WEB DEVELOPMENT & CORE JAVA LAB

Syllabus

Java programs using classes &objects and various control constructs such as loops etc, and structure such as arrays, structures and functions.

Java programs for creating applets for display of images, text and animation

Program related to interfaces and packages

Input output and random files programs in java

Java programs using event driven concept

Programs related to network programming

Rationale behind coverage

Java is an object-oriented programming language developed by Sun Microsystems in the early 1990s. Java applications are, in the official implementation, compiled to bytecode, which is compiled to native machine code at runtime. Sun Microsystems provides a GNU General Public License implementation of a Java compiler and Java virtual machine, in compliance with the specifications of the Java Community Process.

The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities. JavaScript, a scripting language, shares a similar name and has similar syntax, but is not related to Java

Applet

> *Main article: Java applet*

Java applets are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.

```
// Hello.java
import java.applet.Applet;
import java.awt.Graphics;

public class Hello extends Applet {
   public void paint(Graphics gc) {
      gc.drawString("Hello, world!", 65, 95);
   }
}
```

The **`import`** statements direct the Java compiler to include the **`java.applet.Applet`** and **`java.awt.Graphics`** classes in the compilation. The import statement allows these classes to be referenced in the source code using the *simple class name* (i.e. `Applet`) instead of the *fully qualified class name* (i.e. `java.applet.Applet`).

The **`Hello`** class **`extends`** (subclasses) the **`Applet`** class; the `Applet` class provides the framework for the host application to display and control the lifecycle of the applet. The `Applet` class is an Abstract Windowing Toolkit (AWT) `Component`, which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The `Hello` class overrides the **`paint(Graphics)`** method inherited from the `Container` superclass to provide the code to display the applet. The `paint()` method is passed a **`Graphics`** object that contains the graphic context used to display the applet. The

`paint()` method calls the graphic context **`drawString(String, int, int)`** method to display the **"Hello, world!"** string at a pixel offset of (**`65, 95`**) from the upper-left corner in the applet's display.

Software and hardware requirements

Software: jdk1.4

Hardware: PROCESSOR PIV
SPEED 1.8GHz
RAM 256MB
HDD 40GB

LIST OF PRACTICALS

1. Program to create a class Calculator that contains an overloaded method Called “add” to sum of two integers, two float and one integer and one float.

2. Program to create two classes “Chair” and “Wheelchair”. Create the function “adjust Height” calling from class “Chair” override this method in the class “Wheelchair” calling from Wheelchair.

3. Program to create interface BrightnessControl with two functions increase Brightness () and decreaseBrightness(). Now create two classes “TV” and “Monitor and implements interfaces in both classes.

4. Program creates a class call try catch and handle exception divide by zero and array index out of bound.

5. Program to create an Applet class.

6. Program to create the Package.

7. Program to create the MenuBar.

8. Program for array classes.

9. Program for Networking.

10. Program to illustrate the I/O.

Program1

```
/*Program to create a class Calculator that contains an overloaded
method called “add” to sum of two integers, two float and one
integer and one float.*/
class calc
{
public static void main(String arg[])
{
calc obj=new calc();
obj.add(15,25);
obj.add(1.5f,2.5f);
obj.add(10,2.5f);
}
int x,y;
float p,q,result;
void add(int a,int b)
{
x=a;
y=b;
result=x+y;
System.out.println("result="+result);
}
void add(float a,float b)
{
p=a;
q=b;
result=p+q;
System.out.println("result="+result);
}
void add(int a,float b)
{
x=a;
p=b;
result=x+p;
System.out.println("result="+result)}}
```

Program 2

/*Program to create two classes “Chair” and “Wheelchair”. Create the function “adjust Height” calling from class “Chair” override this method in the class “Wheelchair” calling from Wheelchair.*/

```
class chair2
{
        int height;
        void adjustheight()
        {
                height=4;
                system.outprintln("height of char=" + height);
        }
}
class wheelchair extends chair
{
        void adjustheight()
        {
`               height=6;
                system.out.println("height=" + height);
                super adjustheight();
        }
        public static void main(string ar[])
        {
                wheelchair w=new wheelchair();
                w.adjustheight();
        }
}
```

