));

```
private Optional<Date> birth; // before could be null,
                              // now can be Optional.empty()

private String fullName;// mandatory

public PersonInfo(long id,String fullName, Optional<Date> birth)
   assert fullName!=null; // still needed
   assert birth!=null;    // birth==null would defeat the
                          // purpose of Optional
```

Stream?

What is stream?

1. • Rich library to query and process collections
2. list.stream()

```
List<Transaction> transactions = ...
List<Integer> res = transactions
.stream()
.filter(t -> t.value >= 80)
.sorted((t1,t2) -> t2.value-t1.value)
.map(t -> t.id)
.collect(Collectors.toList());
```

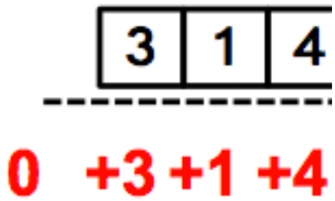
Stream Reduce?

```
List<Integer> myList = Arrays.asList(3, 1, 4);
int result = 0;
for (Integer element : myList)
 result = result + element;
return result;
```

```
List<Integer> myList = Arrays.asList(3, 1, 4);
int result = myList.stream()
      .reduce(0, (s1,s2) -> s1+s2);
```

Initial value of result

Accumulator for sum



Nested classes

Why nested class?

1. Increase logic
2. Increase encapsulation
3. More readable and maintainable code
4.

Non-static?

Not static – needs a parent object!

```
public static void main(String[] args) {
   Shape parent = new Shape();
   Shape.Square square = parent.new Square(1,1,2,3);
}
```

non- static Inner classes

a. have parent pointer
2. –For accessing fields /methods of enclosing class(parent)
3. Parent pointer automatically supplied for new inner class

• Static Inner Classes

1. have NO parent pointer!
2. Can NOT access fields/methods of enclosing class
3. Can construct without providing parent pointer
4. If no need to access enclosing info, then this is more convenient (and potentially more efficient).

Non-static inner class vs Static inner class?

1. Inner Class can access members of enclosing class, static cannot
2. When being constructed, inner class needs to have a parent pointer, static does not need
3. Inner class cannot have static methods, static can

Local class?

Sometimes we want to define classes that are only needed locally

a. Can be defined in any block
   i. Method body – for loop
   ii. if clause
b. Can access members of enclosing class
c. Can access final local variables of enclosing block
d. Can access effectively final local variables
e. Cannot have static methods (same as Inner Classes)
f. Must be non-static, so cannot declare interfaces as local classes
g. Cannot have static member, unless it's final primitive (constant)

```
public void greetInEnglish(String name) {
  interface Greeting { public void greet(); }
  class EnglishGreeting implements Greeting {
    public static String prefix;
    public static final String HELLO = "Hello! ";
    public void greet() {
      System.out.println(HELLO + name);
}}}
```

```
public static void outMethod() {
  int number = 1;
  class Inner {
    public void printNumber() { System.ou
  }
  number = 2;
  Inner c = new Inner();
  c.printNumber();
}
public static void main(String[] args) {
  outMethod();
}}
```

Anonymous classes

a. Make code more concise
b. Declare and instantiate a class at the same time
c. Local class, but with no name

```
ArrayList<String> ls = new ArrayList<String>();
...
Collections.sort(ls, new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
}});
```

Reflection

What is the reflection?

• Reflection in Java

1. This represents class information
2. Each object associated with unique instance of Class
3. Can find out about an object by checking its Class field (o.getClass())

Why is useful?

1. Each object associated with unique instance of Class
2. Can find out about an object by checking its Class field (o.getClass())
3. Reflection gives access to metadata

B

```
Class<? extends String> c2 = s2.getClass();
```

Which can work?