Program 3

/*Program to create interface BrightnessControl with two functions increaseBrightness () and decreaseBrightness(). Now create two classes “TV” and“Monitor and implements interfaces in both classes.*/

```
interface BC
{
public void increase_brightness();
public void decrease_brightness();
}
class TV implements BC
{
public void increase_brightness()
{
System.out.println("increase brightness");
}
public void decrease_brightness()
{
System.out.println("decrease brightness");
}
}
class monitor implements BC
{
public void increase_brightness()
{
System.out.println("increase brightness");
}
public void decrease_brightness()
{
System.out.println("decrease brightness");
}
}
class brightness
{
```

```
public static void main(String[] str)
{
TV obj1=new TV();
monitor obj2=new monitor();
obj1.increase_brightness();
obj1.decrease_brightness();
obj2.increase_brightness();
obj2.decrease_brightness();
}
}
```

Program 4

```
/*Program creates a class call try catch and handle exception
divide by zero and array index out of bound.*/
import java.*;
import java .lang.*;
class ExceptionExample
{
    public void main(String args[])
{
int arr[]={5,10};
int a=20,b=7,c=5;
int res=0;

try{
res=(a/b-c);
system.out.println("the array value is: "+arr[2]);
}

catch(ArithmeticException e)
{
System.out.println("divide by zero");
}

catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Unaccessible Element");
}

catch(ArithmeticException e)
{
System.out.println("divide by zero");
System.out.println("Unaccessible Element");
}
}}
```

Program 5

/*Program to create an Applet class.*\

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet.class" width=500 height=600>
</applet>
*/
public class SimpleApplet extends Applet
{
        public void paint(Graphics g)
        {
         g.drawString("A Simple Applet",10,100);
        }
}
```

Program 6

Program to create the Package.

```
package package;
public class c1
{
public void displayA()
{
System.out.println("class A");
}
}
```

```
import pack.c1
class packagetest
{
```

```
  public static void main(string arg[])
{
 c1.obj=new c1();
obj.displayA();
}
}
```

Program 7

Program to create the MenuBar.

```
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.applet.*;
/*<applet code="priya.class">
</applet>*/

public class priya extends JApplet
{
JFrame f1;
JMenuBar mb;
JMenu m1,m2;
JMenuItem t1,t2,t3,t4,t5,t6;
JTextArea ta;

void init()
{
```

```
f1=new JFrame("NOTEPAD");
mb=new JMenuBar();
m1=new JMenu("file");
m2=new JMenu("edit");
t1=new JMenuItem("new");
t2=new JMenuItem("open");
t3=new JMenuItem("save");
t4=new JMenuItem("undo");
t5=new JMenuItem("redo");
t6=new JMenuItem("exit");
f1.setJMenuBar(mb);
mb.add(m1);
mb.add(m2);
m1.add(t1);
m1.add(t2);
m1.add(t3);
m2.add(t4);
m2.add(t5);
m2.add(t6);
//f1.getContentPane().add(ta);
f1.getContentPane().add(ta);
f1.setSize(300,200);
f1.setVisible(true);


}}
public void paint(Graphics g)
{

}
```

Program 8

Program for array classes.

```
class array
{
	public static void main(String args[])
	{
		int month_days[];
		month_days=new int[12];
		month_days[0]=31;
		month_days[1]=28;
		month_days[2]=31;
		month_days[3]=30;
		month_days[4]=31;
		month_days[5]=30;
		month_days[6]=31;
		month_days[7]=31;
		month_days[8]=30;
		month_days[9]=31;
		month_days[10]=30;
		month_days[11]=31;

	}
  System.out.println("april has " + month_days[3] + "days");
}
```

Program 9

/*Program to get protocol, host and port no. of a web page(Program for Networking)*/

```
import java.lang.*;
import java.io.*;
import java.net.*;
 class UDL
{
 public static void main(String arg[]) throws
MalformedURLException
   {

   try
  {
URL ur1=new URL(  "http://www.yahoo.com ");
    System.out.println("host name is" + ur1.getHost());
    System.out.println("portno is" +ur1.getPort());
    System.out.println("Protocol is" +ur1.getProtocol());
  }
 catch(Exception e)
 {
  System.out.println("error" +e);
 }
}
}
```

Program 10

Program for copying a file into another file(illustrate the I/O)

```
import java.io.*;
class CopyFile
{
public static void main(String args[])
throws IOException
{
int i;
FileInputStream fin=null;
FileOutputStream fout=null;
try
{
fin=new FileInputStream("C:/Documents and Settings/user/My
Documents/a.txt");
}
catch(FileNotFoundException e)
{
System.out.println("Input File Not Found");
return;
}

try
{
fout=new FileOutputStream("C:/Documents and Settings/user/My
Documents/b.txt");
}
catch(FileNotFoundException e)
{
System.out.println("Error In Opening File");
return;
}
```

```
try
{
do
{
i=fin.read();
if(i!=-1)fout.write(i);
}
while(i!=-1);
}
catch(IOException e)
{
System.out.println("File Error");
}
fin.close();
fout.close();
}
}
```

STUDY ABOUT JAVA LANGUAGE

// Hello.java

```
import java.applet.Applet;
import java.awt.Graphics;

public class Hello extends Applet {
   public void paint(Graphics gc) {
      gc.drawString("Hello, world!", 65, 95);
   }
}
```

The **`import`** statements direct the Java compiler to include the **`java.applet.Applet`** and **`java.awt.Graphics`** classes in the compilation. The import statement allows these classes to be referenced in the source code using the *simple class name* (i.e. `Applet`) instead of the *fully qualified class name* (i.e. `java.applet.Applet`).

The **`Hello`** class **`extends`** (subclasses) the **`Applet`** class; the `Applet` class provides the framework for the host application to display and control the lifecycle of the applet. The `Applet` class is an Abstract Windowing Toolkit (AWT) `Component`, which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The `Hello` class overrides the **`paint(Graphics)`** method inherited from the `Container` superclass to provide the code to display the applet. The `paint()` method is passed a **`Graphics`** object that contains the graphic context used to display the applet. The `paint()` method calls the graphic context **`drawString(String, int, int)`** method to display the "**Hello, world!**" string at a pixel offset of (**`65, 95`**) from the upper-left corner in the applet's display.