A ✗ B ✓

getClass

```
public final Class<?> getClass()
```

Returns the runtime class of this `Object`. The returne

The actual result type is `Class<? extends |X|>`

a. .class syntax can be used for primitive types
b. getClass() cannot be used for primitive types
c. Integer.**class** = Integer.getClass()
d. Integer.**class** != int.**class**
e. Integer.**TYPE** = int.**class**
f. • Singular can access inherited members but not private
g. • Declared can access private members but not inherited

```
public boolean equals(Object o) {
  if(o != null && this.getClass().equals(o.getClass())) {
    LogicGate lg = (LogicGate) o;
```

Generics

Generic class?

```java
class Vec {
 private Object[] elems = new Object[16];
 private int end = 0;
 public void add(Object e) {
  if(end == elems.length) { … }
  elems[end] = e; end=end+1;
 }
 public Object get(int index) {
  if(index >= end) { throw … }
  else return elems[index];
}}
```

```java
class Vec<T> {
 private Object[] elems = new Object[16];
 private int end = 0;
 public void add(T e) {
  if(end == elems.length) { … }
  elems[end] = e; end=end+1;
 }
 public T get(int index) {
  if(index >= end) { throw … }
  else return (T) elems[index];
}}
```

```java
Vec v = new Vec();
v.add(new Cat());
Cat c = (Cat) v.get(0); // have to cast :-(
```

```java
Vec<Cat> v = new Vec<Cat>();
v.add(new Cat());
Cat c = v.get(0); // don't have to cast :-)
```

```java
class Pair {
 private Object first;
 private Object second;

 public Pair(Object f, Object s) {
  first = f; second = s;
 }
 public Object first() { return first; }
 public Object second() { return second; }
}
```

```java
class Pair<FIRST,SECOND> {
 private FIRST first;
 private SECOND second;

 public Pair(FIRST f, SECOND s) {
  first = f; second = s;
 }
 public FIRST first() { return first; }
 public SECOND second() { return second; }
}
```

No need to cast!

```java
Pair p1 = new Pair("Cat",1);
Pair p2 = new Pair(10,20);
String c = (String) p1.first();
Integer i = (Integer) p2.first();
```

```java
Pair<String,Integer> p1 = new Pair<String,Integer>("Cat",1);
Pair<Integer,Integer> p2 = new Pair<Integer,Integer>(10,20);
String c = p1.first();
Integer i = p2.first();
```

Extend

```
interface Shape { void draw(Graphics g); }

class Square implements Shape { … }
class Circle implements Shape { … }

class ShapeGroup implements Shape {
 private List<Shape> shapes = new ArrayList<Shape>();

 ...
```

Group of Square or Circle?

B

C

D

```
class ShapeGroup<T> implements Shape {
 private List<T> shapes = new ArrayList<T>();

 ...

 public void draw(Graphics g) {
  for(T s : shapes) {
   s.draw(g);
}}}
```

Are we sure T has a draw() method?

<T extends Shape>

<T extends Type> – Type is the name of class or interface

<T extends T1 & T2 ...>

An upperbound B for a generic type T indicates that any type instantiated for T must be a subtype of B.
That is, T can be the upperbound itself, or a subclass of the upperbound or a class which implements the upper bound (if it is an interface).

```
<T extends Colour> void writeAll(T[] in, ColourPipe<T> out) {
  for(T item : in) {
    out.write(item);
  }
}
```

Generic Methods

<type parameter> (return type) (method)

- <T> T get(List<T> list, int index) { ... }
- <T extends Comparable> void sort(List<T> list) { ... }

```
class PointCmp {
 Point min(Point p1, Point p2) {
  if(p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y)) {
      return p1;
  } else { return p2; }
}}

ColPoint c1 = new ColPoint();
ColPoint c2 = new ColPoint();
c1 = (ColPoint) min(c1,c2);
```

Needs cast on the return value!

```
 <T extends Point> T min(T p1, T p2) {
  if(p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y)) {
      return p1;
  } else { return p2; }
}}
```

why wildcard types?

MyClass<A> has NO relationship with MyClass<B>, no matter whether A and B are related or not

• Can we create relationships between generic classes

– MyClass<A> – MyClass<B> • when A and B are related?

inside inher not whole

Suppose the Java compiler allowed the above program to compile. What problem?

1. Variable pipeCol refers to a Pipe which **can only accept Colour objects**.
2. In contrast, variable pipeObj refers to a Pipe which can accept any kind of Object.
3. If the above were allowed, then pipeObj would refer to the same object as pipeCol.
4. Thus, on line 3, a String object would be written into a Pipe<Colour> object, which breaks its invariant.