```
<!-- Hello.html -->
<html>
  <head>
    <title>Hello World Applet</title>
  </head>
  <body>
    <applet code="Hello" width="200" height="200">
    </applet>
  </body>
</html>
```

An applet is placed in an HTML document using the **`<applet>`** HTML element. The `applet` tag has three attributes set: **`code="Hello"`** specifies the name of the `Applet` class and **`width="200" height="200"`** sets the pixel width and height of the applet. (Applets may also be embedded in HTML using either the `object` or `embed` element, although

support for these elements by Web browsers is inconsistent.[5][6]) However, the `applet` tag is deprecated, so the `object` tag is preferred where supported.

The host application, typically a Web browser, instantiates the **`Hello`** applet and creates an `AppletContext` for the applet. Once the applet has initialized itself, it is added to the AWT display hierarchy. The `paint` method is called by the AWT event dispatching thread whenever the display needs the applet to draw itself.

1. Drawing Lines
2. Drawing Other Stuff
3. Color - *introduces arrays*
4. Mouse Input - *introduces showStatus( ) and Vector*
5. Keyboard Input
6. Threads and Animation - *introduces System.out.println( )*
7. Backbuffers - *introduces Math.random( ) and Graphics.drawImage( )*
8. Painting
9. Clocks
10. Playing with Text - *introduces 2D arrays and hyperlinks*
11. 3D Graphics - *introduces classes*
12. Odds and Ends

****

Event handling

In computer programming, an **event handler** is an asynchronous callback subroutine that handles inputs received in a program. Each *event* is a piece of application-level information from the underlying framework, typically the GUI toolkit. GUI events include key presses, mouse movement, action selections, and timers expiring. On a lower level, events can represent availability of new data for reading a file or network stream. Event handlers are a central concept in event-driven programming.

The events are created by the framework based on interpreting lower-level inputs, which may be lower-level events themselves. For example, mouse movements and clicks are interpreted as menu selections. The events initially originate from actions on the operating system level, such as interrupts generated by hardware devices, software interrupt instructions, or state changes in polling. On this level, interrupt handlers and signal handlers correspond to event handlers.

Created events are first processed by an *event dispatcher* within the framework. It typically manages the associations between events and event handlers, and may queue event handlers or events for later processing. Event dispatchers may call event handlers directly, or wait for events to be dequeued with information about the handler to be executed.

Event-driven programming

Event-driven programming is a computer programming paradigm in which the flow of the program is determined by user actions (mouse clicks, key presses) or messages from other programs. In contrast, in *batch programming* the flow is determined by the programmer. Batch programming is the style taught in beginning programming classes while event driven programming is what is needed in any interactive program. Event driven programs can be written in any language although the task is easier in some languages than in others.

Mouse Input

The source code:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Mouse1 extends Applet
   implements MouseListener, MouseMotionListener {

   int width, height;
   int mx, my;  // the mouse coordinates
   boolean isButtonPressed = false;

   public void init() {
      width = getSize().width;
      height = getSize().height;
      setBackground( Color.black );

      mx = width/2;
      my = height/2;

      addMouseListener( this );
      addMouseMotionListener( this );
   }

   public void mouseEntered( MouseEvent e ) {
      // called when the pointer enters the applet's rectangular area
   }
   public void mouseExited( MouseEvent e ) {
      // called when the pointer leaves the applet's rectangular area
   }
   public void mouseClicked( MouseEvent e ) {
      // called after a press and release of a mouse button
      // with no motion in between
      // (If the user presses, drags, and then releases, there will be
      // no click event generated.)
   }
   public void mousePressed( MouseEvent e ) {  // called after a button
is pressed down
```

```
      isButtonPressed = true;
      setBackground( Color.gray );
      repaint();
      // "Consume" the event so it won't be processed in the
      // default manner by the source which generated it.
      e.consume();
   }
   public void mouseReleased( MouseEvent e ) {  // called after a
button is released
      isButtonPressed = false;
      setBackground( Color.black );
      repaint();
      e.consume();
   }
   public void mouseMoved( MouseEvent e ) {  // called during motion
when no buttons are down
      mx = e.getX();
      my = e.getY();
      showStatus( "Mouse at (" + mx + "," + my + ")" );
      repaint();
      e.consume();
   }
   public void mouseDragged( MouseEvent e ) {  // called during motion
with buttons down
      mx = e.getX();
      my = e.getY();
      showStatus( "Mouse at (" + mx + "," + my + ")" );
      repaint();
      e.consume();
   }

   public void paint( Graphics g ) {
      if ( isButtonPressed ) {
         g.setColor( Color.black );
      }
      else {
         g.setColor( Color.gray );
      }
      g.fillRect( mx-20, my-20, 40, 40 );
   }
}
```

Try clicking and dragging on the resulting applet. Notice how the status bar in your web browser displays the current mouse position -- that's due to the calls to `showStatus()`. (You might see some occasional flickering in this applet. This problem will be addressed in an upcoming lesson.)

****

An **interface** in the Java programming language is an abstract type which is used to specify an interface (in the generic sense of the term) that classes must implement.

Interfaces are introduced with the **`interface`** keyword, and may only contain function signatures and constant declarations (variable declarations which are declared to be both `static` and `final`).

As interfaces are *`abstract`*, they cannot be instantiated. Object references in Java may be specified to be of interface type; in which case they must be bound to null, or an object which *`implements`* the interface.

The primary capability which interfaces have, and classes lack, is multiple inheritance. All classes in Java (other than `java.lang.Object`, the root class of the Java type system) must have exactly one base class (corresponding to the `extends` clause in the class definition; classes without an extends clause are defined to inherit from `Object`); multiple inheritance of classes is not allowed. However, Java classes may *implement* as many interfaces as the programmer desires (with the `implements` clause). A Java class which implements an interface, but which fails to implement all the methods specified in the interface, becomes an **abstract base class**, and must be declared `abstract` in the class definition.

Defining an Interface

Interfaces must be defined using the following formula (compare to Java's class definition).

```
[visibility] interface Interface Name [extends other interfaces] {
        constant declarations
        abstract method declarations
}
```

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the `abstract` keyword is not required.

Thus, a simple interface may be

```
public interface Predator {
       public boolean chasePrey(Prey p);
       public void eatPrey(Prey p);
}
```

Implementing an Interface

The syntax for implementing an interface uses this formula:

```
... implements interface name[, another interface, another, ...] ...
```

Classes may implement an interface. For example,

```
public class Cat implements Predator {

        public boolean chasePrey(Prey p) {
               // programming to chase prey p
        }

        public void eatPrey (Prey p) {
               // programming to eat prey p
        }
}
```

If a class implements an interface and is not abstract, and does not implement a required interface, this will result in a compiler error. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.

Classes can implement multiple interfaces

```
public class Frog implements Predator, Prey { ... }
```

Creating subinterfaces

Subinterfaces can be created as easily as interfaces, using the same formula are described above. For example

```
public interface VenomousPredator extends Predator, Venomous {
        interface body
}
```

is legal. Note how it allows multiple inheritance, unlike classes.

****

A **Java package** is a mechanism for organizing Java classes into namespaces. Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality.

Java source files can include a **`package`** statement at the top of the file to designate the package for the classes the source file defines.

1. A package provides a unique namespace for the types it contains.
2. Classes in the same package can access each other's protected members.
3. A package can contain the following kinds of types.

Classes Interfaces Enumerated types Annotations

Using packages

In Java source files, the package that the file belongs to is specified with the `package` keyword.

```
package java.awt.event;
```

To use a package inside a Java source file, it is convenient to import the classes from the package with an `import` statement. The statement

```
import java.awt.event.*;
```

imports all classes from the `java.awt.event` package, while

```
import java.awt.event.ActionEvent;
```

imports only the `ActionEvent` class from the package. After either of these import statements, the `ActionEvent` class can be referenced using its simple class name:

```
ActionEvent myEvent = new ActionEvent();
```

Classes can also be used directly without an import statement by using the fully-qualified name of the class. For example,

```
java.awt.event.ActionEvent myEvent = new java.awt.event.ActionEvent();
```

doesn't require a preceding import statement.

Package access protection

Classes within a package can access classes and members declared with *default access* and class members declared with the *`protected`* access modifier. Default access is enforced when neither the `public`, `protected` nor `private` access modifier is specified in the declaration. By contrast, classes in other packages cannot access classes and members declared with default access. Class members declared as `protected` can only be accessed from within classes in other packages that are subclasses of the declaring cla AR file being created.

Package naming conventions

Packages are usually defined using a hierarchical naming pattern, with levels in the hierarchy separated by periods (.) (pronounced "dot"). Although packages lower in the naming hierarchy are often referred to a "subpackages" of the corresponding packages higher in the hierarchy, there is no semantic relationship between packages. The Java Language Specification establishes package naming conventions in order to avoid the possibility of two published packages having the same name. The naming conventions describe how to create unique package names, so that packages that are widely distributed will have unique namespaces. This allows packages to be easily and automatically installed and catalogued.

In general, a package name begins with the top level domain name of the organization and then the organization's domain and then any subdomains listed in reverse order. The organization can then choose a specific name for their package. Package names should be all lowercase characters whenever possible.

For example, if an organization in Canada called MySoft creates a package to deal with fractions, naming the package `ca.mysoft.fractions` distinguishes the fractions package from another similar package created by another company. If a US company named MySoft also creates a fractions package, but names it `com.mysoft.fractions`, then the classes in these two packages are defined in a unique and separate namespace.