What is wildcard ?

1. They are anonymous types
2. They are types, but we don't know which they are
3. This differs from a generic type T as
4. there are restrictions placed on what operations can be performed with a wildcard.
5. In particular, we cannot write anything to an instance of Pipe<?>.

The type Pipe<String> is a subtype of Pipe<?>, meaning we can write things like this:

```
Pipe<String> pipeStr = ...
Pipe<?> pipeUnknown = pipeStr;
```

there are restrictions placed on what operations can be performed with a wildcard
since we cannot write anything to such a pipe, we cannot write an object which would break the Pipe<String> invariant.

• E.g.List<?>couldbeaList<String>... • Or,List<?>couldbeaList<Integer>...

• Cup<?>: **subtype of all Cups**

```
void drink(Cup<?> c) {
 System.out.println("Drink a cup of " +
   c.content.toString();
}

Cup<Tea> c1 = new Cup<Tea>(new Tea());
Cup<Coffee> c2 = new Cup<Coffee>(new Coffee());

drink(c1);  } Both are OK
drink(c2);
```

```
void drink(Cup<? extends Drinkable> c) {
 c.content.drink();
 System.out.println("Drink a cup of " +
   c.content.toString();
}

Cup<Tea> c1 = new Cup<Tea>(new Tea());
Cup<Coffee> c2 = new Cup<Coffee>(new Coffee());

drink(c1);
drink(c2);
```

```
interface Drinkable { void drink(); }
class Tea implements Drinkable { … }
class Coffee implements Drinkable { … }
```

```
void foo(List<?> x) {
 x.set(0, x.get(0));
}
```

This is an Object

Cannot confirm what type of Object to set

```
void foo(List<?> x) {
 fooHelper(x);
}
// Helper method created so that the wildcard can be captured
// through type inference.
<T> void fooHelper(List<T> x) {
 x.set(0, x.get(0));
```

Here, "super" indicates a lower bound – i.e. Cannot be subtype of Point!

```
class Point { int x; int y; … }
class ColPoint extends Point { int colour; }

class PointGroup<T extends Point> {
 private List<T> points = …;

 public void write(List<? super Point> out) {
  out.addAll(points);
 }
}
```

	类内部	本包	子类	外部包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

Path Coverage

– Function Coverage: number of methods invoked / # methods
– Statement Coverage: number of statements executed / # statements
– Branch Coverage: number of branches where both true and false side tested / # branches
- A simple execution path is a path through the method which iterates each loop at most once.
An execution path a path through a method's CFG which corresponds to an execution of that method.

White-Box Testing

• Testing with complete knowledge of implementation

i. Test cases generated by looking at program code
ii. Aim to reach high-degree of code coverage
iii. Gives potentially biased approach
iv. Not robust to implementation changes

Black-Box Testing

• Testing with without knowledge of implementation

i. Test cases generated by specification
ii. Gives unbiased approach
iii. robust to implementation changes

```
@Test public void testHasBetween_3() {
assertTrue(new List(ITEMS).hasBetween(0,1));
}
@Test public void testHasBetween_4() {
assertFalse(new List(new int[0]).hasBetween(0,0)); }
```

→calcute coverage

Polymorphism/ inheritance / subtyping

Briefly, discuss why polymorphism in Java can result in an infinite number of execu- tion paths for a given method.

1. When a function accepts a parameter of a non-primitive type, there can potentially be an infinite number of subtypes for it
2. Each of these classes can override one or more methods in the original type, and provide their own different implementations.
3. Thus, to test our function, we would need to try every possible concrete subtype of the parameter — which is infeasible.

Static Type: –> x declared type of a variable

1. This limits the possible values for the variable gate to the subtypes of LogicGate
2. Only methods declared in LogicGate (or its superclasses) can be called on gate, even for subclasses with additional methods.

Dynamic Type: –> type of object referenced by variable， actual type at runtime

the actual method which is executed is determined by the dynamic type.

Abstract class?

i. Contain abstract methods
ii. May also contain concrete methods + fields
iii. Cannot be instantiated
iv. Similar to interfaces in particular since interfaces gained the ability to have default implementations

- **Abstract methods:**
  i. Have no implementation
  ii. Concrete subclasses must provide it

Final class

Final class cannot be extended

```
final class A{
}
```

Does constructors are not inherited

- Constructors are not inherited not
- Constructors use super in first line to forward construction to super -super class

If the programmer does not explicitly write the super call, this call is added by the compiler.

Why inheritance?

•Allows to create a hierarchy of classes/interfaces, and to model our problem domain.

Why Dynamic dispatch？

•Dynamic dispatch (overriding) ensures subclass can change behaviour as needed

Overloading: multiple methods with the same name, but different parameters type.

Overriding: redefinition of the same method in the same method in a subtype.

```
Override
public int hashCode() {

   int code = 11 + cash;
   if(location != null)
       code *=11 + location.hashCode(
   else
       code *=11;

   if(name != null)
       code *=11 + name.hashCode();
   else
       code *=11;

   if(portfolio != null)
       code *=11 + portfolio.hashCode
   else
       code *=11;

   if(token != null)
       code *=11 + token.hashCode();
   else
       code *=11;

   return code;

Override
ublic boolean equals(Object object) {

   if (object == null && this != null
       return false;

   Player other = (Player) object;
   if ((cash != other.cash) || (getCl
       return false;

   if ((location == null && (other.lo
           return false;
   } else if ((token != other.token)|
       return false;
   }

   return true;
```

```
public class ASearchNode implements Comparable<ASearchNode>
@Override
public int compareTo(ASearchNode otherNode){
    double costNode1 = this.GcostFromStart + this.HcosttoTarget;
    double costNode2 = otherNode.GcostFromStart + otherNode.HcosttoTarget;
    if(costNode1 > costNode2){
        return 1;
    }else if (costNode1 < costNode2){
        return -1;
    }else{
        return  0;
    }
}
```

```
List<String>res1=new ArrayList<>(Arrays.asL
res1.sort(new Comparator<String>(){
  public int compare(String s1,String s2){
    return s1.charAt(0)-s2.charAt(0);
  }
});
```

clone purpose?

Create a copy of object

```
LispExpr e1 = new LispInteger(1);
LispExpr e2 = e1.clone();
// e1 != e2
// but, e1.equals(e2) must hold and
// e1.getClass() == e2.getClass() must hold
```

Object.clone()?