****

Arrays

1. Java has array types for each type, including arrays of primitive types, class andinterface types, as well as higher-dimensional arrays of array types.
2. All elements of an array must descend from the same type.
3. All array classes descend from the class `java.lang.Object`, and mirror the hierarchy of the types they contain.
4. Array objects have a read-only `length` attribute that contains the number of elements in the array.
5. Arrays are allocated at runtime, so the specified size in an array creation expression may be a variable (rather than a constant expression as in C).
6. Java arrays have a single dimension. Multi-dimensional arrays are supported by the language, but are treated as arrays of arrays.

```
// Declare the array - name is "myArray", element type is references to
"SomeClass"
SomeClass[] myArray = null;
// Allocate the array
myArray = new SomeClass[10];
// Or Combine the declaration and array creation
```

```
SomeClass[] myArray = new SomeClass[10];
// Allocate the elements of the array (not needed for simple data
types)
for (int i = 0; i < myArray.length; i++)
    myArray[i] = new SomeClass();


**********************8
```

For loop

```
for (initial-expr; cond-expr; incr-expr) {
    statements;
}
```

For-each loop

J2SE 5.0 added a new feature called the for-each loop, which greatly simplifies the task of iterating through every element in a collection. Without the loop, iterating over a collection would require explicitly declaring an iterator:

```
public int sumLength(Set<String> stringSet) {
  int sum = 0;
  Iterator<String> itr = stringSet.iterator();
  while (itr.hasNext())
    sum += itr.next().length();
  return sum;
}
```

The for-each loop greatly simplifies this method:

```
public int sumLength(Set<String> stringSet) {
  int sum = 0;
  for (String s : stringSet)
    sum += s.length();
  return sum
}
**************************
```

Objects

Classes

Java has *nested* classes that are declared within the body of another class or interface. A class that is not a nested class is called a *top level* class. An *inner class* is a non-static nested class.

Classes can be declared with the following modifiers:

`abstract` – cannot be instantiated. Only interfaces and `abstract` classes may contain `abstract` methods. A concrete (non-`abstract`) subclass that extends an `abstract` class must override any inherited `abstract` methods with non-`abstract` methods. Cannot be `final`.

`final` – cannot be subclassed. All methods in a final class are implicity `final`. Cannot be `abstract`.

`strictfp` – all floating-point operations within the class and any enclosed nested classes use strict floating-point semantics. Strict floating-point semantics guarantee that floating-point operations produce the same results on all platforms.

Note that Java classes do not need to be terminated by a semicolon (";"), which is required in C++ syntax.

Method overloading is a feature found in various object oriented programming languages such as C++ and Java that allows the creation of several functions with the same name which differ from each other in terms of the type of the input and the type of the output of the function.

An example of this would be a square function which takes a number and returns the square of that number. In this case, it is often necessary to create different functions for integer and floating point numbers.

Method overloading is usually associated with statically-typed programming languages which enforce type checking in function calls. When overloading a method, you are really just making a number of different methods that happen to have the same name. It is resolved at compile time which of these methods are used.

Method overloading should not be confused with ad-hoc polymorphism or virtual functions. In those, the correct method is chosen at runtime.

**********8

Method overriding, in object oriented programming, is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one

of its superclasses. The implementation in the subclass overrides (replaces) the implementation in the superclass.

A subclass can give its own definition of methods which also happen to have the same signature as the method in its superclass. This means that the subclass's method has the same name and parameter list as the superclass's overridden method. Constraints on the similarity of return type vary from language to language, as some languages support covariance on return types.

Method overriding is an important feature that facilitates polymorphism in the design of object-oriented programs.

Some languages allow the programmer to prevent a method from being overridden, or disallow method overriding in certain core classes. This may or may not involve an inability to subclass from a given class.

In many cases, abstract classes are designed — i.e. classes that exist only in order to have specialized subclasses derived from them. Such abstract classes have methods that do not perform any useful operations and are meant to be overridden by specific implementations in the subclasses. Thus, the abstract superclass defines a common interface which all the subclasses inherit.

Examples

This is an example in Python. First a general class ("Person") is defined. The "self" argument refers to the instance object. The Person object can be in one of three states, and can also "talk".

```
class Person:
    def __init__(self):
        self.state = 0

    def talk(self, sentence):
        print sentence

    def lie_down(self):
        self.state = 0

    def sit_still(self):
        self.state = 1

    def stand(self):
        self.state = 2
```

Then a "Baby" class is defined (subclassed from Person). Objects of this class cannot talk or change state, so exceptions (error conditions) are raised by all methods except "lie_down". This is done by overriding the methods "talk", "sit_still" and "stand"

```
class Baby(Person):
```

```
    def talk(self, sentence):
        raise CannotSpeakError, 'This person cannot speak.'

    def sit_still(self):
        raise CannotSitError, 'This person cannot sit still.'

    def stand(self):
        raise CannotStandError, 'This person cannot stand up.'
```

Many more methods could be added to "Person", which can also be subclassed as "MalePerson" and "FemalePerson", for example. Subclasses of Person could then be grouped together in a data structure (a list or array), and the same methods could be called for each of them regardless of the actual class; each object would respond appropriately with its own implementation or, if it does not have one, with the implementation in the superclass.

Exception handling

Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of some condition that changes the normal flow of execution. The condition is called an **exception**. Alternative concepts are signal and event handler.

In general, current state will be saved in a predefined location and execution will switch to a predefined handler. Depending on the situation, the handler may later resume the execution at the original location, using the saved information to restore the original state. For example, an exception which will usually be resumed is a page fault, while a division by zero usually cannot be resolved transparently.

From the processing point of view, hardware interrupts are similar to resumable exceptions, except they are usually not related to the current program flow.

Exception safety

A piece of code is said to be **exception-safe** if run-time failures within the code will not produce ill-effects, such as memory leaks, garbled data or invalid output. Exception-safe code must satisfy invariants placed on the code even if exceptions occur. There are several levels of exception safety:

- **failure transparency**, operations are guaranteed to succeed and satisfy all requirements even in presence of exceptional situations. (best)
- **commit or rollback semantics**, operations can fail, but failed operations are guaranteed to have no side effects.

- **basic exception safety**, partial execution of failed operations can cause side effects, but invariants on the state are preserved (that is, any stored data will contain valid values).
- **minimal exception safety**, partial execution of failed operations may store invalid data but will not cause a crash.
- **no exception safety**, no guarantees are made. (worst)

Usually at least basic exception safety is required. Failure transparency is difficult to implement, and is usually not possible in libraries where complete knowledge of the application is not available.

Exception support in programming languages

Exception handling syntax

Many computer languages, such as Ada, C++, Common Lisp, D, Delphi, Eiffel, Java, Objective-C, Ocaml, PHP (as of version 5), Python, REALbasic, ML, Ruby, and all .NET languages have built-in support for exceptions and exception handling. In those languages, the advent of an exception (more precisely, an exception handled by the language) unwinds the stack of function calls until an exception handler is found. That is, if function *f* contains a handler *H* for exception *E*, calls function *g*, which in turn calls function *h*, and an exception *E* occurs in *h*, then functions *h* and *g* will be terminated and *H* in *f* will handle *E*.

Excluding minor syntactic differences, there are only a couple of exception handling styles in use. In the most popular style, an exception is initiated by a special statement (`throw`, or `raise`) with an exception object. The scope for exception handlers starts with a marker clause (`try`, or the language's block starter such as `begin`) and ends in the start of the first handler clause (`catch`, `except`, `rescue`). Several handler clauses can follow, and each can specify which exception classes it handles and what name it uses for the exception object.

A few languages also permit a clause (`else`) that is used in case no exception occurred before the end of the handler's scope was reached. More common is a related clause (`finally`, or `ensure`) that is executed whether an exception occurred or not, typically to release resources acquired within the body of the exception-handling block. Notably, C++ lacks this clause, and the Resource Acquisition Is Initialization technique is used to free such resources instead.