provides default implementation

– Is protected so must be explicitly overridden

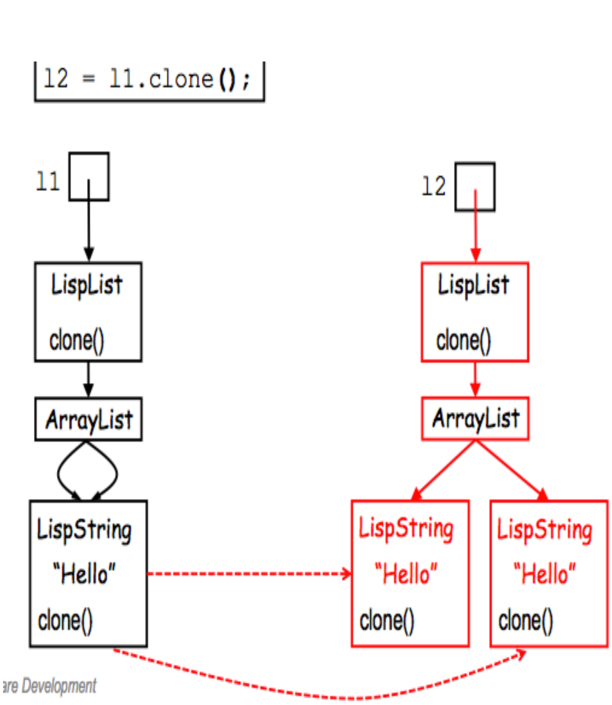
shallow clone

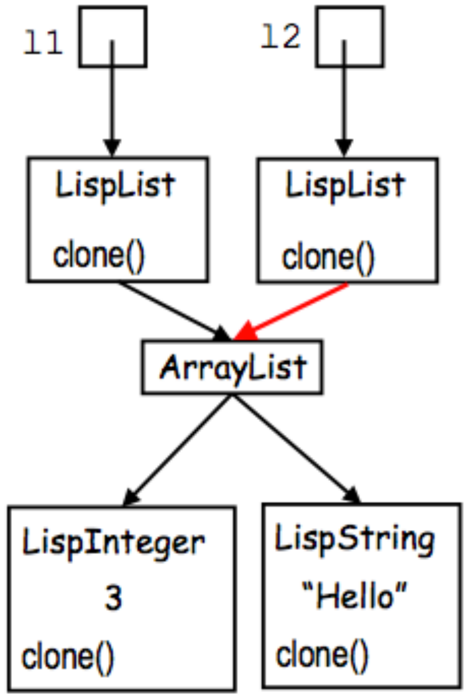
```
class LispList implements Cloneable
public Object clone() {
try { return super.clone(); }
catch(CloneNotSupportedException e) {
return null; // cannot get here
}

LispList l2 = (LispList) l1.clone();
```

Deep clone

This version of clone gives a deep copy: – (i.e. all children recursively cloned)

```
class LispList implements Cloneable
public LispList clone() {
LispList ne = new LispList();
for(LispExpr e : elements) {
ne.add(e.clone()); }
return ne;
}
```





shallow clone:

i. Cloned Object and original object are not 100% disjoint.
ii. Any changes made to cloned object will be reflected in original object or vice versa.
iii. Default version of clone method creates the shallow copy of an object.
iv. Shallow copy is preferred if an object has only primitive fields.

Deep clone:

1. Cloned Object and original object are 100% disjoint.
v. Any changes made to cloned object will not be reflected in original object or vice versa.
vi. To create the deep copy of an object, you have to override clone method.
vii. Deep copy is preferred if an object has references to other objects as fields.

Exception:

```
class ElementNotFound extends RuntimeException{
  public ElementNotFound(String message){super(message);}
}
```

Identify an alternative solution for question (d) and discuss its pros and cons.?

2. An alternative solution would be to return null instead of rethrowing the checked exception as an unchecked exception.

3. This is not really a good solution as it hides the exception which happened, and introduces the likelihood of an unexpected NullPointerException.

```
Public class UncheckedDBException extends RuntimeException{
        Public UncheckedDBException(String msg){
        super(m);
        }

        Public UncheckedDBException(String msg, Throwable cause){
        super(m,cause);
        }

        Public UncheckedDBException(Throwable cause){
        super(cause);
        }


}
```

Why exception is useful?

Exceptions allow problems to be dealt with

4. gracefully
5. in a client-specific manner

what is finnaly?

• Finally clause

   i. gets executed regardless of how try-block exited (e.g. normal execution, caught exception or uncaught exception)
   ii. useful for "cleaning up" allocated resources

why final is sensible?

One example is using finally to deallocate a resource which is allocated by a block of code, and needs to be deallocated under all circumstances

Unchecked Exceptions:

Subclasses of RuntimeException and Error

i. e.g., NullPointerException and IndexOutOfBoundsException
ii. do not require explicit declaration / catching

Checked Exceptions:
Subclasses of Exception,BUT not runtimeException

i. e.g., IOException
ii. must be declared in a method's throws clause: compile-time error, unless all thrown exceptions – are caught or declared (even those caused by called methods)

why checked exceptions?

i. Signal recoverable problems
ii. Force clients to deal with the problem

why unchecked exceptions?

i. Make exception handling
ii. Declaration feasible

Exception rethrown as unchecked exception (OR error)

```
abstract class Statement {
  public abstract void execute();
}

class InputStatement extends Statement {
  public void execute() {
    InputStream input = ...;
    try { input.read(); } // throws IOException
    catch(IOException e) { throw new Error(e); }
  }
}
```

```
assert l!=null; // Precondition
// note, do not use here assertTrue or c
for(int i=0;i<l.size();i++){
  if(elem.equals(l.get(i))){
    assert l.get(i).equals(elem);
    // possibly redundant, since the on
    return i;
  }
}
assert !l.contains(elem);// no i exis
```