In its whole, exception handling code might look like this (in pseudocode):

```
try {
 line = console.readLine();
 if (line.length() == 0) {
  throw new EmptyLineException("The line read from console was
empty!");
 }
 console.printLine("Hello %s!" % line);
```

```
} catch (EmptyLineException e) {
 console.printLine("Hello!");
} catch (Exception e) {
 console.printLine("Error: " + e.message());
} else {
 console.printLine("The program ran successfully");
} finally {
 console.printLine("The program terminates now");
}
```

As a minor variation, some languages use a single handler clause, which deals with the class of the exception internally.

Checked exceptions

The designers of Java devised[1][2] ***checked exceptions***[3] which are a special set of exceptions. The checked exceptions that a method may raise constitute part of the type of the method. For instance, if a method might throw an `IOException` instead of returning successfully, it must declare this fact in its method header. Failure to do so raises a compile-time error.

This is related to exception checkers that exist at least for OCaml. The external tool for OCaml is both transparent (i.e. it does not require any syntactic annotations) and facultative (i.e. it is possible to compile and run a program without having checked the exceptions, although this is not suggested for production code).

The CLU programming language had a feature with the interface closer to what Java has introduced later. A function could raise only exceptions listed in its type, but any leaking exceptions from called functions would automatically be turned into the sole runtime exception, `failure`, instead of resulting in compile-time error. Later, Modula-3 had a similar feature.[4] These features don't include the compile time checking which is central in the concept of checked exceptions and hasn't as of 2006 been incorporated into other major programming languages than Java.[5]

Pros and cons

Checked exceptions can, at compile time, greatly reduce (but not entirely eliminate) the incidence of unhandled exceptions surfacing at runtime in a given application; the unchecked exceptions (`RuntimeExceptions` and `Errors`) can still go unhandled.

However, some see checked exceptions as a nuisance, syntactic salt that either requires large `throws` declarations, often revealing implementation details and reducing encapsulation, or encourages the (ab)use of poorly-considered `try/catch` blocks that can potentially hide legitimate exceptions from their appropriate handlers.

Others do not consider this a nuisance as you can reduce the number of declared exceptions by either declaring a superclass of all potentionally thrown exceptions or by defining and declaring exception types that are suitable for the level of abstraction of the

called method, and mapping lower level exceptions to these types, preferably wrapped using the exception chaining in order to preserve the root cause.

A simple `throws Exception` declaration or `catch (Exception e)` is always sufficient to satisfy the checking. While this technique is sometimes useful, it effectively circumvents the checked exception mechanism, so it should only be used after careful consideration.

One prevalent view is that unchecked exception types should not be handled, except maybe at the outermost levels of scope, as they often represent scenarios that do not allow for recovery: `RuntimeExceptions` frequently reflect programming defects[7], and `Errors` generally represent unrecovereable JVM failures. The view is that, even in a language that supports checked exceptions, there are cases where the use of checked exceptions is not appropriate.

Questions and Exercises: Object-Oriented Programming Concepts

Questions

- Real-world objects contain ___ and ___.
- A software object's state is stored in ___.
- A software object's behavior is exposed through ___.
- Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data ___.
- A blueprint for a software object is called a ___.
- Common behavior can be defined in a ___ and inherited into a ___ using the ___ keyword.
- A collection of methods with no implementation is called an ___.
- A namespace that organizes classes and interfaces by functionality is called a ___.
- The term API stands for ___?

FAQ

Questions and Exercises: Object-Oriented Programming Concepts

Questions

- Real-world objects contain ___ and ___.
- A software object's state is stored in ___.
- A software object's behavior is exposed through ___.
- Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data ___.
- A blueprint for a software object is called a ___.
- Common behavior can be defined in a ___ and inherited into a ___ using the ___ keyword.
- A collection of methods with no implementation is called an ___.
- A namespace that organizes classes and interfaces by functionality is called a ___.
- The term API stands for ___?